VisualDSP++® 5.0 Device Drivers and System Services Manual for Blackfin® Processors

Revision 4.3, January 2011

Part Number
82-000430-01

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

ANALOG
DEVICES

CONTENTS

EXTERNAL BUS INTERFACE UNIT MODULE

PREFACE

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

Purpose of This Manual

The *VisualDSP++ 5.0 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 the architectural descriptions of the device driver design and each system service component. Also included is a description of the API calls into each library.

Intended Audience

The primary audience for this manual is a programmer who is familiar with Analog Devices Blackfin® processors. This manual assumes 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 hardware reference and programming reference manuals, that describe their 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 enables 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, "Programmable Flag Service"
  Describes the programmable flag service that provides interface into the programmable flag subsystem of the Blackfin processor.
- Chapter 8, "Timer Service"
  Describes the timer service that provides interface into the core, watchdog, and general-purpose timers of the Blackfin processor.

- Chapter 9, "Port Control Service"
  Describes the port control manager service used to assign the programmable flag pins to various functions.
- Chapter 10, "Device Driver Manager"
  Describes the device driver model used to control devices, both internal and external, to ADI processors.
- Chapter 11, "Real-Time Clock Service"
  Describes the real-time clock service within the system services library and how to use it to enable the features of the real-time clock on Blackfin processors.
- Chapter 12, "File System Service"
  Describes the file system service (FSS), which provides access to mass storage media from the Blackfin processor.
- Chapter 13, "Pulse-Width Modulation"
  Describes the basic features of the pulse-width modulation (PWM) service and the use of this system service in software applications.
- Chapter 14, "STDIO Service "
  Describes the STDIO service that supports the redirection of STDIO streams to different output peripherals.

What's New in This Manual

This revision (4.3) of the manual documents changes/additions related to device drivers and system services for VisualDSP++® 5.0 and subsequent updates (up to update 9). These changes include:

- Deleted Chapter 14, "Memory Manager Service", as this service has been removed from the tools.
- Incorporated modifications and corrections based on errata reports against the previous revision (4.2) of the manual.

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/technical_support`
- E-mail tools questions to:
  `processor.tools.support@analog.com`
- E-mail processor questions to:
  `processor.support@analog.com (World wide support)`
  `processor.europe@analog.com (Europe support)`
  `processor.china@analog.com (China support)`
- Phone questions to **1-800-ANALOGD**
- Contact your Analog Devices, Inc. local sales office or authorized distributor

Supported Processors

The name *Blackfin* refers to a family of 16-bit, embedded processors. VisualDSP++ supports all ADSP-BFxxx Blackfin processors.

For a complete list of processors supported by VisualDSP++ 5.0, refer to the online Help.

Product Information

Product information can be obtained from the Analog Devices Web site, VisualDSP++ online Help system, and a technical library CD.

Analog Devices Web Site

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

To access a complete technical library for each processor family, go to `http://www.analog.com/processors/technical_library`. The manuals selection opens a list of current manuals related to the product as well as a link to the previous revisions of the manuals. When locating your manual title, note a possible errata check mark next to the title that leads to the current correction report against the manual.

Also note, `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 about products you are interested in. You can choose to receive weekly e-mail notifications containing updates to the Web pages that meet your interests, including documentation errata against all manuals. `MyAnalog.com` provides access to books, application notes, data sheets, code examples, and more.

Visit `MyAnalog.com` to sign up. If you are a registered user, just log on. Your user name is your e-mail address.

VisualDSP++ Online Documentation

Online documentation comprises the VisualDSP++ Help system, software tools manuals, hardware tools manuals, processor manuals, Dinkum Abridged C++ library, and FLEXnet License Tools software documentation. You can search easily across the entire VisualDSP++ documentation set for any topic of interest.

For easy printing, supplementary Portable Documentation Format (`.pdf`) files for all manuals are provided on the VisualDSP++ installation CD.

Each documentation file type is described as follows.

File	Description
`.chm`	Help system files and manuals in Microsoft Help format
`.htm` or `.html`	Dinkum Abridged C++ library and FLEXnet License Tools software documentation. Viewing and printing the `.html` files requires a browser, such as Internet Explorer 6.0 (or higher).
`.pdf`	VisualDSP++ and processor manuals in (PDF) format. Viewing and printing the `.pdf` files requires a PDF reader, such as Adobe Acrobat Reader (4.0 or higher).

Technical Library CD

The technical library CD contains seminar materials, product highlights, a selection guide, and documentation files of processor manuals, VisualDSP++ software manuals, and hardware tools manuals for the following processor families: Blackfin, SHARC®, TigerSHARC®, ADSP-218x, and ADSP-219x.

To order the technical library CD, go to `http://www.analog.com/processors/technical_library`, navigate to the manuals page for your processor, click the request CD check mark, and fill out the order form.

Data sheets, which can be downloaded from the Analog Devices Web site, change rapidly, and therefore are not included on the technical library CD. Technical manuals change periodically. Check the Web site for the latest manual revisions and associated documentation errata.

EngineerZone

EngineerZone is a technical support forum from Analog Devices. It allows you direct access to ADI technical support engineers. You can search FAQs and technical information to get quick answers to your embedded processing and DSP design questions.

Use EngineerZone to connect with other DSP developers who face similar design challenges. You can also use this open forum to share knowledge and collaborate with the ADI support team and your peers. Visit `http://ez.analog.com` to sign up.

Social Networking Web Sites

You can now follow Analog Devices Blackfin development on Twitter and LinkedIn. To access:

- Twitter: `http://twitter.com/blackfin`
- LinkedIn: Network with the LinkedIn group, Analog Devices Blackfin: `http://www.linkedin.com`

Notation Conventions

Text conventions used in this manual are identified and described as follows. Note that additional conventions, which apply only to specific chapters, may appear throughout this document.

Example	Description
`{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.

Example	Description
ⓘ	**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.

1 INTRODUCTION

This manual describes the system services and device driver architecture for Analog Devices embedded 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, allowing you to get 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 the development of new device drivers very straightforward.

At the time of this release, the system services and device drivers are available for use with the following Blackfin processors:

- ADSP-BF504/504F/506F
- ADSP-BF512/514/516/518
- ADSP-BF522/524/526
- ADSP-BF523/525/527

- ADSP-BF531/532/533
- ADSP-BF534/536/537
- ADSP-BF538/539
- ADSP-BF542/544/547/548/549
- ADSP-BF542M/544M/547M/548M/549M
- ADSP-BF561
- ADSP-BF590/592-A

For a complete list of processors supported by VisualDSP++ 5.0, refer to the online Help.

This chapter contains:

- "System Services Overview"
- "Device Driver Overview" on page 1-22

System Services Overview

The system services overview covers the following topics:

- "General" on page 1-3
- "Application Interface" on page 1-8
- "Dependencies" on page 1-10
- "Initialization" on page 1-11
- "Termination" on page 1-12
- "System Services Directory and File Structure" on page 1-13

General

The current revision of the system services library consists of the following 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 a Blackfin processor. Specific functionality allows the application to:
    - Set core and system clock operating frequencies with 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 you 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-KIT Lite® platforms.
- **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 a 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 linear and two-dimensional transfers on DMA channels.
- Enable the DMA manager to loopback on descriptor chains automatically.
- Stream data continuously into or out of a memory stream or peripheral.
- Initiate linear and two-dimensional memory DMA transfers with simple C-like, `memcpy`-type functions.

- **Programmable Flag Service** – The programmable flag service provides a simple interface into the programmable flags, sometimes called general-purpose I/O, of the Blackfin processor. This allows the application to access and control the programmable flags through a clean and consistent interface. The programmable flag service allows the application to:
    - Configure the direction, either input or output, of any flag.
    - Set, clear, and toggle the value of all output flags.

    - Sense the value of input flags.
    - Install callbacks, including live and deferred callbacks when specific trigger conditions occur on a flag.

- **Timer Service** – The timer service provides applications, drivers, and services with a simple mechanism to control general-purpose, core, and watchdog timers of the Blackfin processor. The timer service allows the application to:
    - Configure and control any timer within the processor, including general-purpose, core, and watchdog timers.
    - Install callbacks, including both "live" and deferred callbacks, when timers expire or trigger.
- **Port Control Service** – The port control service configures the pin multiplexing hardware appropriately to ensure proper operation of the peripherals that share common input and output pins. All system services and device drivers automatically make the appropriate calls into the port control service to seamlessly configure the pin muxing hardware without any end-user or application interaction, other than initialization of the service.
- **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.

- **Real-Time Clock Service** – (not available for the ADSP-BF561) The real-time clock service reads and writes the date and time, and installs callbacks for the various real-time clock events. The service includes the following features:
  - Set date and time.
  - Read date and time.
  - Set and read of epoch time.
  - Callbacks for the once only alarm, each day alarm, stop-watch event, one second event, one minute event, one hour event, one day event and register write complete event.
  - Reset the stopwatch.
  - Enable or disable the RTC wakeup to the processor.
- **File System Service** – The file system service provides access to embedded mass storage media, from the Blackfin processor. The file system includes functionality for:
  - File operations (open, close, read, write, seek, tell, IsEOF, remove, rename)
  - Directory operations (open, close, read, seek, tell, rewind, change, get current, create, remove)
  - Other operations (get file/directory status, get number of volumes, get volume info, format volume, media change notification, poll media, register/deregister device)
  - POSIX operations (opendir, closedir, readdir, readdir_r, rewinddir, seekdir, telldir, rename, mkdir, rmdir, remove)
  - Extensibility (additional or replacement drivers can be inserted)

- **Pulse-Width Modulation Service** – (only available on some newer Blackfin processors)
  The pulse-width modulation service facilitates control over the PWM hardware, generating waveforms to drive a three-phase voltage source inverter for use in motor control applications. The service requires the application to select the port muxing, synchronization pulse period and width, dead time, duty cycle, channel enable status, polarity, and operating mode. The service allows optional selection of sync pulse on an output pin, internal or external sync pulse, synchronous or asynchronous external sync pulse, IVG levels for trip and sync interrupt, switch reluctance mode, channel crossover mode, gate chopping mode, and trip input signal disable.

Application Interface

Each system service exports an application programming interface (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 can 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 is independent of the other services; however, redundancies are eliminated by allowing one service to access the functionality of another service.

For example, does 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 involves 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.

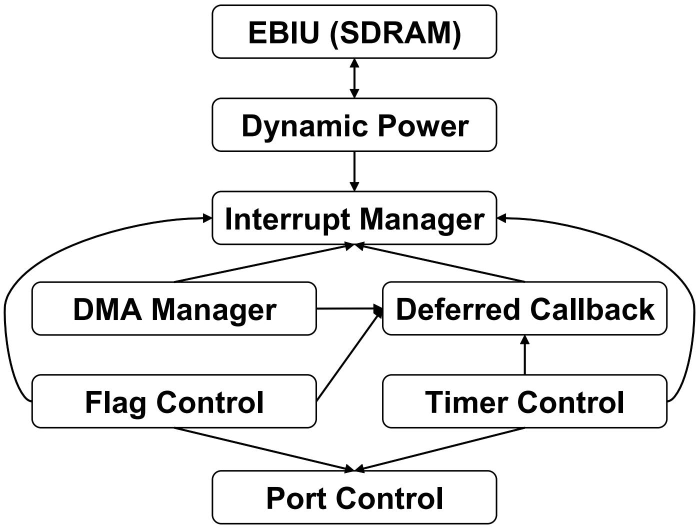

Figure 1-1. System Services and API Interactions

Dependencies

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

All current services, except the EBIU service, invoke the interrupt control service for the management of interrupt processing. The DMA manager, deferred callback, and power management services each depend on the interrupt control 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. However, when configured for live callbacks, the DMA manager does not use 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. Because each service is built as its own object file within the system services library file, you can further reduce the code size of the final executable by commanding the linker to eliminate unused objects. Refer to the development toolset documentation for more information.

Initialization

Some system services rely on other system services; thus, there is a preferred initialization sequence. Usually it is preferable to initialize all services at one time, typically when the whole system is being initialized, rather than spreading out the initialization of various services at different times.

Most applications find the initialization sequence listed below to be optimal. Any service in the sequence that is not used by the application can simply be omitted from the sequence.

1. Interrupt control service
2. External bus interface unit
3. Power management service
4. Port control (if applicable)
5. Deferred callback service
6. DMA manager service
7. Programmable flag service
8. Timer service
9. Real-time clock service
10. Semaphore service

Termination

Many embedded systems operate continuously in an endless loop and may never need to call the termination function of a service. Applications that do not have a need to terminate a service can save memory by never calling the termination function.

For applications that need to terminate services, as with the initialization sequence, there is a preferred sequence of terminating the services.

Most application find the termination sequence listed below to be optimal. Services are usually terminated in the reverse order from which they were initialized. Any service in the sequence that is not used by the application can simply be omitted from the sequence.

1. PWM service (if applicable)
2. Semaphore service
3. Real-time clock service
4. Timer service
5. Programmable flag service
6. DMA manager service
7. Deferred callback service
8. Port control (if applicable)
9. Power management service
10. External bus interface unit
11. Interrupt control service

System Services Directory and File Structure

All files for the system services are contained within the `Blackfin` directory tree. In VisualDSP++ installations, this directory is 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 system services should include the `Blackfin/include/services` directory in the (compiler and/or assembler) preprocessor search path. User source files that access 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 the 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 must be aware of the specific processor variant being targeted. You must 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 current processor variants are shown in Table 1-1, but new defines will be created for each newly-introduced Blackfin processor so reference to the latest include file is essential.

Table 1-1. Processor Variants

`__ADSPBF504__`	The ADSP-BF504 processor
`__ADSPBF504F__`	The ADSP-BF504F processor
`__ADSPBF506F__`	The ADSP-BF506F processor
`__ADSPBF512__`	The ADSP-BF512 processor
`__ADSPBF514__`	The ADSP-BF514 processor
`__ADSPBF516__`	The ADSP-BF516 processor
`__ADSPBF518__`	The ADSP-BF518 processor
`__ADSPBF522__`	The ADSP-BF522 processor
`__ADSPBF523__`	The ADSP-BF523 processor
`__ADSPBF524__`	The ADSP-BF524 processor
`__ADSPBF525__`	The ADSP-BF525 processor
`__ADSPBF526__`	The ADSP-BF526 processor
`__ADSPBF527__`	The ADSP-BF527 processor
`__ADSPBF531__`	The ADSP-BF531 processor
`__ADSPBF532__`	The ADSP-BF532 processor
`__ADSPBF533__`	The ADSP-BF533 processor
`__ADSPBF534__`	The ADSP-BF534 processor
`__ADSPBF535__`	The ADSP-BF535 processor
`__ADSPBF536__`	The ADSP-BF536 processor

Table 1-1. Processor Variants (Cont'd)

`__ADSPBF537__`	The ADSP-BF537 processor
`__ADSPBF538__`	The ADSP-BF538 processor
`__ADSPBF539__`	The ADSP-BF539 processor
`__ADSPBF542__`	The ADSP-BF542 processor
`__ADSPBF544__`	The ADSP-BF544 processor
`__ADSPBF547__`	The ADSP-BF547 processor
`__ADSPBF548__`	The ADSP-BF548 processor
`__ADSPBF549__`	The ADSP-BF549 processor
`__ADSPBF542M__`	The ADSP-BF542M processor
`__ADSPBF544M__`	The ADSP-BF544M processor
`__ADSPBF547M__`	The ADSP-BF547M processor
`__ADSPBF548M__`	The ADSP-BF548M processor
`__ADSPBF549M__`	The ADSP-BF549M processor
`__ADSPBF561__`	The ADSP-BF561 processor
`__ADSPBF590__`	The ADSP-BF590 processor
`__ADSPBF592__`	The ADSP-BF592-A processor

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

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 a ADSP-BF533 processor differs from the number of DMA channels for a ADSP-BF561 processor. 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 Overview" 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. This directory provides a system services library file for each processor variant and processor revision that is supported. You 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_yyyz.dlb` where:

- `xxx` represents the processor variant – This is typically a three-digit number that identifies the processor variant, such as `532` for the ADSP-BF532 processor, `534` for the ADSP-BF534 processor, and so on.
- `_yyy` represents the operating environment – This suffix represents the targeted operating environment, such as `vdk` for VDK-based systems, `uCOS` for uCOS-based systems, and so on. Libraries built for `standalone`, specifically non-RTOS environments, do not include the `_yyy` suffix.
- `z` represents any special conditions for the library – The following combinations are used:
    - `y` – The library is built to avoid all known anomalies for all revisions of silicon.
    - blank – A library without any additional suffix 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.

One system services library file only should be included for the linker to process. Choose the correct library based on the processor variant, operating environment, and processor revision number for your system.

For example, an application targeting silicon revision 0.2 of the ADSP-BF532 processor without an RTOS should link with the `libss1532.dlb` file from the `Blackfin/lib/bf532_rev_0.2` subdirectory. As another example, an application developer who wants a version of the system services library to run on any revision of ADSP-BF532 silicon and uses the VDK, should link with the `libss1532_vdky.dlb` file from the `Blackfin/lib` directory.

It is strongly recommended you use the debug versions of the system services library during development because the built-in error-checking code within the library can save countless hours of development time.

Specify the use of debug versions of the libraries by selecting **Use Debug System libraries** on the **Link:Processor** page of the **Project Options** dialog box.

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 developers with 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. This directory also contains 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:

1. Set the preprocessor include path to include `Blackfin/include/services` and `blackfin/lib/src/services.`
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. Refer to the description of the `_si_revision` switch in your processor's *C/C++ Compiler and Library* manual 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. Refer to these examples for additional information on how to use the system services effectively.

Dual-Core Considerations

For information on how to use the system services on dual-core ADSP-BF561 processors, see "Dual-Core Considerations" on page 3-5.

RTOS Considerations

Deployment of system services and the device driver model within an application based around an RTOS, such as VDK, is highly recommended. However, observe these considerations to avoid conflict with the RTOS and to successfully deploy the system services and device drivers within a multi-threaded application.

The following discussion, which is limited to VDK, is also relevant to other RTOS environments.

Interoperability of System Services With VDK

There are three major considerations to keep in mind when deploying system services and the device driver model within a VDK-based application.

- Interrupt handling – The interrupt manager is a cornerstone of the system services and the device driver model. The interrupt manager is designed to manage only the interrupt vector groups (IVG) that it is requested to manage, as dictated by each call to `adi_int_CECHook()`, leaving the other IVG levels to be handled as per the user's requirements. Thus, VDK-managed interrupts can easily coexist alongside those managed by system services, provided that neither method manages the same IVG levels as the other. It is not possible to have a VDK ISR and an interrupt manager chain assigned to the same IVG level, as one will overwrite the other in the event vector table (EVT).

  All DMA channels and device drivers use the default IVG levels as defined in the `SIC_IARx` registers at the time of device initialization (that is, during the call to `adi_dev_Open()`).

- Prohibited interrupt levels – Appendix A of the *VisualDSP++ 5.0 Kernel (VDK) User's Guide* details four interrupt levels [EVT3 (EVX), EVT6 (IVTMR), IVG14, and IVG15] which are reserved for exclusive use by VDK and must not be managed by the interrupt manager. IVG15 is also excluded from most VisualDSP++ applications as it is used to run the applications in supervisor mode.

- Deferred callbacks – The deferred callback manager offers a similar service to the VDK process running at IVG14. It is highly recommended that the VDK variant of the system services library is used (and indeed the default VDK `.ldf` files ensure its use). This variant essentially passes callbacks posted to the DCB manager to the VDK level 14 process. In this mode of operation, only one callback queue can be used. If the standalone library variant is used, several queues can be managed but none of them can be assigned to the IVG 14 level as this would conflict with the VDK process running at that level.

Deployment of Services Within a Multi-Threaded Application

Bear in mind these two major considerations when deploying system services and the device driver model within a multi-threaded application.

- Critical regions – System services and device drivers use critical regions where atomicity of a code segment is required. These regions are managed through calls to the `adi_int_EnterCriticalRegion` function and the `adi_int_ExitCriticalRegion` function, which are defined in the `adi_int_xxx.c` files within the installation. (For more information, see "Interrupt Manager" on page 2-1.) It is advised that the above functions are used within threads that use system services rather than the VDK push/pop critical region functions.

- Initialization – The initialization of system services and the device manager is performed only once per application. Since their use may be required in several threads, it is important that the initialization is performed prior to any subsequent use. In addition, all device drivers that need to adjust their timing values according to the peripheral clock (SCLK) frequency must employ a call to `adi_pwr_GetFreq()` to determine the frequency (in Hz). The power management module must be initialized prior to the opening of any device driver.

There are basically three approaches that can be adopted:

- Define a function to initialize the system services and device manager and call it from a user-modifiable section of the "start" routine in `<Project>_basiccrt.s`.
- Assign the initialization to the highest-priority boot thread.
- Use a separate boot thread to perform the initialization and set it at the highest priority and let it yield to other threads once completed or be destroyed. Use global and not thread memory to initialize the system services and device manager in this way.

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 provides a simple, convenient method for applications to control devices commonly found in and around Analog Devices processors. It has also provides a simple and efficient mechanism for the creation of new device drivers.

The system services overview covers the following topics:

- "Application Interface" on page 1-23
- "Device Driver Architecture" on page 1-24
- "Initialization" on page 1-26

- "Termination" on page 1-27
- "Device Driver Directory and File Structure" on page 1-27

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 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 1 MB 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 is a model-compliant driver that 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.

Similar to 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 physical driver to provide logic that operates both synchronously and asynchronously. The device manager provides this functionality, relieving each physical driver from reimplementing this capability.

The device manager architecture is illustrated in Figure 1-2.

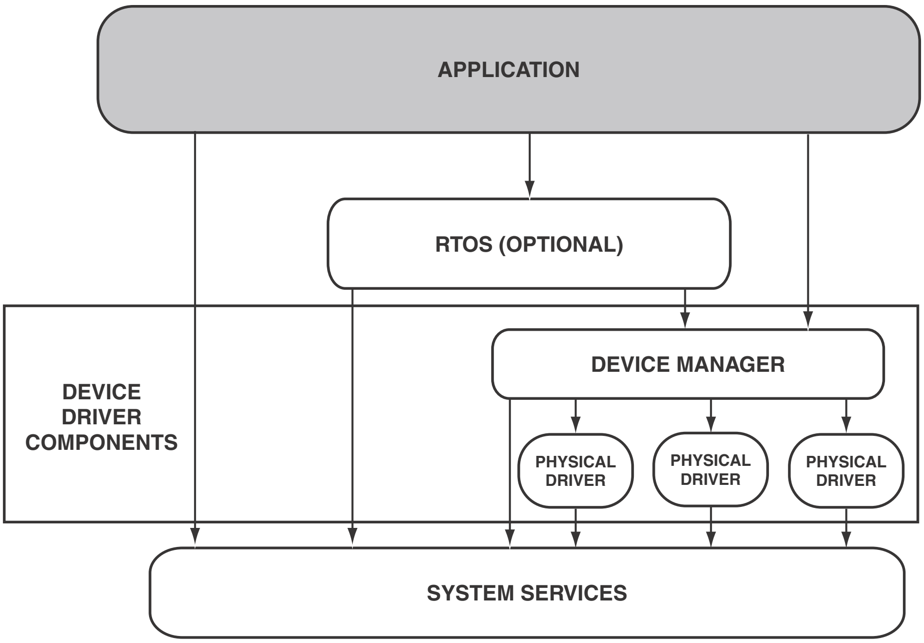

Figure 1-2. Device Manager Architecture

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 only one device manager 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.

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 code size and data memory savings, this approach provides each software component with 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 because each driver does not have to reimplement 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 set up 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 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 1-11 for information on system services initialization.

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. An application that operates in an endless loop 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 reinitialized 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 directory that stores 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 to use include files in their source code, and link with a device driver library and a system services library module.

Accessing the Device Driver API

User source files accessing the device manager API should include the files `services.h` and `adi_dev.h`, located in the `Blackfin/include/services` and `Blackfin/include/drivers` directories, respectively. In addition, your source file should use the include file of the physical driver that will be accessed.

For example, user code that accesses the Analog Devices parallel peripheral interface (PPI) driver would include the following lines in their source file (in order):

```
#include <services/services.h>          // system services
#include <drivers/adi_dev.h>            // device manager
#include <drivers/ppi/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. Regardless of the Blackfin processor being targeted, application software does not change. 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. You must 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, however, should ensure that the processor definition macro is appropriately defined. The `services.h` file enumerates the specific processor variants that are supported. These processor variants are listed in Table 1-1 on page 1-14.

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.

Device Driver File Locations

Device drivers for on-chip peripherals are provided in the `libdrvxxx.dlb` library for the various processor derivatives, silicon revisions, and so on. Device drivers for off-chip peripherals are not provided within the library, but rather must be included separately with the application. Include files for off-chip peripheral drivers are included in following subdirectories:

```
$ADI_DSP\Blackfin\include\drivers
```

where `$ADI_DSP` is the location of your VisualDSP installation, which is, by default, located at:

`C:\Program Files\Analog Devices\VisualDSP` *<version>*

Source files for off-chip peripheral drivers are included in subdirectories:

```
$ADI_DSP\Blackfin\lib\src\drivers
```

When creating applications that include off-chip device drivers, the application should include the `.h` file for the driver. This is typically done with something like this:

```
#include <drivers\codec\adi_ad1836.h>
```

The source code for an off-chip peripheral driver should be included in the source file list of the VisualDSP++ project. For example, if using the AD1836 device driver, the file

```
$ADI_DSP\Blackfin\lib\src\drivers\codec\adi_ad1836.c
```

should be included in the source file list.

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 supported processor variant. You 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 `libdrvxxxz.dlb` where:

- `xxx` represents the processor variant – This is typically a three-digit number that identifies the processor variant, such as `532` for the ADSP-BF532 processor, `534` for the ADSP-BF534 processor, and so on.
- `z` represents any special conditions for the library – The following combinations are used:
  - `y` – The library is built to avoid all known anomalies for all revisions of silicon.
  - blank – A library without an additional suffix 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.

One device driver library file should be included for the linker to process. Choose the correct library based on the processor variant for your system.

For example, an application developer targeting silicon revision 0.2 of the ADSP-BF532 processor should link with the `libdrv532.dlb` file from the `Blackfin/lib/bf532_rev_0.2` subdirectory. As another example, the application developer who wants a 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 you 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.

Specify the use of debug versions of the libraries by selecting **Use Debug System libraries** on the **Link:Processor** page of the **Project Options** dialog box.

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 ability to tailor the implementation to a user's 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 who use 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 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:

1. Set the preprocessor 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. Refer to the compiler's `-si-revision` switch for more information.
4. Compile/assemble all files in the `Blackfin/lib/src/drivers` directory.
5. Link the appropriate compiled/assembled objects including all object files into a library.

Examples on Distribution

The device driver distribution includes examples that illustrate how to use the device drivers. 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 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 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 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 these interrupts wake the processor from an idled operating mode.

In systems that employ Blackfin processors, often there are more potential interrupt sources than IVG levels. As stated above, some events (such as NMI) map one-to-one to an IVG level. Other events, 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, whether they 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. When 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 processes only 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 to manually control interrupts.

Multi-core Blackfin processors extend this capability 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, `typedef` statements 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. This allows applications to quickly and easily determine whether any errors occurred in processing.

Interrupt Manager Initialization

To use the interrupt manager, a function must initialize the interrupt manager. The function that initializes the interrupt manager is called `adi_int_Init`. The application that calls `adi_int_Init` passes an argument defining the memory that the interrupt manager uses when operating.

The amount of memory provided depends on the number of secondary handlers 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 (that is, only the primary interrupt handler and no secondary handlers are present), the application does not need to provide memory to the interrupt manager's initialization function. However, if the application hooks in secondary interrupt handlers, the application must provide additional memory to support the secondary handlers.

The `ADI_INT_SECONDARY_MEMORY` macro is defined to be the amount of memory (in bytes) required to support a single secondary handler. Thus, the application should provide to the initialization function "`n`" times `ADI_INT_SECONDARY_MEMORY`, where "`n`" is the number of secondary handlers that are supported.

Another parameter passed to the initialization function is the parameter that the interrupt manager passes to the `adi_int_EnterCriticalRegion()` function. This value depends upon the operating environment of the application. See the `adi_int_EnterCriticalRegion` function for more information.

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

Interrupt Manager 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_Terminate()`. 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 the that the interrupt manager was controlling. After calling the termination function, any memory provided to the initialization function may be reused 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 reinitialize interrupt manager services by calling the `adi_init_Init` function.

Core Event Controller Functions

Only two functions are necessary to provide complete control over the core event controller (CEC): `adi_int_CECHook()` and `adi_int_CECUnhook()`, as described next.

adi_int_CECHook() Function

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 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 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 such that they interrogate the system quickly when determining whether 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, it should immediately return with the `ADI_INT_RESULT_NOT_PROCESSED` return code. The interrupt manager then 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 should return the ADI_INT_RESULT_PROCESSED return code.

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 (EVT). 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, 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, and 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() Function

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 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, no interrupt handlers are 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 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.

Therefore, each peripheral interrupt handler must 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 a peripheral interrupt wakes up the processor from an idled state.
- `adi_int_SICInterruptAsserted` – Detects whether a peripheral interrupt is asserted.
- `adi_int_SICGlobalWakeup` – Disables all peripherals from waking the processor, or restores all wakeups to previous state.

Except for the global wakeup disable/enable function, all of these SIC functions take as a parameter an enumeration value that uniquely identifies a peripheral interrupt. The `ADI_INT_PERIPHERAL_ID` enumeration identifies each possible peripheral interrupt source for the processor. This enumeration is defined in the `adi_int.h` file. Refer to this header 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 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 power-up, the Blackfin processor invokes a default mapping of peripheral interrupts to the IVG level. This function alters that mapping. In addition to the `ADI_INT_PERIPHERAL_ID` parameter, this function is passed to 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.

adi_int_SICGlobalWakeup

The SIC interrupt wakeup register contains bits which correspond to each peripheral which may wake up the core, when the interrupt trigger and the core are in an idled state, and the bit corresponding to the peripheral is set. By default, all bits in the SIC interrupt wakeup register are set.

The `adi_int_SICGlobalWakeup()` function is used to globally disable all the peripheral interrupts from waking up the core, by setting all bits of the SIC interrupt wakeup register to zero. The function is also used to restore the SIC interrupt wakeup register to a previous state. This function is passed a `TRUE/FALSE` flag, and a pointer to a `ADI_INT_WAKEUP_REGISTER` structure, into which the values of the wakeup register are saved, when globally disabling wakeups, or from which they are restored, when globally re-enabling wakeups.

If the flag is `FALSE`, the function interrogates the fields of the SIC interrupt wakeup register and stores the values into the corresponding fields of the `ADI_INT_WAKEUP_REGISTER` structure, referenced by the function argument pointer. The SIC interrupt wakeup register is then programmed such that no peripheral interrupts can wake up the core.

If the flag passed in is `TRUE`, this function programs the SIC wakeup register according to the fields of the `ADI_INT_WAKEUP_REGISTER` structure, referenced by the function argument pointer. These values are presumed to have been saved by a previous call to `adi_int_SICGlobalWakeup`.

For example, this function may be used to switch to a low power mode, when only selected wakeups are to be left enabled.

```
/* Declare a wakeup register structure for saving the wakeup reg-
ister state */
ADI_INT_WAKEUP_REGISTER RegIWR;

/* Globally disable wakeups, saving the wakeup register state */
adi_init_SICGlobalWakeup(FALSE, &RegIWR);

/* Enable the PLL wakeup, to change power modes */
adi_int_SICWakeup(ADI_INT_PLL_WAKEUP, 1) ;

/* Enable any other wakeups, to wake the processor from sleep */

/* Go to sleep */
adi_pwr_SetPowerMode(ADI_PWR_MODE_SLEEP);

/* Upon wakeup, restore the wakeups to their previous state */
adi_int_SICGlobalWakeup(TRUE, &RegIWR);
```

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 Code 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: `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 must be used in pairs.

The actual implementation of these functions varies from operating environment to operating environment. For example, in a standalone system (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 the same, regardless of implementation. In this way, application code always operates the same way, and does not 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 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 returned from the `adi_int_EnterCriticalRegion()` function must be passed to the corresponding `adi_int_ExitCriticalRegion()` function. Although 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 realizes 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 `Foo()` function while in a critical section of code. The `Foo()` function 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 passed into the `adi_int_EnterCriticalRegion()` function depends upon the actual implementation for the given operating environment. In some operating environments, the value is not used and can be NULL. For more information on the `pArg` parameter, 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.

ⓘ 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 controls the `IMASK` register tightly 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 functions 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: `adi_int_SetIMASKBits` and `adi_int_ClearIMASKBits`. These functions take as a parameter a value that corresponds to the `IMASK` register of the targeted processor. When the `adi_int_SetIMASKBits()` function is called, the function sets to 1 those bits in the `IMASK` register that have a 1 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 will ever need 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 `adi_int.h` header file. 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 only one of the system services library files. These files are located in the `Blackfin/lib` directory. See the appropriate section in Chapter 6, 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 because 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 – 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

The `adi_int_Init()` function sets aside and initializes memory for the interrupt manager. It also initializes other tables and vectors within the interrupt manager. This function is only called once per core. Separate memory areas are assigned for each core.

Prototype

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

Arguments

`pMemory`	Pointer to an area of memory used by the interrupt manager
`MemorySize`	Size, in bytes, of memory 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

`ADI_INT_RESULT_SUCCESS`	Successfully initialized.

adi_int_Terminate

Description

The `adi_int_Terminate()` 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 (IVG) that were enabled and controlled by the interrupt manager are disabled.

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

The function takes no arguments.

Return Value

`ADI_INT_RESULT_SUCCESS`	Process completed successfully.

adi_int_CECHook

Description

The `adi_int_CECHook()` 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 called first, 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.

File Structure

Prototype

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

Arguments

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

Return Value

`ADI_INT_RESULT_SUCCESS`	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`	IVG level is invalid.

adi_int_CECUnhook

Description

The `adi_int_CECUnhook()` 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 the 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 present in the chain, 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,
        void                *ClientArg
);
```

File Structure

Arguments

`IVG`	Interrupt vector group number being addressed
`Handler`	Client's interrupt handler removed from the chain for the given IVG
`ClientArg`	A `void *` value that is passed to the interrupt handler. To remove the interrupt handler successfully, match this value to the `ClientArg` parameter that was passed to the `adi_int_CECHook()` function when the interrupt handler was hooked into the chain.

Return Value

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

adi_int_ClearIMaskBits

Description

The adi_int_ClearIMaskBits() 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 whether the processor is within a protected region of code (refer to the adi_int_EnterCriticalRegion and adi_int_ExitCriticalRegion functions, respectively). 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. Upon entering this function, if 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.c, located in the blackfin/lib/src/services/int directory, where xxx is the operating environment.

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

The `adi_int_EnterCriticalRegion()` function creates a condition that protects a critical region of code. The companion function, `adi_int_ExitCriticalRegion`, removes the condition. These functions are used to bracket a section of code that requires protection from other processing. These functions are used in pairs sparingly and only when critical regions of code needs to be protected.

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

The actual condition that is created depends 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.c`, located in the `blackfin/lib/src/services/int` directory, where `xxx` is the operating environment.

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

The `adi_int_ExitCriticalRegion()` function removes the condition that was established by the `adi_int_EnterCriticalRegion` to protect a critical region of code. These functions are used to bracket a section of code that needs protection from other processing. These functions are used sparingly and only when critical regions of code require protection.

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`	Return value from the corresponding `adi_int_EnterCriticalRegion()` function call

Return Value

None

adi_int_GetCurrentIVGLevel

Description

The `adi_int_GetCurrentIVGLevel()` is a function that senses the IVG level at which the processor is currently running.

Prototype

```
ADI_INT_RESULT adi_int_GetCurrentIVGLevel(
        u32      *pIVG
);
```

Arguments

`pIVG`	Pointer to the memory location in which the current IVG level is returned

Return Value

`ADI_INT_RESULT_SUCCESS`	IVG level was successfully returned.
`ADI_INT_RESULT_NOT_ASSERTED`	No interrupt is currently active.

adi_int_GetLibraryDetails

Description

The `adi_int_GetLibraryDetails()` function accepts a pointer to an `ADI_INT_LIBRARY_DETAILS` data structure. This function also returns the library details in the `ADI_INT_LIBRATY_DETAILS` structure.

Prototype

```
ADI_INT_RESULT adi_int_GetLibraryDetails(
        ADI_INT_LIBRARY_DETAILS      *pLibraryDetails
);
```

Arguments

`pLibraryDetails`	`ADI_INT_LIBRARY_DETAILS` stucture in which the library details are stored.

Return Value

`ADI_INT_RESULT_SUCCESS`	Function completed successfully.

adi_int_SICDisable

Description

The `adi_int_SICDisable()` 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`	`ADI_INT_PERIPHERAL_ID` enumeration value that identifies an interrupt source

Return Value

`ADI_INT_RESULT_SUCCESS`	System interrupt controller has been successfully configured.
`ADI_INT_RESULT_INVALID_PERIPHERALID`	Peripheral ID specified is invalid.

adi_int_SICEnable

Description

The `adi_int_SICEnable()` function configures the system interrupt controller to enable the given interrupt and allow it to pass 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`	`ADI_INT_PERIPHERAL_ED` enumeration value that identifies a peripheral interrupt source

Return Value

`ADI_INT_RESULT_SUCCESS`	System interrupt controller has been successfully configured.
`ADI_INT_RESULT_INVALID_PERIPHERAL_ID`	Peripheral ID specified is invalid.

adi_int_SICGetIVG

Description

The `adi_int_SICGetIVG()` 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_SICGetIVG(
        const ADI_INT_PERIPHERAL_ID   PeripheralID,
        u32                           *pIVG
);
```

Arguments

`PeripheralID`	`ADI_INT_PERIPHERAL_ID` enumeration value that identifies a peripheral interrupt source
`pIVG`	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

`ADI_INT_RESULT_SUCCESS`	Function completed successfully.
`ADI_INT_RESULT_INVALID_PERIPHERAL_ID`	Peripheral ID specified is invalid.
`ADI_INT_RESULT_INVALID_IVG`	Interrupt vector group level is invalid.

adi_int_SICInterruptAsserted

Description

The `adi_int_SICInterruptAsserted()` function determines whether a given peripheral interrupt source is asserting an interrupt. This function is typically called in an application's interrupt handler to determine whether 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`	`ADI_INT_PERIPHERAL_ID` enumeration value that identifies a peripheral interrupt source

Return Value

`ADI_INT_RESULT_INVALID_PERIPHERAL_ID`	Peripheral ID specified is invalid.
`ADI_INT_RESULT_ASSERTED`	Specified peripheral is asserting an interrupt.
`ADI_INT_RESULT_NOT_ASSERTED`	Specified peripheral is not asserting an interrupt.

adi_int_SICSetIVG

Description

The `adi_int_SICSetIVG()` 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`	`ADI_INT_PERIPHERAL_ID` enumeration value that identifies a peripheral interrupt source
`IVG`	Interrupt vector group assigned to the peripheral

Return Value

`ADI_INT_RESULT_SUCCESS`	Function completed successfully.
`ADI_INT_RESULT_INVALID_PERIPHERAL_ID`	Peripheral ID specified is invalid.
`ADI_INT_RESULT_INVALID_IVG`	Interrupt vector group level is invalid.

adi_int_SetIMaskBits

Description

The adi_int_SetIMaskBits() 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 depends upon the operating environment. In the standalone version of the service, this function detects whether the processor is within a protected region of code (refer to 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. Upon entering this function, if 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.c, located in the blackfin/lib/src/services/int directory, where xxx is the operating environment.

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

The `adi_intSICWakeup()` 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`	`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

`ADI_INT_RESULT_SUCCESS`	System interrupt controller has been successfully configured.
`ADI_INT_RESULT_INVALID_PERIPHERAL_ID`	Peripheral ID specified is invalid.

adi_int_SICGlobalWakeup

Description

The `adi_int_SICGlobalWakeup()` function is used to program the system interrupt controller (SIC) wakeup register to either disable all the peripheral interrupts from waking up the core, or to restore the SIC interrupt wakeup register to a previous state.

If the flag is `FALSE`, the function saves the contents of the system interrupt controller wakeup register in the `ADI_INT_WAKEUP_REGISTER` structure, referenced by the function argument pointer. It then sets all bits of the SIC wakeup register to zero, disabling all the peripherals from waking up the core.

If the flag passed in is `TRUE`, the function programs the SIC wakeup register according to the fields of the `ADI_INT_WAKEUP_REGISTER` structure, referenced by the function argument pointer. These values are presumed to have been saved by a previous call to `adi_int_SICGlobalWakeup`.

The actual servicing of interrupts is not affected by this function.

Prototype

```
ADI_INT_RESULT adi_int_SICGlobalWakeup(
        u32                         WakeupFlag,
        pADI_INT_WAKEUP_REGISTER   SaveIWR
);
```

Arguments

`WakeupFlag`	Wakeup enable flag to enable or disable waking of the core(s) upon triggering of peripheral interrupts (`TRUE/FALSE`)
`SaveIWR`	Pointer to a structure used for saving the wakeup register state. (Defined according to the processor type, in `adi_int.h`)

Return Value

`ADI_INT_RESULT_SUCCESS`	System interrupt controller wakeup register has been programmed according to the function arguments.

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 (full-on, active, sleep, deep sleep, and hibernate) to another
- 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, 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. For more information about the EBIU module, see "External Bus Interface Unit Module" on page 4-1.

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 clock and system clock frequencies, the valid values nearest to the chosen ones are used and the voltage level of the processor is adjusted accordingly.

In both cases, validity checks are performed at all stages, ensuring that the processor is not stalled or harmed.

"PM Module Operation – Getting Started" on page 3-3 describes the basic operating stages required to use the power management module.

The power management module uses an unambiguous naming convention to safeguard against conflicts with other software libraries provided by Analog Devices or other companies. To this end, all enumeration values and `typedef` statements use the `ADI_PWR_` prefix, and functions and global variables use the lowercase, `adi_pwr_` equivalent.

Two versions of the library exist for each processor. These correspond to the debug and release configurations in the current VisualDSP++ release. 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_NOT_INITIALIZED` when the PM module has not been initialized prior to the function call.

ⓘ In order to better facilitate the configuration of timing parameters for device drivers, the default unit of frequency for communicating with the power management functions is hertz (Hz) rather than megahertz (MHz). Refer to the `adi_pwr_Init` function (on page 3-22) for more information.

PM Module Operation – Getting Started

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

Step 1:
If used in conjunction with the EBIU controller to adjust SDRAM settings, initialize the EBIU module by calling `adi_ebiu_Init()`. For more information about the EBIU module, see "External Bus Interface Unit Module" on page 4-1.

Step 2:
Initialize the power module by calling `adi_pwr_Init`, passing the parameters for the hardware configuration used. For example, the following code configures the ADSP-BF533 EZ-KIT Lite.

```
ADI_PWR_COMMAND_PAIR power_init_table[] = {
    {
ADI_PWR_CMD_SET_PROC_VARIANT,(void*)ADI_PWR_PROC_BF533SKBC600 },
                    /* 600 MHz ADSP-BF533 variant */
    { ADI_PWR_CMD_SET_PACKAGE,(void*)ADI_PWR_PACKAGE_MBGA },
                    /* in MBGA packaging */
    { ADI_PWR_CMD_SET_VDDEXT, (void*)ADI_PWR_VDDEXT_330 },
                    /* 3.3 V External supplied
                    to voltage regulator */
    { ADI_PWR_CMD_SET_CLKIN,  (void*) 25 },
                    /* 25 MHz clock in */
    { ADI_PWR_CMD_END,        0 }
                    /* no more commands after this */
};
adi_pwr_Init(power_init_table);
```

Step 3:
Decide on the 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 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 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);
```

Dual-Core Considerations

The following sections explain how to use system services with a dual-core configuration.

Using Automatic Synchronization

The PLL programming sequence for a dual-core processor requires that both cores be brought to the IDLE state while changes are applied to the `PLL` and `VR` registers. A dual-core processor may execute a program on each core, or it may execute a program on just one core. When both cores are used to execute a program, a mechanism is required for both cores to go to the IDLE state, and stay there while the registers are written. The power management module provides a mechanism that uses the supplemental interrupt to synchronize the cores for PLL programming. This mechanism is invoked automatically by calling the `adi_pwr_Init()` function on both cores and passing the command `ADI_PWR_CMD_SET_AUTO_SYNC_ENABLED` with a NULL argument, as shown in the following command-pair table for the ADSP-BF561 EZ-KIT Lite.

```
ADI_PWR_COMMAND_PAIR power_init_table[] = {
    {
ADI_PWR_CMD_SET_PROC_VARIANT,(void*)ADI_PWR_PROC_BF561SKBCZ500X
},
                    /* 500 MHz ADSP-BF561 variant */
    { ADI_PWR_CMD_SET_PACKAGE,(void*)ADI_PWR_PACKAGE_MBGA },
                    /* in MBGA packaging */
    { ADI_PWR_CMD_SET_VDDEXT, (void*)ADI_PWR_VDDEXT_330 },
                    /* 3.3 V External supplied
                    to voltage regulator */
    { ADI_PWR_CMD_SET_CLKIN,  (void*) 30 },
                    /* 30 MHz clock in */
    { ADI_PWR_CMD_SET_AUTO_SYNC_ENABLED,  NULL },
                    /* enable auto-synchronization */
    { ADI_PWR_CMD_END,        0 }
                    /* no more commands after this */
};
adi_pwr_Init(power_init_table);
```

Synchronization Requirement

Blackfin dual-core processors are capable of running one core while the other core is idle. Power management and EBIU management require that both cores be placed in the IDLE state when making power management and EBIU controller changes. If the EBIU module has been initialized, and the system clock frequency is changed, the SDRAM timing parameters are automatically adjusted. To avoid corruption of SDRAM, the automatic core synchronization mechanism forces both cores to execute outside of the SDRAM memory space, while the SDRAM timing parameters are updated.

There are two possible operating modes: running on one core, and running applications on both cores.

Running Applications on One Core Only

In this case, one core is used and the other core (core B) is disabled. Upon reset, core B remains disabled until the code running on core A starts core B running, by clearing bit 5 of the `SICA_SYSCR` register. For example:

```
*pSICA_SYSCR &= 0xFFDF;      // clears bit 5 so Core B
                             // will start running
```

Note that this does not wake core B if it is in the IDLE state. It only allows core B to start executing instructions on startup. To wake core B from IDLE, use one of the two supplemental interrupts (supplemental interrupt 0 is taken over by system services, leaving supplemental interrupt 1 for other uses).

Single-core applications loaded from flash memory or via the SPI port satisfy the above synchronization requirement with no further intervention. However, an emulator session within VisualDSP++ unavoidably wakes up core B. The application developer must return core B to the disabled state to meet the PLL programming requirements. There are two ways to do this. The simplest is to run the following C code on core B:

```
void main() {
    while(1) {
        asm(“IDLE;”);
    }
}
```

Whenever core B wakes up (due to the PLL programming sequence executed by the power management service) it is immediately returned to the IDLE state.

The other method is to disable the PLL wakeup bit in the `SICB_IWR0` register and go to IDLE. If this is done in the assembler, the following code can take the place of the startup code:

```
#include <defBF561.h>
.section program;

start:
      P0.H = HI(SICB_IWR0); P0.L = LO(SICB_IWR0);
      R0 = 0;
      [P0] = R0;
      IDLE;

.start.end:
.global start;
.type start,STT_FUNC;
```

Running Applications on Both Cores

In this case, both cores execute code. Both cores need to synchronize to ensure that both cores are IDLE and, in some cases, do not execute out of SDRAM as described in the requirement above. There are two choices: 1) define your own synchronization strategy, or 2) use the built-in synchronization provided by the power management module (which must be enabled by a separate command).

To use the built-in synchronization, include the following command-value pair to `adi_pwr_Init()` on both cores:

```
{ ADI_PWR_CMD_SET_AUTO_SYNC_ENABLED, NULL }
```

Once activated, the built-in synchronization has exclusive control over supplemental interrupt 0 and chains an appropriate interrupt handler to the appropriate IVG level using the interrupt manager. This prevents the application from using the interrupt for any other synchronization between cores. However, the supplemental interrupt 1 is still available for use outside of system services for other core synchronization purposes.

Additional commands can be used to tailor the synchronization requirements (see Table 3-1.)

Table 3-1. Additional Commands for Tailoring Synchronization

Command	Description
Available on Core A and Core B	
`ADI_PWR_CMD_SET_SYNC_LOCK_VARIABLE`	Provides the address of an alternative unsigned int lock variable in L2 as an alternative to the built-in lock variable. Not normally needed.
Available on Core B only	
`ADI_PWR_CMD_SET_COREB_SUPP_INTO_IVG`	Specifies the IVG level assigned to supplemental interrupt 0 on core B. Not normally needed.

Synchronization Between Cores

Either core can interrupt the other core using a supplemental interrupt. There are two of these on the ADSP-BF561 processor: 0 and 1. A shared lock variable located in L2 memory can send information between the cores as a method of synchronization.

The built-in mechanism requires that core A initiates all power management changes, with core B configured to respond to a supplemental interrupt 0 event, raised by core A. The configuration and handling of this interrupt is managed within the power management module itself. Table 3-2 describes the synchronization sequence.

Table 3-2. Synchronization Sequence Between Cores

Core A	Core B
Raises supplemental interrupt 0, sets the shared `adi_pwr_lockvar` lock variable, and waits acknowledgement.	Responds to supplemental interrupt 0 by entering interrupt handler.
	Runs first (optional) callback function.
On receiving acknowledgement, performs PLL programming sequence, and configures the SDC accordingly.	Acknowledges interrupt and goes to `IDLE`.
	Wakes on PLL wakeup and waits for the lock variable to clear.
Completes the process by clearing the lock variable.	Runs second (optional) callback function, and returns from interrupt.

Built-In Lock Variable and Linking Considerations

The lock variable, `adi_pwr_lockvar`, is declared within the file `Blackfin\lib\src\ services\pwr\adi_pwr_lockvar.c` as:

```
section ("l2_shared") testset_t adi_pwr_lockvar = 0;
```

where the memory input section, `l2_shared`, is mapped to the `MEM_L2_SRAM` output section in both the default and generated linker description files (`.ldf`).

According to Appendix A of the *VisualDSP++ 5.0 C/C++ Compiler and Library Manual for Blackfin Processors*, there are two possible approaches for building applications that run across both cores.

- One application per core, where executables are built for each core using two passes of the linker
- One application across both cores, where a five-project group is used and a single linker pass builds executables for both processors

The latter approach maps the lock variable to L2 memory shared by both processors without any user intervention. The “one application per core” approach requires user intervention to ensure that the lock variable is mapped to the same address in L2 memory in each of the executables. This is achieved by using the `RESOLVE` statement in the `.ldf` file, which can be used to resolve a symbol to its memory location assigned in the executable for the other core.

The default and generated `.ldf` files for core B contain the following:

```
   /* $VDSG<customize-shared-symbols>                      */
   /* This code is preserved if the LDF is re-generated.   */

////////////////////////////////////////////////////////////
   // ldf_shared_symbols

   /* Issue resolve statement for shared symbols mapped in CoreA.
   ** Below is an example of how to do that.
   */
#if defined(OTHERCORE)  /* OTHERCORE is a macro defined to name
                           of the CoreA DXE */
#  include <shared_symbols.h> /* C runtime library
                                 shared symbols,
                                 ** uses macro OTHERCORE.
                               */
#if 0
                     /* example resolve for user shared data*/
   RESOLVE(_a_shared_datum, OTHERCORE)
#endif

#endif /* OTHERCORE */

   /* $VDSG<customize-shared-symbols>                          */
```

The `shared_symbols.h` header file contains the `RESOLVE` statements for the C/C++ libraries' shared symbols and includes the additional header file, `services/services_shared_symbols.h`, containing the `RESOLVE` statements for the system services shared symbols. Currently, the adi_pwr_lockvar variable is the only shared symbol required by system services.

All that is required is to define OTHERCORE within the user-modifiable block ahead of where it is tested (or by setting its value through the **Link:LDF Preprocessing** page of the **Project Options** dialog box within VisualDSP++). For example, if the executable, CoreA.dxe (say), for core A is in the Release subdirectory of a directory, CoreA, adjacent to the CoreB project directory, you need to define OTHERCORE as ..\CoreA\Release\CoreA.dxe. For example:

```
#define OTHERCORE ..\CoreA\Release\CoreA.dxe
#if defined(OTHERCORE)
:
```

Refer to Appendix A of the *VisualDSP++ 5.0 C/C++ Compiler and Library Manual for Blackfin Processors* and the *VisualDSP++ 5.0 Linker and Utilities Manual* for details.

SDRAM Initialization Prior to Loading an Executable

Applications that require code (and/or data) to be located in SDRAM at load time require the SDRAM controller to be initialized beforehand. This is the case for all applications where instruction and/or data caching are enabled. However, the EBIU service's initialization routine, adi_ebiu_Init(), is not executed until after the application has loaded.

There are two ways to load an application into the processor core:

- Through an emulator session connected to the VisualDSP++ IDDE
- From flash memory or a device attached to the SPI port when the processor is reset

In the first case, it is imperative that the **Use XML reset values** option is selected in the **Target Options** dialog box (available through the **Settings** menu). This ensures that the SDRAM is correctly (if not optimally) configured prior to the application loading. Once loaded, the application's use of the power management services and EBIU services ensures that the SDRAM is optimally configured.

When the application is not loaded from within an emulator session, it is necessary for the boot loader to initialize SDRAM prior to loading the application. This is achieved by using an initialization block as described in the *VisualDSP++ 5.0 Loader and Utilities* manual, where the example given demonstrates the initialization of SDRAM.

This initialization block code is compiled into an executable and is passed to the loader via the `-init` filename option or in the **Initialization file** field of the **Load:Options** page of the **Project Options** dialog box. A separate project is thus required for the initialization block. An example initialization block project is provided in the relevant directory (for the processor) under the `Blackfin\ldr\init_code` directory of the VisualDSP++ installation. The values required for the SDRAM configuration registers can be set to the ones used in the relevant `.xml` file for the processor, for example `ADSP-BF533-proc.xml`, located in the `System\ArchDef` directory of the VisualDSP++ installation.

When a different memory configuration is required (other than the one supplied with the EZ-KIT Lite evaluation systems), the user is required to work out the appropriate values. When loading a program from the IDDE, either use the Custom Board Support feature, now available with VisualDSP++5.0 (described in `Help\Graphical Environment\Custom Board Support`), or simply change the values at the bottom of the relevant `.xml` file after backing up the original file.

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 – 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

The `adi_pwr_AdjustFreq()` 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(
           ADI_PWR_CSEL csel,
           ADI_PWR_SSEL ssel
);
```

Arguments

`csel`	`ADI_PWR_CSEL` value specifies how the voltage core oscillator (VCO) frequency is divided to obtain a new core clock frequency. The divider value cannot exceed the `ssel` value. See "ADI_PWR_CSEL" on page 3-48.
`ssel`	`ADI_PWR_SSEL` value specifies how the VCO frequency is divided to obtain a new system clock frequency. See "ADI_PWR_SSEL" on page 3-55.

Return Value

`ADI_PWR_RESULT_SUCCESS`	Process completed successfully.
`ADI_PWR_RESULT_NOT_INITIALIZED`	PM module has not been initialized.
`ADI_PWR_RESULT_INVALID_CSEL`	Invalid value for `CSEL` has been specified.
`ADI_PWR_RESULT_INVALID_SSEL`	Invalid value for `SSEL` has been specified.
`ADI_PWR_INVALID_CSEL_SSEL_COMBINATION`	Core clock divider is greater than the system clock divider value, or both `ADI_PWR_CSEL_NONE` and `ADI_PWR_SSEL_NONE` are specified.

adi_pwr_Control

Description

The `adi_pwr_Control()` function enables the dynamic power management registers to be configured or queried according to command-value pairs ("ADI_PWR_COMMAND_PAIR" on page 3-48), 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 3-42 and "Public Data Types and Enumerations" on page 3-42 for the complete list of commands and associated values.

Prototype

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

Arguments

`Command`	`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 3-42.

Return Value

`ADI_PWR_RESULT_SUCCESS`	Function completed successfully.
`ADI_PWR_RESULT_BAD_COMMAND`	Invalid command has been specified.
`ADI_PWR_RESULT_NOT_INITIALIZED`	PM module has not been initialized.
`ADI_PWR_RESULT_INVALID_INPUT_DELAY`	Input delay value is invalid.
`ADI_PWR_RESULT_INVALID_OUTPUT_DELAY`	Output delay value is invalid.
`ADI_PWR_RESULT_INVALID_LOCKCNT`	PLL lock count value is invalid.

adi_pwr_GetConfigSize

Description

The `adi_pwr_GetConfigSize()` 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 values of `adi_pwr_GetConfigSize` and the macro, `ADI_PWR_SIZEOF_CONFIG`, incorporate the size of the EBIU module configuration, regardless whether the latter is initialized.

Prototype

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

Return Value

The size of the configuration structure.

adi_pwr_GetFreq

Description

The `adi_pwr_GetFreq()` 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`	Address of location to store the current `CCLK` value (Hz)
`fsclk`	Address of location to store the current `SCLK` value (Hz)
`fvco`	Address of location to store the VCO frequency (Hz)

Return Value

`ADI_PWR_RESULT_SUCCESS`	Process completed successfully.
`ADI_PWR_RESULT_NOT_INITIALIZED`	PM module has not been initialized.

adi_pwr_GetPowerMode

Description

The `adi_pwr_GetPowerMode()` 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

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

Prototype

```
u32 adi_pwr_GetPowerSaving(void);
```

Return Value

The percentage power saving value.

adi_pwr_Init

Description

The `adi_pwr_Init()` function initializes the power management module. The following values must be set for successful initialization:

Processor variant	`ADI_PWR_PROC_KIND` value describes the processor variant. See "ADI_PWR_PROC_KIND" on page 3-51.
Package kind	`ADI_PWR_PACKAGE_KIND` value describes the packaging type of the processor. See"ADI_PWR_PACKAGE_KIND" on page 3-50.
Core voltage (V_{DDINT})	`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 1.4 V required for the processor to run at 750 MHz.
External voltage (V_{DDEXT})	`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. See "ADI_PWR_VDDEXT" on page 3-56.
`CLKIN`	Frequency of the external clock oscillator supplied to the processor in either MHz or Hz.

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,     25 /* MHz */ },
    { ADI_PWR_CMD_END,           0 }
};
```

Table 3-3 lists valid command-value pairs.

Table 3-3. adi_pwr_Init Command-Value Pairs

Command	Description
`ADI_PWR_CMD_SET_CCLK_TABLE`	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 are used instead of the data sheet values.
`ADI_PWR_CMD_SET_PROC_VARIANT`	`ADI_PWR_PROC_KIND` value specifies the processor variant (mandatory). See "ADI_PWR_PROC_KIND" on page 3-51.
`ADI_PWR_CMD_SET_PACKAGE`	`ADI_PWR_PACKAGE_KIND` value describes the packaging type of the processor (mandatory). See "ADI_PWR_PROC_KIND" on page 3-51.
`ADI_PWR_CMD_SET_CLKIN`	`u16` value specifies the external clock frequency, `CLKIN`, supplied to the processor (mandatory).
`ADI_PWR_CMD_SET_VDDINT`	`ADI_PWR_VLEV` value specifies the core voltage level. This should only be passed to `adi_pwr_Init` if an external voltage regulator is used, as its presence instructs the module to bypass the internal regulator. See "ADI_PWR_VLEV" on page 3-56.
`ADI_PWR_CMD_SET_VDDEXT`	`ADI_PWR_VDDEXT` value specifies the external voltage level applied to the internal voltage regulator (mandatory). See "ADI_PWR_VDDEXT" on page 3-56.

Table 3-3. adi_pwr_Init Command-Value Pairs (Cont'd)

Command	Description
ADI_PWR_CMD_SET_IVG	interrupt_kind value (see exception.h) specifies the IVG level for the PLL_WAKEUP event.
ADI_PWR_CMD_SET_INPUT_DELAY	ADI_PWR_INPUT_DELAY value specifies whether to add approximately 200 ps of delay to the time when inputs are latched on the external memory interface. See "ADI_PWR_INPUT_DELAY" on page 3-49.
ADI_PWR_CMD_SET_OUTPUT_DELAY	ADI_PWR_OUTPUT_DELAY value specifies whether to add approximately 200 ps of delay to external memory output signals. See "ADI_PWR_OUTPUT_DELAY" on page 3-49.

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

Table 3-4 lists valid command-value pairs for an ADSP-BF561 dual-core processor.

Table 3-4. ADSP-BF561 Dual-Core Processor Command-Value Pairs

Command	Description
Commands relevant to ADSP-BF561 dual-core processor only.	
ADI_PWR_CMD_SET_AUTO_SYNC_ENABLED	Instructs the power management module to use its built-in mechanism for synchronizing the cores across changes to the PLL. Use NULL as the associated value. This command is to be passed to adi_pwr_Init() on both cores.
ADI_PWR_CMD_SET_COREB_SUPP_INTO_IVG	IVG level that is assigned to supplemental interrupt 0 on core B. This command is passed to adi_pwr_Init() on core B only.

Table 3-4. ADSP-BF561 Dual-Core Processor Command-Value Pairs (Cont'd)

Command	Description
`ADI_PWR_CMD_SET_SYNC_LOCK_VARIABLE`	Address of a lock variable in L2 that is used for built-in synchronization. The default is to use the built-in, `adi_pwr_lockvar`, variable.
`ADI_PWR_CMD_SET_FIRST_CLIENT_CALLBACK`	Address of a function called by core B before PLL changes are made. This command is passed to `adi_pwr_Init()` on core B only.
`ADI_PWR_CMD_SET_SECOND_CLIENT_CALLBACK`	Address of a function called by core B after PLL changes are made. This command is passed to `adi_pwr_Init()` on core B only.
`ADI_PWR_CMD_SET_CLIENT_HANDLE`	`void*` value/address that is sent to the callback functions. This command is passed to `adi_pwr_Init()` on core B only.

Prototype

```
ADI_PWR_RESULT adi_pwr_Init(
        const ADI_PWR_COMMAND_PAIR *table
);
```

Arguments

`ConfigData`	Address of a table of command-value pairs as defined by "ADI_PWR_COMMAND_PAIR" on page 3-48 and "Public Data Types and Enumerations" on page 3-42. 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 results codes listed below. Otherwise, the value of `ADI_PWR_RESULT_SUCCESS` is returned, or the value of `ADI_PWR_RESULT_ALREADY_INITIALIZED` is returned when the PM module is already initialized.

In order to better facilitate the configuration of timing parameters for device drivers, the default unit of frequency for communicating with the power management functions is hertz (Hz) rather than megahertz (MHz).

Should the application require MHz rather than Hz, the power management service can be commanded to use MHz by passing the new command `ADI_PWR_CMD_SET_FREQ_AS_MHZ` to the `adi_pwr_Init()` function. The companion value parameter is ignored with this command. For example, if passing a table of commands to the `adi_pwr_Init()` function, the following command should be added to the table:

```
{ ADI_PWR_CMD_SET_FREQ_AS_MHZ, NULL },
```

Table 3-5 lists and explains the return codes.

Table 3-5. adi_pwr_Init Return Codes

Return Value	Explanation
`ADI_PWR_RESULT_SUCCESS`	Function completed successfully.
`ADI_PWR_RESULT_BAD_COMMAND`	Invalid command has been specified.
`ADI_PWR_RESULT_ALREADY_INITIALIZED`	Module has already been initialized.
`ADI_PWR_RESULT_INVALID_VLEV`	Invalid core voltage level has been specified.
`ADI_PWR_RESULT_INVALID_VDDEXT`	Invalid external voltage level has been specified.
`ADI_PWR_RESULT_VDDINT_MUST_BE_SUPPLIED`	When using external voltage regulation, the externally-supplied `VDDINT` must be passed to `adi_pwr_Init`.
`ADI_PWR_RESULT_INVALID_PROCESSOR`	Processor type specified is invalid.
`ADI_PWR_RESULT_INVALID_IVG`	IVG level supplied is invalid.
`ADI_PWR_RESULT_INVALID_INPUT_DELAY`	Input delay value is invalid.
`ADI_PWR_RESULT_INVALID_OUTPUT_DELAY`	Output delay value is invalid.
`ADI_PWR_RESULT_INVALID_LOCKCNT`	PLL lock count value is invalid.
`ADI_PWR_RESULT_INVALID_EZKIT`	Invalid EZ-KIT Lite type specified.
`ADI_PWR_RESULT_CANT_HOOK_SUPPLEMENTAL_INTERRUPT`	Unable to hook supplemental interrupt, for halting other core (dual-core only)

adi_pwr_LoadConfig

Description

The `adi_pwr_LoadConfig()` 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 is programmed.

Prototype

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

Arguments

`hConfig`	Address of the memory area where the current configuration is restored
`szConfig`	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

`ADI_PWR_RESULT_SUCCESS`	Function completed successfully.
`ADI_PWR_RESULT_NO_MEMORY`	Value of `szConfig` is insufficient.
`ADI_PWR_RESULT_FAILED`	Address of `hConfig` is zero.
`ADI_PWR_RESULT_NOT_INITIALIZED`	PM module has not been initialized.

adi_pwr_Reset

Description

The `adi_pwr_Reset()` 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

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

Arguments

`hConfig`	Address of the memory location where the current configuration is restored
`szConfig`	Number of bytes available at the given address. The value must be greater than or equal to the `adi_pwr_GetConfigSize()` return value.

Return Value

`ADI_PWR_RESULT_SUCCESS`	Function completed successfully.
`ADI_PWR_RESULT_NO_MEMORY`	Value of `szConfig` is insufficient.
`ADI_PWR_RESULT_FAILED`	Address of `hConfig` is zero.
`ADI_PWR_RESULT_NOT_INITIALIZED`	PM module has not been initialized.

adi_pwr_SetFreq

Description

The `adi_pwr_SetFreq()` function sets the PLL controller to provide `CCLK` and `SCLK` values as close as possible to the requested values (in Hz). 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 affect 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, 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`	Requested `CCLK` value in Hz. 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`	Requested `SCLK` value in Hz
`df`	The `ADI_PWR_DF` enumeration (see "ADI_PWR_DF" on page 3-49) is used in this case to indicate whether this function should enable the PLL input divider, to minimize the difference between the requested clock frequency and the actual frequency that can be obtained. Enabling it can also lead to lower power dissipation. Passing `ADI_PWR_DF_ON` indicates that the PLL input divider has already been enabled. Passing `ADI_PWR_DF_NONE` indicates that the function may enable it to achieve better granularity. (`ADI_PWR_DF_OFF` has no meaning in this context.)

Return Value

`ADI_PWR_RESULT_SUCCESS`	Process completed successfully.
`ADI_PWR_RESULT_NOT_INITIALIZED`	PM module has not been initialized.

adi_pwr_SetMaxFreqForVolt

Description

The `adi_pwr_SetMaxFreqForVolt()` 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 affect the changes.

Prototype

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

Arguments

`vlev`	Required voltage level is set as an `ADI_PWR_VLEV` enumeration value. See "ADI_PWR_VLEV" on page 3-56.

Return Value

`ADI_PWR_RESULT_SUCCESS`	Function completed successfully.
`ADI_PWR_RESULT_INVALID_VLEV`	`vlev` value is invalid.
`ADI_PWR_RESULT_VR_BYPASSED`	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 3-37.
`ADI_PWR_RESULT_NOT_INITIALIZED`	PM module has not been initialized.

adi_pwr_SetPowerMode

Description

The `adi_pwr_SetPowerMode()` 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 the `adi_pwr_SetFreq` or `adi_pwr_SetVoltageRegulator` functions. 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 the `adi_pwr_SetFreq` or `adi_pwr_SetVoltageRegulator` functions. DMA is restricted to external memory. See instructions below for selecting wakeup(s) to bring the processor out of sleep mode.
- **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. See instructions below for selecting wakeup(s) to bring the processor out of hibernate mode.

Until SDRAM is properly configured and the refresh rate is appropriate, data held in SDRAM will decay. This only applies to exiting hibernate or deep sleep mode by a hardware reset event. For

ADSP-BF531, ADSP-BF532, and ADSP-BF533 processor cores, the SCKE pin on the processor is asserted on reset, causing the SDRAM to exit self-refresh mode. This behavior is a constraint of PC-133 compliance. For some processors, currently including the ADSP-BF537 family and the ADSP-BF527 family, this restriction can be circumvented by enabling the CKELOW bit in the VR_CTL register (see "adi_pwr_SetVoltageRegulator" on page 3-37). This can also be achieved by inserting the following command-value pair to the table that is passed to the adi_pwr_Init function:

```
{ ADI_PWR_CMD_SET_PC133_COMPLIANCE, 0 }
```

To specify the method of wakeup from sleep or hibernate mode:

1. Call adi_int_SICGlobalWakeup to disable all wakeups.
2. Call adi_int_SICWakeup for each wakeup that is to be left enabled, while the processor is in the low power mode.
3. Call adi_pwr_SetVoltageRegulator to enable the appropriate wakeup bit(s) in the VR_CTL register (if not already enabled).
4. Call adi_pwr_SetPowerMode to set the power mode.
5. Upon wakeup, restore wakeup registers to their previous state by calling adi_int_SICGlobalWakeup.

Prototype

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

Arguments

`mode`	`ADI_PWR_MODE` value indicates the state the processor is transitioned to. See "ADI_PWR_MODE" on page 3-50.

Return Value

`ADI_PWR_RESULT_SUCCESS`	Process completed successfully.
`ADI_PWR_RESULT_NOT_INITIALIZED`	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

The `adi_pwr_SetVoltageRegulator()` function sets the voltage regulator control register, `VR_CTL`, with one or more of the following fields.

`VLEV`	Required voltage level as an `ADI_PWR_VLEV` enumeration value. See "ADI_PWR_VLEV" on page 3-56.
`FREQ`	Required voltage regulator switching oscillator frequency as an `ADI_PWR_VR_FREQ` enumeration value. See "ADI_PWR_VR_FREQ" on page 3-58. Note: supply `ADI_PWR_VR_FREQ_POWERDOWN` to bypass the on-board voltage regulator.
`GAIN`	Required gain value as an `ADI_PWR_VR_GAIN` enumeration value. See "ADI_PWR_VR_GAIN" on page 3-58.
`WAKE`	`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 3-60.
`PHYWE`	`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 3-59.
`CANWE`	`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 3-57.
`CLKBUFOE`	`ADI_PWR_VR_CLKBUFOE` enumeration value to govern whether other devices, most likely the Ethernet `PHY`, are clocked by the input clock, `CLKIN`. This bit is set if the Ethernet `PHY` is used on the ADSP-BF537 EZ-KIT Lite board. See "ADI_PWR_VR_CLKBUFOE" on page 3-57.
`CKELOW`	`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 is set if the SDRAM is placed into self-refresh mode while the processor is in hibernate state. See "ADI_PWR_VR_CKELOW" on page 3-57.
`USBWE`	`ADI_PWR_VR_USBWE` enumeration value indicating whether the voltage regulator can be awakened from power-down by activity on the USB interface. See "ADI_PWR_VR_USBWE" on page 3-59.

These values are communicated to the `adi_pwr_SetVoltageRegulator` function by passing 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 3-16.

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

```
adi_pwr_SetVoltageRegulator(ADI_PWR_SET_VR_FREQ,
        (void*) ADI_PWR_VR_FREQ_POWERDOWN);
```

Table 3-6 defines the command-value pairs that can be used with the adi_pwr_SetVoltageRegulator function. Use of any other pairs is invalid.

Table 3-6. Command-Value Pairs for adi_pwr_SetVoltageRegulator Function

Command	Associated Data Value
`ADI_PWR_CMD_END`	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`	`ADI_PWR_VLEV` value specifying the voltage level required of the voltage regulator. See "ADI_PWR_VLEV" on page 3-56.
`ADI_PWR_CMD_SET_VR_FREQ`	`ADI_PWR_VR_FREQ` value specifying the required voltage regulator switching oscillator frequency. See "ADI_PWR_VR_FREQ" on page 3-58. Use the `ADI_PWR_VR_FREQ_POWERDOWN` value to bypass the on-board voltage regulator.
`ADI_PWR_CMD_SET_VR_GAIN`	`ADI_PWR_VR_GAIN` value specifying the internal loop gain of the switching regulator loop. See "ADI_PWR_VR_GAIN" on page 3-58.

Table 3-6. Command-Value Pairs for adi_pwr_SetVoltageRegulator Function (Cont'd)

Command	Associated Data Value
ADI_PWR_CMD_SET_VR_WAKE	ADI_PWR_VR_WAKE value indicating whether to enable/disable the WAKE bit. See "ADI_PWR_VR_WAKE" on page 3-60.
ADI_PWR_CMD_SET_VR_PHYWE	ADI_PWR_VR_PHYWE enumeration value indicating whether to enable/disable the PHYWE bit. See "ADI_PWR_VR_PHYWE" on page 3-59.
ADI_PWR_CMD_SET_VR_CANWE	ADI_PWR_VR_CANWE enumeration value indicating whether to enable/disable the CANWE bit. See "ADI_PWR_VR_CANWE" on page 3-57.
ADI_PWR_CMD_SET_VR_CLKBUFOE	ADI_PWR_VR_CLKBUFOE enumeration value indicating to enable/disable the CLKBUFOE bit. See "ADI_PWR_VR_CLKBUFOE" on page 3-57.
ADI_PWR_CMD_SET_VR_CKELOW	ADI_PWR_VR_CKELOW enumeration value indicating whether to enable/disable the CKELOW bit. See "ADI_PWR_VR_CKELOW" on page 3-57.
ADI_PWR_CMD_SET_VR_USBWE	ADI_PWR_VR_USBWE enumeration value indicating whether to enable/disable the USB wakeup bit. See "ADI_PWR_VR_USBWE" on page 3-59.

The processor's CCLK and SCLK frequencies are not adjusted. When necessary, the processor is idled to effect the changes. If the requested voltage level is insufficient to sustain the current frequency values, the function returns an error without amending any settings.

Prototype

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

Arguments

`Command`	`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 3-37.

Return Value

`ADI_PWR_RESULT_SUCCESS`	Function completed successfully.
`ADI_PWR_RESULT_INVALID_VLEV`	`VLEV` argument is invalid or insufficient to sustain the current core and system clock frequencies.
`ADI_PWR_RESULT__INVALID_VR_FREQ`	`FREQ` value is invalid.
`ADI_PWR_RESULT__INVALID_VR_GAIN`	`GAIN` value is invalid.
`ADI_PWR_RESULT__INVALID_VR_WAKE`	`WAKE` value is invalid.
`ADI_PWR_RESULT_INVALID_VR_PHYWE`	`PHYWE` value is invalid.
`ADI_PWR_RESULT_INVALID_VR_CANWE`	`CANWE` value is invalid.
`ADI_PWR_RESULT_INVALID_VR_CLKBUFOE`	`CLKBUFOE` value is invalid.
`ADI_PWR_RESULT_INVALID_VR_CKELOW`	`CKELOW` value is invalid.
`ADI_PWR_RESULT_INVALID_VR_USBWE`	`USB` wakeup value is invalid.
`ADI_PWR_RESULT_BAD_COMMAND`	`Command` argument is unrecognized.
`ADI_PWR_RESULT_NOT_INITIALIZED`	PM module has not been initialized.

adi_pwr_Terminate

Description

The `adi_pwr_Terminate()` function terminates the power management module, resets the initialized flag, and unhooks the supplemental interrupt, if dual-core synchronization was used.

Prototype

```
ADI_PWR_RESULT adi_pwr_Terminate(void);
```

Arguments

The function takes no arguments.

Return Value

`ADI_PWR_RESULT_SUCCESS`	Function completed successfully.

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. Table 3-7 details the available commands, the associated data values, and the valid context for their use.

Table 3-7. ADI_PWR_COMMAND Available Commands

Command	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`	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_INSTALL_CLK_CLIENT_CALLBACK`	A value of type `pADI_PWR_CALLBACK_ENTRY` pointing to an `ADI_PWR_CALLBACK_ENTRY` structure which contains the callback function to install, along with the `ClientHandle` value that will be passed to the callback.

Table 3-7. ADI_PWR_COMMAND Available Commands (Cont'd)

Command	Associated Data Value
`ADI_PWR_CMD_SET_INPUT_DELAY`	`ADI_PWR_INPUT_DELAY` value specifying whether to add approximately 200 ps of delay to the time when inputs are latched on the external memory interface. See "ADI_PWR_INPUT_DELAY" on page 3-49.
`ADI_PWR_CMD_SET_OUTPUT_DELAY`	`ADI_PWR_OUTPUT_DELAY` value specifying whether to add approximately 200 ps of delay to external memory output signals. See "ADI_PWR_OUTPUT_DELAY" on page 3-49.
`ADI_PWR_CMD_SET_PLL_LOCKCNT`	`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_PROC_VARIANT`	`ADI_PWR_PROC_KIND` value specifying the processor variant. See "ADI_PWR_PROC_KIND" on page 3-51.
`ADI_PWR_CMD_SET_PACKAGE`	`ADI_PWR_PACKAGE_KIND` value describing the packaging type of the processor. See "ADI_PWR_PACKAGE_KIND" on page 3-50.
`ADI_PWR_CMD_SET_CLKIN`	`u16` value specifying the external clock frequency, `CLKIN`, supplied to the processor in either MHz or Hz.
`ADI_PWR_CMD_SET_VDDINT`	`ADI_PWR_VLEV` value specifying the core voltage level provided by an external voltage regulator. See "ADI_PWR_VLEV" on page 3-56.

Table 3-7. ADI_PWR_COMMAND Available Commands (Cont'd)

Command	Associated Data Value
ADI_PWR_CMD_SET_VDDEXT	ADI_PWR_VDDEXT value specifying the external voltage level applied to the internal voltage regulator. See "ADI_PWR_VDDEXT" on page 3-56.
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	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 are used instead of the data sheet values.
ADI_PWR_CMD_SET_IVG	u16 value specifying the IVG level for the PLL_WAKEUP event. This defaults to 7.
ADI_PWR_CMD_SET_PC133_COMPLIANCE	ADI_PWR_PC133_COMPLIANCE value specifying whether 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 does not apply to all processors).
Commands valid only when passed to the adi_pwr_SetVoltageRegulator function.	
ADI_PWR_CMD_SET_VR_VLEV	ADI_PWR_VLEV value specifying the voltage level required of the voltage regulator. See "ADI_PWR_VLEV" on page 3-56.
ADI_PWR_CMD_SET_VR_FREQ	ADI_PWR_VR_FREQ value specifying the required voltage regulator switching oscillator frequency. Use the ADI_PWR_FREQ_POWERDOWN value to bypass the on-board voltage regulator. See "ADI_PWR_VR_FREQ" on page 3-58.

Table 3-7. ADI_PWR_COMMAND Available Commands (Cont'd)

Command	Associated Data Value
ADI_PWR_CMD_SET_VR_GAIN	ADI_PWR_VR_GAIN value specifying the internal loop gain of the switching regulator loop. See "ADI_PWR_VR_GAIN" on page 3-58.
ADI_PWR_CMD_SET_VR_WAKE	ADI_PWR_VR_WAKE value specifying if the voltage regulator is awakened from power-down upon an interrupt from the RTC or a low- going edge on the RESET# pin. See "ADI_PWR_VR_WAKE" on page 3-60.
ADI_PWR_CMD_SET_VR_PHYWE	ADI_PWR_VR_PHYWE enumeration value indicating whether to enable/disable the PHYWE bit (processors with PHYWE bit only). See "ADI_PWR_VR_PHYWE" on page 3-59.
ADI_PWR_CMD_SET_VR_CANWE	ADI_PWR_VR_CANWE enumeration value indicating whether to enable or disable the CANWE bit (for processors with CAN interface only). See "ADI_PWR_VR_CANWE" on page 3-57.
ADI_PWR_CMD_SET_VR_CLKBUFOE	ADI_PWR_VR_CLKBUFOE enumeration value indicating whether to enable or disable the CLKBUFOE bit (processors with CLKBUFOE bit only). See "ADI_PWR_VR_CLKBUFOE" on page 3-57.
ADI_PWR_CMD_SET_VR_CKELOW	ADI_PWR_VR_CKELOW enumeration value indicating whether to enable or disable the CKELOW bit (processors with CKELOW bit only). See "ADI_PWR_VR_CKELOW" on page 3-57.
ADI_PWR_CMD_SET_VR_USBWE	ADI_PWR_VR_USBWE enumeration value indicating whether to enable/disable the USB wakeup bit. See "ADI_PWR_VR_USBWE" on page 3-59.

Table 3-7. ADI_PWR_COMMAND Available Commands (Cont'd)

Command	Associated Data Value
`ADI_PWR_CMD_SET_VR_GPWE_MXVRWE`	`ADI_PWR_VR_GPWE_MXVRWE` enumeration value indicating whether to enable or disable the `GPWE` (`MXVRWE`) bit (processors with general-purpose or `MXVR` wakeup bit only). See "ADI_PWR_VR_GPWE_MXVRWE" on page 3-59.
Commands valid only when passed to the adi_pwr_Control function.	
`ADI_PWR_CMD_GET_VDDINT`	`ADI_PWR_VLEV` value containing the maximum core voltage level. See "ADI_PWR_VLEV" on page 3-56.
`ADI_PWR_CMD_GET_VR_VLEV`	`ADI_PWR_VLEV` value containing the current voltage level of the internal voltage regulator. Not applicable when the internal regulator is bypassed. See "ADI_PWR_VLEV" on page 3-56.
`ADI_PWR_CMD_GET_VR_FREQ`	`ADI_PWR_FREQ` value containing the current voltage regulator switching oscillator frequency. See "ADI_PWR_VR_FREQ" on page 3-58.
`ADI_PWR_CMD_GET_VR_GAIN`	`ADI_PWR_GAIN` value containing the internal loop gain of the switching regulator loop. See "ADI_PWR_VR_GAIN" on page 3-58.
`ADI_PWR_CMD_GET_VR_WAKE`	`ADI_PWR_VR_WAKE` value specifying if the voltage can be awakened from power-down upon an interrupt from the RTC or a low-going edge on the `RESET#` pin. See "ADI_PWR_VR_WAKE" on page 3-60.
`ADI_PWR_CMD_GET_VR_PHYWE`	`ADI_PWR_VR_PHYWE` enumeration value indicating if the `PHYWE` bit has been enabled/disabled (processors with `PHYWE` bit only). See "ADI_PWR_VR_PHYWE" on page 3-59.

Table 3-7. ADI_PWR_COMMAND Available Commands (Cont'd)

Command	Associated Data Value
`ADI_PWR_CMD_GET_VR_CANWE`	`ADI_PWR_VR_CANWE` enumeration value indicating if the `CAN` wakeup bit has been enabled/disabled (processors with CAN interface only). See "ADI_PWR_VR_CANWE" on page 3-57.
`ADI_PWR_CMD_GET_VR_USBWE`	`ADI_PWR_VR_USBWE` enumeration value indicating if the `USB` wakeup bit has been enabled/disabled (processors with USB interface only). See "ADI_PWR_VR_USBWE" on page 3-59.
`ADI_PWR_CMD_GET_VR_GPWE_MXVRWE`	`ADI_PWR_VR_GPWE_MXVRWE` enumeration value indicating whether if the `GPWE` (`MXVRWE`) bit has been enabled/disabled (processors with general-purpose or `MXVR` wakeup bit only). See "ADI_PWR_VR_GPWE_MXVRWE" on page 3-59.
`ADI_PWR_CMD_GET_VR_CLKBUFOE`	`ADI_PWR_VR_CLKBUFOE` enumeration value indicating if the `CLKBUFOE` bit has been enabled or disabled (processors with `CLKBU-FOE` bit only). See "ADI_PWR_VR_CLKBUFOE" on page 3-57.
`ADI_PWR_CMD_GET_VR_CKELOW`	`ADI_PWR_VR_CKELOW` enumeration value indicating if the `CKELOW` bit has been enabled or disabled (processors with `CKELOW` bit only). See "ADI_PWR_VR_CKELOW" on page 3-57.
`ADI_PWR_CMD_GET_PLL_LOCKCNT`	`u16` value containing the value in the `PLL_LOCKCNT` register.
`ADI_PWR_CMD_REMOVE_CLK_CLIENT_CALLBACK`	A value of type `ADI_PWR_CALLBACK_FN` specifying the callback function to remove.

ADI_PWR_COMMAND_PAIR

This data type is used to generate a table of control commands. These commands are 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;
```

Refer to "ADI_PWR_COMMAND" on page 3-42 for valid values of the `kind` field.

ADI_PWR_CSEL

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

`ADI_PWR_CSEL_1`	Divides voltage core oscillator frequency by 1.
`ADI_PWR_CSEL_2`	Divides voltage core oscillator frequency by 2.
`ADI_PWR_CSEL_4`	Divides voltage core oscillator frequency by 4.
`ADI_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 *ADSP-BF533 Blackfin Processor Hardware Reference*, this leads to lower power dissipation[1].

`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.

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_DISABLE`	Do not add input delay.
`ADI_PWR_INPUT_DELAY_ENABLE`	Add approximately 200 ps 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_DISABLE`	Do not add output delay.
`ADI_PWR_OUTPUT_DELAY_ENABLE`	Add approximately 200 ps of delay to external memory output signals.

[1] See *ADSP-BF533 Blackfin Processor Hardware Reference*, Revision 3.4, April 2009, page 8-4.

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	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 3-56). This value 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 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. It is passed to the adi_pwr_Init() function, along with the command ADI_PWR_CMD_SET_PROC_VARIANT.

The current list of processor variants is shown in Table 3-8. New processors are introduced frequently, so the most accurate information is found in the ADI_PWR_PROC_KIND enumeration itself, grouped by processor family, inside the Power Management Service API header file, adi_pwr.h. Processor variants which are not found there may be defined as "equivalents" by the macros in the 'equivalent values' section of adi_pwr.h. Refer to the data sheet for the specific part number for a complete description of the clock and power capabilities.

Table 3-8. Processor Variants

Enumeration Name	Corresponding Processor
ADI_PWR_PROC_BF561SKBCZ_6A	The ADSP-BF561SKBCZ-6A 600 MHz processor
ADI_PWR_PROC_BF561SKBCZ500X	The ADSP-BF561SKBCZ500X 500 MHz processor
ADI_PWR_PROC_BF561SKBCZ600X	The ADSP-BF561SKBCZ600X 600 MHz processor
ADI_PWR_PROC_BF561SBB600	The ADSP-BF561SBB600 600 MHz processor
ADI_PWR_PROC_BF533SKBC750	The ADSP-BF533SKBC750 750 MHz processor

Table 3-8. Processor Variants (Cont'd)

Enumeration Name	Corresponding Processor
ADI_PWR_PROC_BF533SKBC600	The ADSP-BF533SKBC600 600 MHz processor
ADI_PWR_PROC_BF533SBBC500	The ADSP-BF533SBBC500 500 MHz processor
ADI_PWR_PROC_BF531_OR_BF532	All package types and speed grades of the ADSP-BF531 and ADSP-BF532 processors
ADI_PWR_PROC_BF533SKBC600_6V	The ADSP-BF533SKBC600-6V 600 MHz processor
ADI_PWR_PROC_BF537SKBC1600	The ADSP-BF537SKBC1600 600 MHz processor
ADI_PWR_PROC_BF537SBBC1500	The ADSP-BF537SBBC1500 500 MHz processor
ADI_PWR_PROC_BF536SBBC1400	The ADSP-BF536SBBC1400 400 MHz processor
ADI_PWR_PROC_BF536SBBC1300	The ADSP-BF536SBBC1300 300 MHz processor
ADI_PWR_PROC_BF537BBCZ_5AV	The ADSP-BF537BBCZ-5AV 500 MHz processor
ADI_PWR_PROC_BF548SKBC1600	The ADSP-BF548SKBC1600 600 MHz processor
ADI_PWR_PROC_BF548SBBC1533	The ADSP-BF548SBBC1533 533 MHz processor
ADI_PWR_PROC_BF548SBBC1400	The ADSP-BF548SBBC1400 400 MHz processor
ADI_PWR_PROC_BF538BBCZ500	The ADSP-BF538BBCZ500 500 MHz processor
ADI_PWR_PROC_BF538BBCZ400	The ADSP-BF538BBCZ400 400 MHz processor
ADI_PWR_PROC_BF539BBCZ500	The ADSP-BF539BBCZ500 500 MHz processor
ADI_PWR_PROC_BF539BBCZ400	The ADSP-BF539BBCZ400 400 MHz processor
ADI_PWR_PROC_BF527SBBC1600	The ADSP-BF527SBBC1600 600 MHz processor
ADI_PWR_PROC_BF527SBBC1533	The ADSP-BF527SBBC1533 533 MHz processor
ADI_PWR_PROC_BF526SBBC1400	The ADSP-BF526SBBC1400 400 MHz processor
ADI_PWR_PROC_BF512SBBC1300	The ADSP-BF512SBBC1300 300 MHz processor
ADI_PWR_PROC_BF512SBBC1400	The ADSP-BF512SBBC1400 400 MHz processor

ADI_PWR_RESULT

The power management module functions return a result code of the enumeration type, ADI_PWR_RESULT. Table 3-9 lists and describes the PM module return values.

Table 3-9. PM Module Return Values

Return Value	Explanation
ADI_PWR_RESULT_SUCCESS	Routine completed successfully.
ADI_PWR_RESULT_FAILED	Generic failure was encountered.
ADI_PWR_RESULT_NO_MEMORY	Insufficient memory for configuration values to be stored.
ADI_PWR_RESULT_BAD_COMMAND	Command is not recognized.
ADI_PWR_RESULT_NOT_INITIALIZED	Function call has been ignored with no action taken, due to the PM module not being initialized.
ADI_PWR_RESULT_ALREADY_INITIALIZED	A call to adi_pwr_Init has been ignored with no action taken, due to the PM module having already been initialized.
ADI_PWR_RESULT_INVALID_VDDEXT	Invalid external voltage level has been specified.
ADI_PWR_RESULT_VDDINT_MUST_BE_SUPPLIED	When using external voltage regulation, the externally-supplied VDDINT must be passed to adi_pwr_Init.
ADI_PWR_RESULT_INVALID_PROCESSOR	Processor type specified is invalid.
ADI_PWR_RESULT_INVALID_IVG	IVG level supplied for PLL wakeup is invalid.
ADI_PWR_RESULT_INVALID_INPUT_DELAY	Input delay value is invalid.
ADI_PWR_RESULT_INVALID_OUTPUT_DELAY	Output delay value is invalid.
ADI_PWR_RESULT_INVALID_LOCKCNT	PLL lock count value is invalid.
ADI_PWR_RESULT_INVALID_MODE	Invalid operating mode has been specified.
ADI_PWR_RESULT_INVALID_CSEL	Invalid value for CSEL has been specified.
ADI_PWR_RESULT_INVALID_SSEL	Invalid value for SSEL has been specified.

Table 3-9. PM Module Return Values (Cont'd)

Return Value	Explanation
ADI_PWR_INVALID_CSEL_SSEL_COMBINATION	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_VLEV	VLEV argument is invalid or insufficient to sustain the current core and system clock frequencies.
ADI_PWR_RESULT_INVALID_VR_FREQ	FREQ value is invalid.
ADI_PWR_RESULT_INVALID_VR_GAIN	GAIN value is invalid.
ADI_PWR_RESULT_INVALID_VR_WAKE	WAKE value is invalid.
ADI_PWR_RESULT_INVALID_VR_PHYWE	PHYWE value is invalid.
ADI_PWR_RESULT_INVALID_VR_CANWE	CAN wakeup value is invalid.
ADI_PWR_RESULT_INVALID_VR_USBWE	USBE wakeup value is invalid.
ADI_PWR_RESULT_INVALID_VR_GPWE_MXVRWE	General-purpose or MXVR wakeup value is invalid.
ADI_PWR_RESULT_INVALID_VR_CLKBUFOE	CLKBUFOE value is invalid.
ADI_PWR_RESULT_INVALID_VR_CKELOW	CKELOW value is invalid.
ADI_PWR_RESULT_CANT_HOOK_SUPPLEMENTAL_INTERRUPT	Unable to hook supplemental interrupt, for halting other core (dual-core only)
ADI_PWR_RESULT_NO_CALLBACK_INSTALLED	Tried to remove a callback that was not installed
ADI_PWR_RESULT_EXCEEDED_MAX_CALLBACKS	Could not install a callback. Maximum number of callbacks have been installed.

ADI_PWR_SSEL

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

`ADI_PWR_SSEL_1`	Divides voltage core oscillator frequency by 1.
`ADI_PWR_SSEL_2`	Divides voltage core oscillator frequency by 2.
`ADI_PWR_SSEL_3`	Divides voltage core oscillator frequency by 3.
`ADI_PWR_SSEL_4`	Divides voltage core oscillator frequency by 4.
`ADI_PWR_SSEL_5`	Divides voltage core oscillator frequency by 5.
`ADI_PWR_SSEL_6`	Divides voltage core oscillator frequency by 6.
`ADI_PWR_SSEL_7`	Divides voltage core oscillator frequency by 7.
`ADI_PWR_SSEL_8`	Divides voltage core oscillator frequency by 8.
`ADI_PWR_SSEL_9`	Divides voltage core oscillator frequency by 9.
`ADI_PWR_SSEL_10`	Divides voltage core oscillator frequency by 10.
`ADI_PWR_SSEL_11`	Divides voltage core oscillator frequency by 11.
`ADI_PWR_SSEL_12`	Divides voltage core oscillator frequency by 12.
`ADI_PWR_SSEL_13`	Divides voltage core oscillator frequency by 13.
`ADI_PWR_SSEL_14`	Divides voltage core oscillator frequency by 14.
`ADI_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 V
`ADI_PWR_VDDEXT_250`	2.5 V

ADI_PWR_VLEV

This data type defines the acceptable voltage levels for the voltage regulator. The values for ADSP-BF533 and ADSP-BF561 processors are:

`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 power-down by activity on the controller area network (CAN) interface.

`ADI_PWR_VR_CANWE_DISABLED`	Disable wakeup by CAN activity.
`ADI_PWR_VR_CANWE_ENABLED`	Enable wakeup 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 valid values for all Blackfin processors are:

`ADI_PWR_VR_FREQ_POWERDOWN`	Power-down/bypass on-board 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_GPWE_MXVRWE

This data type defines the values for the `GPWE` general-purpose wakeup or `MXVR` bit, in the voltage regulator control register for some processors. If enabled (`ADI_PWR_VR_GPWE_MXVRWE_ENABLED`), the voltage regulator can be awakened from hibernate upon an interrupt from a general-purpose wakeup or `MXVR`.

`ADI_PWR_VR_GPWE_MXVRWE_DISABLED`	Disables general-purpose wakeup.
`ADI_PWR_VR_GPWE_MXVRWE_ENABLED`	Enables general-purpose wakeup.

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 power-down by activity on the PHY interface.

`ADI_PWR_VR_PHYWE_DISABLED`	Disable wakeup by PHY activity.
`ADI_PWR_VR_PHYWE_ENABLED`	Enable wakeup by PHY activity.

ADI_PWR_VR_USBWE

This data type defines the valid values for the `USBWE` bit in the voltage regulator control register. If enabled, the voltage regulator can be awakened from power-down by activity on the universal serial bus (USB) interface.

`ADI_PWR_VR_USBWE_DISABLED`	Disable wakeup by USB activity.
`ADI_PWR_VR_USBWE_ENABLED`	Enable wakeup by USB 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 power-down (`ADI_PWR_VR_FREQ_POWERDOWN`) upon an RTC interrupt or a low-going edge on the `RESET` pin.

`ADI_PWR_VR_WAKE_DISABLED`	Disables wakeup by RTC and `RESET`.
`ADI_PWR_VR_WAKE_ENABLED`	Enables wakeup by RTC and `RESET`.

PM Module Macros

Table 3-10 lists and describes PM (power management) module macros.

Not shown here is the list of processor variants which are functionally “equivalent” to those in the `ADI_PWR_PROC_KIND` enumeration list. New processors are introduced frequently, so please refer to the macros in the ‘equivalent values’ section of the Power Management Service API header file, `adi_pwr.h`, for the complete list of processor variant “equivalents”.

Table 3-10. PM Module Macros

Macro	Explanation
`ADI_PWR_VLEV_DEFAULT`	Default/reset voltage level `ADI_PWR_VLEV_130`
`ADI_PWR_VLEV_MIN`	Minimum voltage level `ADI_PWR_VLEV_085`
`ADI_PWR_VLEV_MAX`	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`	Default/reset switching frequency value, `ADI_PWR_FREQ_1MHZ`
`ADI_PWR_VR_FREQ_MIN`	Minimum switching frequency value, `ADI_PWR_FREQ_POWERDOWN`
`ADI_PWR_VR_FREQ_MAX`	Maximum switching frequency value, `ADI_PWR_FREQ_1MHZ`

Table 3-10. PM Module Macros (Cont'd)

Macro	Explanation
ADI_PWR_VR_GAIN_DEFAULT	Default/reset voltage regulator gain value, ADI_PWR_GAIN_20
ADI_PWR_VR_GAIN_MIN	Minimum voltage regulator gain value, ADI_PWR_GAIN_5
ADI_PWR_VR_GAIN_MAX	Default/reset voltage regulator gain value, ADI_PWR_GAIN_20
ADI_PWR_PACKAGE_PBGA	Equivalent package type to ADI_PWR_PACKAGE_MBGA

4 EXTERNAL BUS INTERFACE UNIT MODULE

This chapter describes the external bus interface unit (EBIU) module. The EBIU enables the configuration of the asynchronous memory controller and the SDRAM or DDR interface. It also allows the DDR or SDRAM to be automatically adjusted 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 or DDR 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, see "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 data sheet for the external memory device. Thereafter, the refresh rate for SDRAM or DDR is adjusted automatically each time the power management module changes `SCLK`. The asynchronous memory controller is not automatically adjusted but can be explicitly reconfigured via a call to `adi_ebiu_Control`. The "Using the EBIU Module" section provides a step-by-step description of how to work with the EBIU module. Sample code is also included.

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 `typedef` statements use the `ADI_EBIU_` prefix, and functions and global variables use the lowercase equivalent, `adi_ebiu_`.

Two versions of the library are available for each processor, corresponding to the debug and release configurations in VisualDSP++. In addition to the usual defaults for the debug configuration, the API functions perform checks on the passed arguments 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_NOT_INITIALIZED` when 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 necessary parameters for the external memory interfaces that are used. In this step, a table of command-value pairs is passed to the `adi_ebiu_Init` function. The information required is described in detail in `adi_ebiu_Init` in the Description section (on page 4-15). The amount and the type of information that must be passed depends on the individual board configuration.

In the following example, assume that the ADSP-BF533 EZ-KIT Lite (Rev 2.1) is configured. Specify the command-pair table as follows:

```
/* Asynch global control register field - clkout enable */
ADI_EBIU_ASYNCH_CLKOUT clkout_enable =
ADI_EBIU_ASYNCH_CLKOUT_ENABLE;
/* Asynch global control register field - select which banks to
enable */
ADI_EBIU_ASYNCH_BANK_ENABLE banks_enable =
ADI_EBIU_ASYNCH_BANK0_1_2_3;
/* Asynch bank timing parameters, using same value for all 4
banks - specified in either cycles or timing units, but NOT BOTH,
*/
ADI_EBIU_ASYNCH_BANK_TIMING asynch_bank_trans_time =
{ADI_EBIU_BANK_ALL, { ADI_EBIU_ASYNCH_TT_4_CYCLES, { 0,
ADI_EBIU_TIMING_UNIT_NANOSEC } }
ADI_EBIU_ASYNCH_BANK_TIMING asynch_bank_setup_time =
{ADI_EBIU_BANK_ALL, { ADI_EBIU_ASYNCH_ST_3_CYCLES, { 0,
ADI_EBIU_TIMING_UNIT_NANOSEC } } };
ADI_EBIU_ASYNCH_BANK_TIMING asynch_bank_hold_time =
{ADI_EBIU_BANK_ALL, { ADI_EBIU_ASYNCH_HT_2_CYCLES, { 0,
ADI_EBIU_TIMING_UNIT_NANOSEC } } };
ADI_EBIU_ASYNCH_BANK_TIMING asynch_bank_read_access_time =
{ADI_EBIU_BANK_ALL,  { 0xB, { 0, ADI_EBIU_TIMING_UNIT_NANOSEC } }
};
```

```
ADI_EBIU_ASYNCH_BANK_TIMING asynch_bank_write_access_time =
{ADI_EBIU_BANK_ALL, { 7, { 0, ADI_EBIU_TIMING_UNIT_NANOSEC } } };
ADI_EBIU_ASYNCH_BANK_VALUE asynch_bank_ardy_enable = {
ADI_EBIU_BANK_ALL, { ardy_enable: ADI_EBIU_ASYNCH_ARDY_DISABLE }
};
ADI_EBIU_ASYNCH_BANK_VALUE asynch_bank_ardy_polarity = {
ADI_EBIU_BANK_ALL, { ardy_polarity:
ADI_EBIU_ASYNCH_ARDY_POLARITY_LOW } };
/* SDRAM timing parameters, specified according to data sheet */
ADI_EBIU_TIMING_VALUE  twrmin  = {1,{7500,
ADI_EBIU_TIMING_UNIT_PICOSEC}};  /* set min TWR to 1 SCLK cycle +
7.5ns */
ADI_EBIU_TIMING_VALUE  refresh = {8192,{64,
ADI_EBIU_TIMING_UNIT_MILLISEC}}; /* set refresh period to 8192
cycles in 64ms */
ADI_EBIU_TIME  trasmin = {44, ADI_EBIU_TIMING_UNIT_NANOSEC};   /*
set min TRAS to 44ns */
ADI_EBIU_TIME  trpmin  = {20, ADI_EBIU_TIMING_UNIT_NANOSEC};
/* set min TRP to 20ns */
ADI_EBIU_TIME  trcdmin  = {20, ADI_EBIU_TIMING_UNIT_NANOSEC};
/* set min TRCD to 20ns */
u32  cl_threshold = 100; /* set cl threshold to 100 Mhz */
ADI_EBIU_SDRAM_BANK_VALUE bank_size = { 0, { size:
ADI_EBIU_SDRAM_BANK_64MB }};     /* bank size is 64MB */
ADI_EBIU_SDRAM_BANK_VALUE bank_width = { 0, {
width: ADI_EBIU_SDRAM_BANK_COL_10BIT }};  /* column address width
is 10-Bit */
/* set up the command pair table using the above definitions */
ADI_EBIU_COMMAND_PAIR ebiu_init_table[] = {
 { 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*)cl_threshold },
 { 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_SET_ASYNCH_CLKOUT_ENABLE, (void*)&clkout_enable
},
 { ADI_EBIU_CMD_SET_ASYNCH_BANK_ENABLE, (void*)&banks_enable },
 { ADI_EBIU_CMD_SET_ASYNCH_BANK_TRANSITION_TIME,
(void*)&asynch_bank_trans_time },
 { ADI_EBIU_CMD_SET_ASYNCH_BANK_READ_ACCESS_TIME,
(void*)&asynch_bank_read_access_time },
 { ADI_EBIU_CMD_SET_ASYNCH_BANK_WRITE_ACCESS_TIME,
(void*)&asynch_bank_write_access_time },
 { ADI_EBIU_CMD_SET_ASYNCH_BANK_SETUP_TIME,
(void*)&asynch_bank_setup_time },
 { ADI_EBIU_CMD_SET_ASYNCH_BANK_HOLD_TIME,
(void*)&asynch_bank_hold_time },
 { ADI_EBIU_CMD_SET_ASYNCH_BANK_ARDY_ENABLE,
(void*)&asynch_bank_ardy_enable },
 { ADI_EBIU_CMD_SET_ASYNCH_BANK_ARDY_POLARITY,
(void*)&asynch_bank_ardy_polarity },
 { ADI_EBIU_CMD_END, 0 }
};
```

The second argument in the call to `adi_ebiu_Init` is reserved and should be set to zero. The EBIU module should be initialized prior to initializing the power management module, so that subsequent calls to `adi_pwr_SetFreq` or `adi_pwr_SetMaxFreqForVolt` in the power management module will automatically adjust the SDRAM or DDR.

To illustrate what is required for Blackfin processors that support DDR memory, a command table is shown below. Replace the SDRAM parameters, above, with the DDR parameters shown below, and add the asynchronous memory controller commands shown in the above example.

```
ADI_EBIU_TIMING_VALUE RC      { 8, {60,
ADI_EBIU_TIMING_UNIT_NANOSEC  }};      /* cycles between one
  active command and the next */
ADI_EBIU_TIMING_VALUE RAS =  { 6, {42,
ADI_EBIU_TIMING_UNIT_NANOSEC  }};      /* cycles between active
  command and precharge command */
ADI_EBIU_TIMING_VALUE RP =   { 2, {15,
ADI_EBIU_TIMING_UNIT_NANOSEC  }};      /* cycles between
  precharge command and active command */
ADI_EBIU_TIMING_VALUE RFC =   { 10,{72,
ADI_EBIU_TIMING_UNIT_NANOSEC  }};      /* cycles for SDRAM to
  recover from REFRESH signal */
ADI_EBIU_TIMING_VALUE WTR =   { 2,
{7500,ADI_EBIU_TIMING_UNIT_PICOSEC }}; /* cycles from last write
  data until next read command */
ADI_EBIU_TIMING_VALUE tWR =   { 2, {15,
ADI_EBIU_TIMING_UNIT_NANOSEC  }};      /* write recovery time is
  2 or 3 cycles */
ADI_EBIU_TIMING_VALUE tMRD =  { 2, {15,
ADI_EBIU_TIMING_UNIT_NANOSEC  }};       /* cycles from setting of
  mode */
ADI_EBIU_TIMING_VALUE RCD =   { 2, {15,
ADI_EBIU_TIMING_UNIT_NANOSEC  }};      /* cycles from active
  command to next R/W */
ADI_EBIU_TIMING_VALUE REFI =  { 1037,{7777,
ADI_EBIU_TIMING_UNIT_NANOSEC}};       /* cycles from one REFRESH
  signal to the next */

ADI_EBIU_COMMAND_PAIR ebiu_init_table[] = {
```

```
{ ADI_EBIU_CMD_SET_DDR_REFI,         (void*)&REFI  }, /* command
  to set refresh interval */
{ ADI_EBIU_CMD_SET_DDR_RFC,          (void*)&RFC   }, /* command
  to set auto refresh period */
{ ADI_EBIU_CMD_SET_DDR_RP,           (void*)&RP    }, /* command
  to set precharge to active time */
{ ADI_EBIU_CMD_SET_DDR_RAS,          (void*)&RAS   }  /* command
  to set active to precharge time */
{ ADI_EBIU_CMD_SET_DDR_RC,           (void*)&RC    }, /* command
  to set active to active time */
{ ADI_EBIU_CMD_SET_DDR_WTR,          (void*)&WTR   }, /* command
  to set write to read time */
{ ADI_EBIU_CMD_SET_DDR_DEVICE_SIZE,  (void*)0     },  /* command
  to set size of device */
{ ADI_EBIU_CMD_SET_DDR_CAS,          (void*)2      }, /* command
  to set cycles from assertion of R/W until first valid data */
{ ADI_EBIU_CMD_SET_DDR_DEVICE_WIDTH, (void*)2     }, /* command
  to set width of device */
{ ADI_EBIU_CMD_SET_DDR_EXTERNAL_BANKS,(void*)0    }, /* command
  to set number of external banks */
{ ADI_EBIU_CMD_SET_DDR_DATA_WIDTH,    (void*)2    }, /* command
  to set data width */
{ ADI_EBIU_CMD_SET_DDR_WR,            (void*)&tWR }, /* command
   to set write recovery time */
{ ADI_EBIU_CMD_SET_DDR_MRD,           (void*)&tMRD },/* command
  to set cycles from setting mode reg until next command */
{ ADI_EBIU_CMD_SET_DDR_RCD,           (void*)&RCD  }, /* command
  to set cycles from active command to a read-write assertion */

{ ADI_EBIU_CMD_END, 0                              }  /* indicate
the last command of the table */
};
```

In the sample code above, note that the SDRAM minimum TWR value is defined as a structure called ADI_EBIU_TIMING_VALUE which consists of two main parts: a number of cycles, and a number of timing units, in this case, picoseconds. This representation reflects the definition found in the appropriate SDRAM data sheet where the value is expressed as one cycle of SCLK plus 7.5 ns. For the SDRAM refresh period, this structure expresses the time taken for the given number of refresh cycles. The sample code shows that the refresh period is 64 milliseconds, which takes 8192 cycles.

For hardware that uses a Micron SDRAM module, the command-pair table can be abbreviated to just specify the type of the module and the size of the bank, as shown below, adding asynchronous memory controller commands, as needed:

```
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 ebiu_init_table[] = {
    // 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(ebiu_init_table, 0);
```

Further changes can be made at any time by passing 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 – 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 following values for SDRAM: the TRAS, TRP, TRCD, and TWR values in the EBIU_SDGCTL register and the RDIV value in the EBIU_SDRRC register. The function calculates and sets the following values for DDR: the RAS, RP, RFC, REFI, and RC values in the DDRCTL0 register and the RCD, MRD, and WR values in the DDRCTL1 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

```
ADI_EBIU_RESULT adi_ebiu_AdjustSDRAM(
              u32 fsclk
);
```

Arguments

fsclk	System clock (SCLK) frequency in MHz

Return Value

ADI_EBIU_RESULT_SUCCESS	Process completed successfully.
ADI_EBIU_RESULT_NOT_INITIALIZED	SDRAM has not been successfully initialized, or SDRAM had not been enabled.

adi_ebiu_Control

Description

The `adi_ebiu_Control()` function enables the EBIU SDRAM and EBIU DDR registers to be configured according to command-value pairs using one of the following options. (See "ADI_EBIU_COMMAND_PAIR" on page 4-39.)

- 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 4-32 and "Command Value Enumerations" on page 4-39 for the complete list of commands and associated values for both the SDRAM and DDR interfaces and the asynchronous memory interface.

Prototype

```
ADI_EBIU_RESULT adi_ebiu_Control(
        ADI_EBIU_COMMAND Command,
        void *Value
);
```

Arguments

`Command`	`ADI_EBIU_COMMAND` enumeration value specifying the meaning of the associated value argument
`Value`	Required value. (See Description above.)

Return Value

`ADI_EBIU_RESULT_BAD_COMMAND`	Command is not recognized.
`ADI_EBIU_RESULT_SUCCESS`	Function completed successfully.
`ADI_EBIU_RESULT_NOT_INITIALIZED`	EBIU module is not initialized.
`ADI_EBIU_RESULT_INVALID_SDRAM_SRFS`	Invalid self-refresh value is specified. See "ADI_EBIU_SDRAM_TCSR" on page 4-43.
`ADI_EBIU_RESULT_INVALID_SDRAM_PUPSD`	Invalid power-up start delay bit value is specified. See "ADI_EBIU_SDRAM_EBUFE" on page 4-44.

ADI_EBIU_RESULT_INVALID_SDRAM_PSM	Invalid SDRAM power-up sequence bit value is specified. See “ADI_EBIU_SDRAM_PUPSD” on page 4-44.
ADI_EBIU_RESULT_INVALID_SDRAM_EBUFE	Invalid external buffering bit value is specified. See “ADI_EBIU_SDRAM_SRFS” on page 4-43.
ADI_EBIU_RESULT_INVALID_SDRAM_FBBRW	Invalid fast back-to-back, read-to-write bit value is specified. See “ADI_EBIU_SDRAM_FBBRW” on page 4-45.
ADI_EBIU_RESULT_INVALID_SDRAM_CDDBG	Invalid control disable during bus grant bit value is specified. See “ADI_EBIU_SDRAM_CDDBG” on page 4-46.
ADI_EBIU_RESULT_INVALID_SDRAM_EBE	Invalid SDRAM enable selection. See “ADI_EBIU_SDRAM_ENABLE” on page 4-40.
ADI_EBIU_RESULT_INVALID_ASYNCH_BANK_READ_ACCESS_TIME	Invalid asynchronous memory read access time
ADI_EBIU_RESULT_INVALID_ASYNCH_BANK_WRITE_ACCESS_TIME	Invalid asynchronous memory write access time
ADI_EBIU_RESULT_INVALID_ASYNCH_BANK_SETUP_TIME	Invalid asynchronous memory bank setup time
ADI_EBIU_RESULT_INVALID_ASYNCH_BANK_HOLD_TIME	Invalid asynchronous memory bank hold time

adi_ebiu_GetConfigSize

Description

The `adi_ebiu_GetConfigSize()` 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

The adi_ebiu_Init function initializes the EBIU module. Currently, the module is configured to handle either a DDR or a SDRAM controller, plus an asynchronous memory controller. For the EBIU service which supports SRDRAM, 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. For the EBIU service that supports DDR, the adi_ebiu_Init function sets up the DDR control registers, DDRCTL0, DDRCTL1, and DDRCTL2. For successful initialization of the SDRAM or the DDR controller, certain values must be passed to adi_ebiu_Init, as outlined in Table 4-1 and Table 4-2—one for SDRAM and one for DDR. Table 4-1 shows the values that must be passed to adi_ebiu_Init to initialize SDRAM.

Table 4-1. Values for Initialization of SDRAM

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	u32
Minimum TRAS[2] (ns)	ADI_EBIU_CMD_SET_SDRAM_TRASMIN	ADI_EBIU_TIME
Min. TRP[3] (ns)	ADI_EBIU_EBIU_CMD_SET_SDRAM_TRPMIN	ADI_EBIU_TIME
Min. TRCD[4] (ns)	ADI_EBIU_CMD_SET_SDRAM_TRCDMIN	ADI_EBIU_TIME
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.

Table 4-2 shows the values that must be passed to `adi_ebiu_Init` to initialize DDR.

Table 4-2. Values for Initialization of DDR

Description	Command	Value Type
Width of data	`ADI_EBIU_CMD_SET_DDR_DATA_WIDTH`	`u32`
Number of external banks	`ADI_EBIU_CMD_SET_DDR_EXTERNAL_BANKS`	`u32`
Width of device	`ADI_EBIU_CMD_SET_DDR_DEVICE_WIDTH`	`u32`
Size of device	`ADI_EBIU_CMD_SET_DDR_DEVICE_SIZE`	`u32`
Auto-refresh interval	`ADI_EBIU_CMD_SET_DDR_REFI`	`ADI_EBIU_TIMING_VALUE`
Auto-refresh command period	`ADI_EBIU_CMD_SET_DDR_RFC`	`ADI_EBIU_TIMING_VALUE`
Interval between R/W command and valid data	`ADI_EBIU_CMD_SET_DDR_CAS`	`u32`
Interval between active and R/W command	`ADI_EBIU_CMD_SET_DDR_RCD`	`ADI_EBIU_TIMING_VALUE`
Active to active interval	`ADI_EBIU_CMD_SET_DDR_RC`	`ADI_EBIU_TIMING_VALUE`
Active to precharge time	`ADI_EBIU_CMD_SET_DDR_RAS`	`ADI_EBIU_TIMING_VALUE`
Precharge to active time	`ADI_EBIU_CMD_SET_DDR_RP`	`ADI_EBIU_TIMING_VALUE`

Table 4-2. Values for Initialization of DDR (Cont'd)

Description	Command	Value Type
Interval between setting of mode register and next command	`ADI_EBIU_CMD_SET_DDR_MRD`	`ADI_EBIU_TIMING_VALUE`
Write to read interval	`ADI_EBIU_CMD_SET_DDR_WTR`	`ADI_EBIU_TIMING_VALUE`
Write recovery time	`ADI_EBIU_CMD_SET_DDR_WR`	`ADI_EBIU_TIMING_VALUE`

Upon successful initialization of the module, subsequent calls to `adi_ebiu_AdjustSDRAM` adjust the SDRAM refresh rate in the `EBIU_SDRRC` or `DDRCTL0` register to correspond with the given system clock frequency.

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.

If the system configuration makes use of low power (2.5 V) 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_EMREN`
Partial array self-refresh	`ADI_EBIU_CMD_SET_SDRAM_PASR`	`ADI_EBIU_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, which can also be passed to `adi_ebiu_Control`. See `adi_ebiu_Control` for a description of those additional command-value pairs.

The `adi_ebiu_Init` function should be called only 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 Reserved
);
```

Arguments

`ConfigData`	Address of a table of command-value pairs as defined by "ADI_EBIU_COMMAND" on page 4-32 and "Command Value Enumerations" on page 4-39. The last command in the table must be the `ADI_EBIU_CMD_END` command.
`Reserved`	`u16` value reserved for future use

Return Value

In debug mode, the returned values from calling `adi_ebiu_Init` to initialize SDRAM are:

`ADI_EBIU_RESULT_BAD_COMMAND`	Command-value pair is invalid.
`ADI_EBIU_RESULT_FAILED`	Not all required items are initialized.
`ADI_EBIU_RESULT_ALREADY_INITIALIZED`	EBIU module is already initialized.
`ADI_EBIU_RESULT_INVALID_SDRAM_SCTLE`	Invalid `SCTLE` value specified.
`ADI_EBIU_RESULT_INVALID_SDRAM_MODULE`	Invalid memory module type is specified.
`ADI_EBIU_RESULT_INVALID_SDRAM_BANK_SIZE`	Invalid bank size is specified.

`ADI_EBIU_RESULT_INVALID_SDRAM_COL_WIDTH`	Invalid column address width is specified.
`ADI_EBIU_RESULT_INVALID_SDRAM_TWRMIN`	Invalid `TWRMIN` value is specified.
`ADI_EBIU_RESULT_INVALID_SDRAM_EMREN`	Invalid `EMREN` value is specified.
`ADI_EBIU_RESULT_INVALID_SDRAM_PASR`	Invalid `PASR` value is specified.
`ADI_EBIU_RESULT_INVALID_SDRAM_TCSR`	Invalid `TCSR` value is specified.

In debug mode, the returned values from calling `adi_ebiu_Init` to initialize DDR are:

`ADI_EBIU_RESULT_SUCCESS`	Generic success
`ADI_EBIU_RESULT_FAILED`	Generic failure
`ADI_EBIU_RESULT_BAD_COMMAND`	Invalid control command
`ADI_EBIU_RESULT_INVALID_DDR_MODULE`	Invalid SDRAM module type
`ADI_EBIU_RESULT_INVALID_DDR_REFI`	Invalid auto-refresh interval
`ADI_EBIU_RESULT_INVALID_DDR_RFC`	Invalid auto-refresh command period
`ADI_EBIU_RESULT_INVALID_DDR_RP`	Invalid precharge to active interval
`ADI_EBIU_RESULT_INVALID_DDR_RAS`	Invalid active to precharge interval
`ADI_EBIU_RESULT_INVALID_DDR_RC`	Invalid active to active interval
`ADI_EBIU_RESULT_INVALID_DDR_WTR`	Invalid write to read interval
`ADI_EBIU_RESULT_INVALID_DDR_DEVICE_SIZE`	Invalid device size
`ADI_EBIU_RESULT_INVALID_DDR_DEVICE_WIDTH`	Invalid device width
`ADI_EBIU_RESULT_INVALID_DDR_EXTERNAL_BANKS`	Invalid number of external banks
`ADI_EBIU_RESULT_INVALID_DDR_DATA_WIDTH`	Invalid data width
`ADI_EBIU_RESULT_INVALID_DDR_WR`	Invalid write recovery time

`ADI_EBIU_RESULT_INVALID_DDR_MRD`	Invalid mode register selection
`ADI_EBIU_RESULT_INVALID_DDR_RCD`	Invalid active to R/W interval
`ADI_EBIU_RESULT_INVALID_DDR_CAS`	Invalid delay R/W to valid data
`ADI_EBIU_RESULT_INVALID_DDR_PASR`	Invalid partial array self-refresh request
`ADI_EBIU_RESULT_INVALID_DDR_SOFT_RESET`	Invalid soft reset request
`ADI_EBIU_RESULT_INVALID_DDR_SELF_REFRESH_REQUEST`	Invalid self-refresh request
`ADI_EBIU_RESULT_INVALID_DDR_MOBILE_DDR_ENABLE`	Invalid mobile DDR enable request
`ADI_EBIU_RESULT_ALREADY_INITIALIZED`	EBIU service already initialized

In debug mode, the returned values from calling `adi_ebiu_Init` to initialize the asynchronous memory controller are:

`ADI_EBIU_RESULT_SUCCESS`	Generic success
`ADI_EBIU_RESULT_FAILED`	Generic failure
`ADI_EBIU_RESULT_INVALID_ASYNCH_CLKOUT_ENABLE`	Invalid selection for `CLKOUT` enable
`ADI_EBIU_RESULT_INVALID_ASYNCH_BANK_ENABLE`	Invalid selection for bank enable
`ADI_EBIU_RESULT_INVALID_ASYNCH_BANK_NUMBER`	Invalid bank number specified in command argument
`ADI_EBIU_RESULT_INVALID_ASYNCH_BANK_16_BIT_PACKING_ENABLE`	For ADSP-BF561 only. Invalid specification for 16-bit packing enable
`ADI_EBIU_RESULT_INVALID_ASYNCH_BANK_TRANSITION_TIME`	Invalid transition time
`ADI_EBIU_RESULT_INVALID_ASYNCH_BANK_READ_ACCESS_TIME`	Invalid read access time
`ADI_EBIU_RESULT_INVALID_ASYNCH_BANK_WRITE_ACCESS_TIME`	Invalid write access time

`ADI_EBIU_RESULT_INVALID_ASYNCH_BANK_SETUP_TIME`	Invalid setup time
`ADI_EBIU_RESULT_INVALID_ASYNCH_BANK_HOLD_TIME`	Invalid hold time
`ADI_EBIU_RESULT_INVALID_ASYNCH_BANK_ARDY_ENABLE`	Invalid selection for `ARDY` enable
`ADI_EBIU_RESULT_INVALID_ASYNCH_BANK_ARDY_ POLARITY`	Invalid selection for `ARDY` polarity
`ADI_EBIU_RESULT_ALREADY_INITIALIZED`	EBIU service already initialized

adi_ebiu_LoadConfig

Description

The `adi_ebiu_LoadConfig` 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`	Address of the memory area where the current configuration is 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

`ADI_EBIU_RESULT_SUCCESS`	Process completed successfully.
`ADI_EBIU_RESULT_NO_MEMORY`	Value `szConfig` is too small.
`ADI_EBIU_RESULT_NOT_INITIALIZED`	SDRAM has not been successfully initialized.

adi_ebiu_SaveConfig

Description

The `adi_ebiu_SaveConfig()` 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`	Address of the memory location where the current configuration is 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

`ADI_EBIU_RESULT_SUCCESS`	Process completed successfully.
`ADI_EBIU_RESULT_NO_MEMORY`	Value `szConfig` is too small.
`ADI_EBIU_RESULT_NOT_INITIALIZED`	SDRAM has not been successfully initialized.

adi_ebiu_Terminate

Description

The `adi_ebiu_Terminate()` function terminates the use of the EBIU module.

Prototype

```
ADI_EBIU_RESULT adi_ebiu_Terminate(void);
```

Argument

The function takes no arguments.

Return Value

`ADI_EBIU_RESULT_SUCCESS`	Process completed successfully.

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`. Note that SDRAM-related result codes typically begin with the text `ADI_EBIU_RESULT_INVALID_SDRAM` while DDR-related result codes typically begin with the text `ADI_EBIU_RESULT_INVALID_DDR`. Table 4-3 lists possible values.

Table 4-3. EBIU Module Function Result Codes

Result Code	Explanation
`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_NOT_INITIALIZED`	Function call 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.

Table 4-3. EBIU Module Function Result Codes (Cont'd)

Result Code	Explanation
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	Invalid value for TWRMIN is specified and causes TWR to be greater than 3.
ADI_EBIU_RESULT_NO_MEMORY	Insufficient memory to load/save configuration.
ADI_EBIU_RESULT_INVALID_EZKIT	Invalid EZ-KIT revision
ADI_EBIU_RESULT_INVALID_SDRAM_SCTLE	Invalid SCTLE value
ADI_EBIU_RESULT_INVALID_SDRAM_MODULE	Invalid SDRAM module type

Table 4-3. EBIU Module Function Result Codes (Cont'd)

Result Code	Explanation
`ADI_EBIU_RESULT_INVALID_IVG`	Invalid IVG level supplemental interrupt (for dual-core processors only
`ADI_EBIU_RESULT_INVALID_SDRAM_BANK`	Invalid bank number given
`ADI_EBIU_RESULT_INVALID_SDRAM_SCK1E`	Invalid `SCK1E` value
`ADI_EBIU_RESULT_INVALID_DDR_MODULE`	Invalid SDRAM (DDR) module type
`ADI_EBIU_RESULT_INVALID_DDR_REFI`	Invalid refresh interval
`ADI_EBIU_RESULT_INVALID_DDR_RFC`	Invalid auto-refresh command
`ADI_EBIU_RESULT_INVALID_DDR_RP`	Invalid precharge to active interval
`ADI_EBIU_RESULT_INVALID_DDR_RAS`	Invalid active to precharge interval
`ADI_EBIU_RESULT_INVALID_DDR_RC`	Invalid active to active interval
`ADI_EBIU_RESULT_INVALID_DDR_WTR`	Invalid write to read interval
`ADI_EBIU_RESULT_INVALID_DDR_DEVICE_SIZE`	Invalid device size
`ADI_EBIU_RESULT_INVALID_DDR_DEVICE_WIDTH`	Invalid device width
`ADI_EBIU_RESULT_INVALID_DDR_EXTERNAL_BANKS`	Invalid number of external banks
`ADI_EBIU_RESULT_INVALID_DDR_DATA_WIDTH`	Invalid data width
`ADI_EBIU_RESULT_INVALID_DDR_WR`	Invalid write recovery time
`ADI_EBIU_RESULT_INVALID_DDR_MRD`	Invalid mode register selection
`ADI_EBIU_RESULT_INVALID_DDR_RCD`	Invalid active to R/W interval
`ADI_EBIU_RESULT_INVALID_DDR_CAS`	Invalid R/W to valid data interval
`ADI_EBIU_RESULT_INVALID_DDR_PASR`	Invalid partial array self-refresh request
`ADI_EBIU_RESULT_INVALID_DDR_SOFT_RESET`	Invalid soft reset request

Table 4-3. EBIU Module Function Result Codes (Cont'd)

Result Code	Explanation
ADI_EBIU_RESULT_INVALID_DDR_MOBILE_ENABLE	Invalid mobile DDR enable request
ADI_EBIU_RESULT_INVALID_DDR_SELF_REFRESH_REQUEST	Invalid self refresh request

ADI_EBIU_SDRAM_BANK_VALUE

The ADI_EBIU_SDRAM_BANK_VALUE structure specifies the settings that are 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;
        } value;
} ADI_EBIU_SDRAM_BANK_VALUE;
```

See "ADI_EBIU_SDRAM_BANK_SIZE" on page 4-40 and "ADI_EBIU_SDRAM_BANK_COL_WIDTH" on page 4-41 for details of the size and width fields.

The bank field is intended for use only with Blackfin processors that have multiple SDRAM banks.

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 as follows.

`ADI_EBIU_TIMING_UNIT_MILLISEC`	Time value specified by the associated value in the `ADI_EBIU_TIME` structure is in milliseconds (ms)
`ADI_EBIU_TIMING_UNIT_MICROSEC`	Time value specified by the associated value in the `ADI_EBIU_TIME` structure is in microseconds (ms)
`ADI_EBIU_TIMING_UNIT_NANOSEC`	Time value specified by the associated value in the `ADI_EBIU_TIME` structure is in nanoseconds (ns)
`ADI_EBIU_TIMING_UNIT_PICOSEC`	Time value specified by the associated value in the `ADI_EBIU_TIME` structure is in picoseconds (ps)
`ADI_EBIU_TIMING_UNIT_FEMTOSEC`	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 board has a minimum `TWR` value of one `SCLK` cycle and 7.5 ns. The time value must be passed as 7500 ps. 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 to correctly set the SDRAM control registers are specified on the appropriate processor's 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 as:

```
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 4-29.

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

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

ADI_EBIU_ASYNCH_BANK_TIMING

The asynchronous memory controller supports a number of different interfaces, therefore a structure is provided which allows the bank specific timing parameters to be specified either in cycles, or in timing units, *but not both*.

If the parameter is specified in cycles, the value is written directly to the register. If the value is specified in timing units, it is converted to cycles based on the presence of a 133 MHz system clock, and the converted value is written to the register.

Timing values used to set the asynchronous memory control registers should be derived from the appropriate data sheet for the type of memory device used.

The structure used to specify the timing parameters for the asynchronous memory interface is shown below. It contains two other structures: the enumeration "ADI_EBIU_BANK_NUMBER" on page 4-46 and the structure "ADI_EBIU_TIMING_VALUE" on page 4-30.

```
typedef struct ADI_EBIU_ASYNCH_BANK_TIMING
{
        ADI_EBIU_BANK_NUMBER        bank_number;
        ADI_EBIU_TIMING_VALUE       bank_time;
} ADI_EBIU_ASYNCH_BANK_TIMING;
```

ADI_EBIU_ASYNCH_BANK_VALUE

Because many of the EBIU parameters are bank specific, and specify a binary value such as `enabled` and `disabled`, a structure is provided which contains a bank number along with a union of three different enumerations that have two possible values.

The `ADI_EBIU_ASYNCH_BANK_VALUE` structure, shown below, is used for the `ARDY` polarity, which is either low or high. (See "ADI_EBIU_ASYNCH_BANK_ARDY_POLARITY" on page 4-48.) It is used for the `ARDY` enable, which is either enabled or disabled. (See "ADI_EBIU_ASYNCH_BANK_ARDY_ENABLE" on page 4-48.) It is also used for the 16-bit packing enable field (*for the ADSP-BF561 only*), which is either 16-bit packing enabled or 32-bit packing disabled. (See "ADI_EBIU_ASYNCH_BANK_DATA_PATH" on page 4-47.)

```
typedef struct ADI_EBIU_ASYNCH_BANK_VALUE
{
   u32 bank_number;
   union
```

```
    {
        ADI_EBIU_ASYNCH_BANK_ARDY_POLARITY      ardy_polarity;
        ADI_EBIU_ASYNCH_BANK_ARDY_ENABLE        ardy_enable;
#if defined(__ADSP_TETON__)
        ADI_EBIU_ASYNCH_BANK_DATA_PATH          data_path;
#endif
    } value;
} ADI_EBIU_ASYNCH_BANK_VALUE;
```

Setting Control Values in the EBIU Module

To set control values in the EBIU module, the user passes command-value pairs (of the 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 used in an `ADI_EBIU_COMMAND_PAIR` couplet to set a configuration value in calls to `adi_ebiu_Init` and `adi_ebiu_Control`. Note that SDRAM-related commands typically begin with the text `ADI_EBIU_CMD_SET_SDRAM` while DDR-related commands typically begin with the text `ADI_EBIU_CMD_SET_DDR`.

Table 4-4. 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	ADI_EBIU_SDRAM_EBUFE value specifying whether external buffers are used when several SDRAM devices are used. See "ADI_EBIU_SDRAM_EBUFE" on page 4-44.
ADI_EBIU_CMD_SET_SDRAM_FBBRW	ADI_EBIU_SDRAM_FBBRW value specifying whether to enable/disable fast back-to-back read/write operations. See "ADI_EBIU_SDRAM_FBBRW" on page 4-45.
ADI_EBIU_CMD_SET_SDRAM_CDDBG	ADI_EBIU_SDRAM_CDDBG value specifying 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 4-46.
ADI_EBIU_CMD_SET_SDRAM_PUPSD	ADI_EBIU_SDRAM_PUPSD value specifying whether the power-up start sequence is delayed by 15 SCLK cycles. See "ADI_EBIU_SDRAM_PUPSD" on page 4-44.
ADI_EBIU_CMD_SET_SDRAM_PSM	ADI_EBIU_SDRAM_PSM value specifying the order of events in the power-up start sequence. See "ADI_EBIU_SDRAM_PSM" on page 4-45.
ADI_EBIU_CMD_SET_ASYNCH_BANK_TRANSITION_TIME	ADI_EBIU_ASYNCH_BANK_TIMING value specifying an ADI_EBIU_BANK_NUMBER and an ADI_EBIU_TIMING_VALUE that specifies the transition time in either cycles or timing units. See "ADI_EBIU_ASYNCH_BANK_TIMING" on page 4-30.
ADI_EBIU_CMD_SET_ASYNCH_BANK_READ_ACCESS_TIME	ADI_EBIU_ASYNCH_BANK_TIMING value specifying an ADI_EBIU_BANK_NUMBER and an ADI_EBIU_TIMING_VALUE that specifies the read access time in either cycles or timing units. See "ADI_EBIU_ASYNCH_BANK_TIMING" on page 4-30.

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

Command	Associated Data Value
ADI_EBIU_CMD_SET_ASYNCH_BANK_WRITE_ACCESS_TIME	ADI_EBIU_ASYNCH_BANK_TIMING value specifying an ADI_EBIU_BANK_NUMBER and an ADI_EBIU_TIMING_VALUE that specifies the write access time in either cycles or timing units. See "ADI_EBIU_ASYNCH_BANK_TIMING" on page 4-30.
ADI_EBIU_CMD_SET_ASYNCH_BANK_SETUP_TIME	ADI_EBIU_ASYNCH_BANK_TIMING value specifying an ADI_EBIU_BANK_NUMBER and an ADI_EBIU_TIMING_VALUE that specifies the setup time in either cycles or timing units. See "ADI_EBIU_ASYNCH_BANK_TIMING" on page 4-30.
ADI_EBIU_CMD_SET_ASYNCH_BANK_HOLD_TIME	ADI_EBIU_ASYNCH_BANK_TIMING value specifying an ADI_EBIU_BANK_NUMBER and an ADI_EBIU_TIMING_VALUE that specifies the hold time in either cycles or timing units. See "ADI_EBIU_ASYNCH_BANK_TIMING" on page 4-30.
Commands valid only when passed to the adi_ebiu_Init function.	
ADI_EBIU_CMD_SET_SDRAM_MODULE	ADI_EBIU_SDRAM_MODULE_TYPE value containing the configured Micron memory module. This value applies to all banks in use. See "ADI_EBIU_SDRAM_MODULE_TYPE" on page 4-41.
ADI_EBIU_CMD_SET_SDRAM_BANK_SIZE	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 4-28 and "ADI_EBIU_SDRAM_BANK_SIZE" on page 4-40.

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

Command	Associated Data Value
ADI_EBIU_CMD_SET_SDRAM_BANK_COL_WIDTH	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 4-28 and "ADI_EBIU_SDRAM_BANK_COL_WIDTH" on page 4-41.
ADI_EBIU_CMD_SET_SDRAM_CL_THRESHOLD	u32 value specifying the SCLK frequency threshold, which determines the CAS latency value to use.
ADI_EBIU_CMD_SET_SDRAM_TRASMIN	ADI_EBIU_TIME value setting the minimum TRAS value described in the appropriate Blackfin processor data sheet of the appropriate SDRAM. See "ADI_EBIU_TIME" on page 4-29.
ADI_EBIU_CMD_SET_SDRAM_TRPMIN	ADI_EBIU_TIME value setting the minimum TRP value as described in the appropriate Blackfin processor data sheet of the appropriate SDRAM. See "ADI_EBIU_TIME" on page 4-29.
ADI_EBIU_CMD_SET_SDRAM_TRCDMIN	ADI_EBIU_TIME value setting the minimum TRCD value as described in the appropriate Blackfin processor data sheet of the appropriate SDRAM. See "ADI_EBIU_TIME" on page 4-29.
ADI_EBIU_CMD_SET_SDRAM_TWRMIN	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 4-30.
ADI_EBIU_CMD_SET_SDRAM_REFRESH	Address of an ADI_EBIU_TIMING_VALUE structure containing the maximum t_{REF} value described in the appropriate Blackfin processor data sheet of the appropriate SDRAM. See "ADI_EBIU_TIME" on page 4-29.
ADI_EBIU_CMD_SET_SDGCTL_REG	u32 word containing the entire contents of the EBIU_SDGCTL register

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

Command	Associated Data Value
ADI_EBIU_CMD_SET_SDBCTL_REG	u16 word containing the entire contents of the EBIU_SDBCTL register
ADI_EBIU_CMD_SET_SDRAM_EMREN	ADI_EBIU_SDRAM_EMREN value specifying whether low power (2.5 V) SDRAM is used. See "ADI_EBIU_SDRAM_MODULE_TYPE" on page 4-41.
ADI_EBIU_CMD_SET_SDRAM_PASR	ADI_EBIU_SDRAM_PASR value specifying which banks are refreshed. Applicable only to low power SDRAM. See "ADI_EBIU_CMD_SET_SDRAM_SCTLE" on page 4-41.
ADI_EBIU_CMD_SET_SDRAM_TCSR	ADI_EBIU_SDRAM_TCSR value specifying the temperature-compensated, self-refresh value. This command can only be used for low power SDRAM. See "ADI_EBIU_SDRAM_PASR" on page 4-42.
ADI_EBIU_CMD_SET_SDRAM_SCTLE	ADI_EBIU_SDRAM_SCTLE value specifying whether the SDC is enabled. See "ADI_EBIU_CMD_SET_SDRAM_SCTLE" on page 4-41.
ADI_EBIU_CMD_SET_DDR_DATA_WIDTH	Set DDR width of data
ADI_EBIU_CMD_SET_DDR_EXTERNAL_BANKS	Set number of DDR external banks
ADI_EBIU_CMD_SET_DDR_DEVICE_WIDTH	Set DDR width of device
ADI_EBIU_CMD_SET_DDR_DEVICE_SIZE	Set size of device
ADI_EBIU_CMD_SET_DDR_REFI	Set DDR auto-refresh interval
ADI_EBIU_CMD_SET_DDR_RFC	Set auto-refresh command period
ADI_EBIU_CMD_SET_DDR_CAS	Set DDR CAS latency: cycles from R/W to first valid data
ADI_EBIU_CMD_SET_DDR_RCD	Set interval between active command and R/W assertion
ADI_EBIU_CMD_SET_DDR_RC	Set interval between successive DDR activate commands

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

Command	Associated Data Value
ADI_EBIU_CMD_SET_DDR_RAS	Set DDR active to precharge interval
ADI_EBIU_CMD_SET_DDR_RP	Set DDR precharge to active interval
ADI_EBIU_CMD_SET_DDR_MRD	Set DDR interval between setting of mode register and next command
ADI_EBIU_CMD_SET_DDR_WTR	Set DDR interval between write and read command
ADI_EBIU_CMD_SET_DDR_WR	Set DDR write recovery time
ADI_EBIU_CMD_SET_DDR_PASR	Set DDR partial array self-refresh for mobile DDR only. See "ADI_EBIU_DDR_PASR" on page 4-51.
ADI_EBIU_CMD_SET_DDR_SOFT_RESET	Issue DDR soft reset
ADI_EBIU_CMD_MOBILE_DDR_ENABLE	Enable mobile DDR
ADI_EBIU_CMD_SET_FREQ_AS_MHZ	Sets DDR frequency units to megahertz
ADI_EBIU_CMD_SET_ASYNCH_BANK_ARDY_ENABLE	ADI_EBIU_ASYNCH_BANK_VALUE specifying a bank number and an ADI_EBIU_ASYNCH_BANK_ARDY_ENABLE that specifies whether the ARDY input will be sampled for this bank. See "ADI_EBIU_ASYNCH_BANK_ARDY_ENABLE" on page 4-48.
ADI_EBIU_CMD_SET_ASYNCH_BANK_ARDY_POLARITY	ADI_EBIU_ASYNCH_BANK_VALUE specifying a bank number and an ADI_EBIU_ASYNCH_BANK_ARDY_POLARITY that specifies the polarity of the ARDY input sample that indicates the completion of the access time. See "ADI_EBIU_ASYNCH_BANK_ARDY_POLARITY" on page 4-48.

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

Command	Associated Data Value
ADI_EBIU_CMD_SET_ASYNCH_BANK_16_BIT_PACKING_ENABLE	*For ADSP-BF561 processors only* ADI_EBIU_ASYNCH_BANK_VALUE specifying a bank number and an ADI_EBIU_ASYNCH_BANK_DATA_PATH that specifies whether or not 16-bit packing is enabled. See "ADI_EBIU_ASYNCH_BANK_DATA_PATH" on page 4-47.
ADI_EBIU_CMD_SET_ASYNCH_BANK_ENABLE	ADI_EBIU_ASYNCH_BANK_ENABLE value specifying which banks to enable. See "ADI_EBIU_ASYNCH_BANK_ENABLE" on page 4-47.
ADI_EBIU_CMD_SET_ASYNCH_CLKOUT_ENABLE	ADI_EBIU_ASYNCH_CLKOUT value specifying whether to enable or disable CLKOUT in the asynchronous global control register. See "ADI_EBIU_ASYNCH_CLKOUT" on page 4-47.
ADI_EBIU_CMD_SET_ASYNCH_AMGCTL	16-bit numeric value used to simultaneously set all the fields of the asynchronous memory global control register
ADI_EBIU_CMD_SET_ASYNCH_AMBCTL0	32-bit numeric value used to simultaneously set all the fields of the asynchronous memory bank control register 0 at once
ADI_EBIU_CMD_SET_ASYNCH_AMBCTL1	32-bit numeric value used to simultaneously set all the fields of the asynchronous memory bank control register 1 at once
Commands valid only when passed to the adi_ebiu_Control 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	ADI_EBIU_SDRAM_ENABLE value enabling/disabling external SDRAM. Automatically set upon initialization. See "ADI_EBIU_SDRAM_ENABLE" on page 4-40.

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

Command	Associated Data Value
`ADI_EBIU_CMD_SET_SDRAM_SRFS`	`ADI_EBIU_SDRAM_SRFS` value enabling/disabling self-refresh of SDRAM during inactivity. See "ADI_EBIU_SDRAM_TCSR" on page 4-43.
`ADI_EBIU_CMD_DDR_SELF_REFRESH_REQUEST`	Request DDR self-refresh

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 *Engineer-to-Engineer Note EE-210*[1].

[1] Refer to *SDRAM Selection Guidelines and Configuration for ADI Processors*, EE-210, Rev 2, August 2004.

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.

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

The default value is specified by the following macro:

```
#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`	6 4MB 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 an SDRAM module type when the command `ADI_EBIU_CMD_SET_SDRAM_MODULE` is used to initialize the SDRAM controller. The enumerator values contain relevant module information such as the speed grade and configuration settings. The external memory bank size must also be specified using the `ADI_EBIU_CMD_SET_SDRAM_BANK_SIZE` command. Because Analog Devices EZ-KIT Lite boards include SDRAM supplied by Micron, this information applies only to Micron parts. The list of valid enumeration values are found in the API header file, `adi_pwr.h`.

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 (2.5 V) SDRAM is 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 (2.5 V) SDRAM is used, this enumeration specifies the banks to refresh. This enumeration corresponds to the `PASR` bits in the `EBIU_SDGCTL` register.

`ADI_EBIU_SDRAM_PASR_ALL`	All four SDRAM banks are refreshed.
`ADI_EBIU_SDRAM_PASR_INT01`	Internal SDRAM banks 0 and 1 are refreshed.
`ADI_EBIU_PASR_INT0_ONLY`	Only internal bank 0 is 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 (2.5 V) 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`	SDRAM banks are refreshed if the temperature exceeds 45° C.
`ADI_EBIU_SDRAM_TCSR_85DEG`	SDRAM banks are refreshed if the temperature exceeds 85° C.

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 whether 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 “deep sleep” via the power management module. For more information, see “Power Management Module” on page 3-1.

`ADI_EBIU_SDRAM_SRFS_DISABLE`	Disables SDRAM self-refresh on inactivity.
`ADI_EBIU_SDRAM_SRFS_ENABLE`	Enables 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 the EBIU uses external buffers when several SDRAM devices are used in parallel. This enumeration corresponds to the `EBUFE` bit in the `EBIU_SDGCTL` register.

`ADI_EBIU_SDRAM_EBUFE_DISABLE`	Disables the use of external buffers when several SDRAM devices are used in parallel.
`ADI_EBIU_SDRAM_EBUFE_ENABLE`	Enables the use of external buffers when several SDRAM devices are 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 the power-up start sequence is 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`	Power-up start sequence is 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`	SDC performs a `Precharge All` command, followed by eight auto-refresh cycles, and then a `Load Mode Register` command.
`ADI_EBIU_SDRAM_PSM_REFRESH_LAST`	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 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 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`	Disables the SDRAM control signals when the external memory interface is granted to an external controller.
`ADI_EBIU_SDRAM_CDDBG_ENABLE`	Enables 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
```

ADI_EBIU_BANK_NUMBER

This enumeration is used to specify the bank number 0, 1, 2, or 3, for which the associated command applies. It can also be used to specify all banks.

`ADI_EBIU_BANK_0`	Command is for bank 0.
`ADI_EBIU_BANK_1`	Command is for bank 1.
`ADI_EBIU_BANK_2`	Command is for bank 2.
`ADI_EBIU_BANK_3`	Command is for bank 3.
`ADI_EBIU_BANK_ALL`	Command is for ALL four banks.

ADI_EBIU_ASYNCH_BANK_ENABLE

This enumeration specifies which banks are being enabled. It corresponds to the `AMBEN` bits in the asynchronous memory global control register.

`ADI_EBIU_ASYNCH_DISBALE_ALL`	Disables all banks.
`ADI_EBIU_ASYNCH_BANK0`	Enables bank 0.
`ADI_EBIU_ASYNCH_BANK0_1`	Enables banks 0 and 1.
`ADI_EBIU_ASYNCH_BANK0_1_2`	Enables banks 0, 1, and 2.
`ADI_EBIU_ASYNCH_BANK0_1_2_3`	Enables all banks.

ADI_EBIU_ASYNCH_CLKOUT

This enumeration specifies whether `CLKOUT` is enabled for external memory access. It corresponds to the `AMCKEN` bits in the asynchronous memory global control register.

`ADI_EBIU_ASYNCH_CLKOUT_DISABLE`	`CLKOUT` is disabled.
`ADI_EBIU_ASYNCH_CLKOUT_ENABLE`	`CLKOUT` is enabled.

ADI_EBIU_ASYNCH_BANK_DATA_PATH

This enumeration is for the ADSP-BF561 only. It specifies whether 16-bit packing is enabled on the asynchronous memory bus. It corresponds to the `BXPEN` bits in the asynchronous memory global control register, where `X` is the bank number.

`ADI_EBIU_ASYNCH_BANK_DATA_PATH_32`	16-bit packing in NOT enabled.
`ADI_EBIU_ASYNCH_BANK_DATA_PATH_16`	16-bit packing is enabled.

ADI_EBIU_ASYNCH_BANK_ARDY_ENABLE

Each asynchronous bank can be programmed to sample the ARDY input, which allows the bank access time to be extended. Sampling the ARDY pin determines how long to extend the access time. This enumeration specifies whether or not the ARDY signal will be sampled. It corresponds to the BXRDYEN bit (where X is the bank number) in the asynchronous memory bank control 0 register (for banks 0 and 1) or the asynchronous memory bank control 1 register (for banks 2 and 3).

ADI_EBIU_ASYNCH_ARDY_DISABLE	Sampling of ARDY is disabled.
ADI_EBIU_ASYNCH_ARDY_ENABLE	Sampling of ARDY is enabled.

ADI_EBIU_ASYNCH_BANK_ARDY_POLARITY

This enumeration specifies, if ARDY is enabled, whether the access time is complete when the ARDY signal is low or high. It corresponds to the BXRDYPOL bit (where X is the bank number) in the asynchronous memory bank control 0 register (for banks 0 and 1) or the asynchronous memory bank control 1 register (for banks 2 and 3).

ADI_EBIU_ASYNCH_ARDY_POLARITY_LOW	Transaction is complete if ARDY is low.
ADI_EBIU_ASYNCH_ARDY_POLARITY_HIGH	Transaction is complete if ARDY is high.

ADI_EBIU_ASYNCH_HOLD_TIME

The hold time for the asynchronous memory controller is specified in the bank_time field of an "ADI_EBIU_ASYNCH_BANK_TIMING" structure. That field is of type "ADI_EBIU_TIMING_VALUE", which in this case can either specify a number of cycles or an "ADI_EBIU_TIME" value, *but not both*.

When cycles are used, the ADI_EBIU_ASYNCH_HOLD_TIME enumeration specifies the number of cycles of hold time. It corresponds to the BXHT bit (where X is the bank number) in the asynchronous memory bank control 0 register (for banks 0 and 1) or the asynchronous memory bank control 1 register (for banks 2 and 3).

ADI_EBIU_ASYNCH_HT_0_CYCLES	0 cycles hold time
ADI_EBIU_ASYNCH_HT_1_CYCLES	1 cycles hold time
ADI_EBIU_ASYNCH_HT_2_CYCLES	2 cycles hold time
ADI_EBIU_ASYNCH_HT_3_CYCLES	3 cycles hold time

ADI_EBIU_ASYNCH_SETUP_TIME

The setup time for the asynchronous memory controller is specified in the bank_time field of an "ADI_EBIU_ASYNCH_BANK_TIMING" structure. That field is of type "ADI_EBIU_TIMING_VALUE", which in this case can either specify a number of cycles or an "ADI_EBIU_TIME" value, *but not both*.

When cycles are used, the ADI_EBIU_ASYNCH_SETUP_TIME enumeration specifies the number of cycles of setup time. It corresponds to the BXST bit (where X is the bank number) in the asynchronous memory bank control 0 register (for banks 0 and 1) or the asynchronous memory bank control 1 register (for banks 2 and 3).

ADI_EBIU_ASYNCH_ST_4_CYCLES	4 cycles setup time
ADI_EBIU_ASYNCH_ST_1_CYCLES	1 cycles setup time
ADI_EBIU_ASYNCH_ST_2_CYCLES	2 cycles setup time
ADI_EBIU_ASYNCH_ST_3_CYCLES	3 cycles setup time

ADI_EBIU_ASYNCH_TRANSITION_TIME

The transition time for the asynchronous memory controller is specified in the `bank_time` field of an "ADI_EBIU_ASYNCH_BANK_TIMING" structure. That field is of type "ADI_EBIU_TIMING_VALUE", which in this case can either specify a number of cycles or an "ADI_EBIU_TIME" value, *but not both*. When cycles are used, the `ADI_EBIU_ASYNCH_TRANSITION_TIME` enumeration specifies the number of cycles of transition time. It corresponds to the `BXHT` bit (where `X` is the bank number) in the asynchronous memory bank control 0 register (for banks 0 and 1) or the asynchronous memory bank control 1 register (for banks 2 and 3).

`ADI_EBIU_ASYNCH_TT_4_CYCLES`	4 cycles transition time
`ADI_EBIU_ASYNCH_TT_1_CYCLES`	1 cycles transition time
`ADI_EBIU_ASYNCH_TT_2_CYCLES`	2 cycles transition time
`ADI_EBIU_ASYNCH_TT_3_CYCLES`	3 cycles transition time

ADI_EBIU_DDR_MOBILE_DS

This enumeration specifies the drive strength for the memory device. The value is written to the `DS` field of the `EBIU_DDRCTL3` register. This enumeration is used for mobile DDR products only. For non-mobile DDR products, see "ADI_EBIU_DDR_DS". The possible enumeration values are shown below.

`ADI_EBIU_DDR_DS_1`	00: Full strength drive
`ADI_EBIU_DDR_DS_2`	01: Half strength drive
`ADI_EBIU_DDR_DS_4`	10: Quarter strength drive
`ADI_EBIU_DDR_DS_8`	11: One-eighth strength drive

ADI_EBIU_DDR_DS

This enumeration specifies the drive strength for the memory device. The value is written to the `DS` field of the `EBIU_DDRCTL3` register. This enumeration is used for non-mobile DDR products only. For mobile DDR products, see "ADI_EBIU_DDR_MOBILE_DS". The possible enumeration values are shown below.

`ADI_EBIU_DDR_DS_FULL`	00: Full strength drive
`ADI_EBIU_DDR_DS_REDUCED`	01: Reduced strength drive (default)

ADI_EBIU_DDR_PASR

This enumeration specifies the partial array self-refresh value written to the `PASR` field of the `EBIU_DDRCTL3` register. This field is available only on mobile DDR products. The possible enumeration values are shown below.

`ADI_EBIU_DDR_PASR_1`	0: Full array (all banks)
`ADI_EBIU_DDR_PASR_2`	1: Half array
`ADI_EBIU_DDR_PASR_4`	2: Quarter array
`ADI_EBIU_DDR_PASR_RESERVED3`	3: (Reserved value)
`ADI_EBIU_DDR_PASR_RESERVED4`	4: (Reserved value)
`ADI_EBIU_DDR_PASR_8`	5: Eighth array
`ADI_EBIU_DDR_PASR_16`	6: Sixteenth array

5 DEFERRED CALLBACK MANAGER

This chapter describes the deferred callback (DCB) manager used by the application developer to manage the deferred execution of function calls. A detailed description of the application programming interface (API) provided by the deferred callback manager is included.

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 ISRs 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 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 manager provides this service by managing one or more queues of deferred callbacks, such that their invocation typically 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 deferred callback (DCB) manager is designed to operate 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 the DCB manager is implemented in standalone mode or in conjunction with one of the above RTOSs also impacts the number and size of queues. When implemented in conjunction with VDK, the DCB manager can support only 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 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", along with code segments illustrating its use in standalone mode.Implications for its use in conjunction with an RTOS are given in "Interoperability With an RTOS" on page 5-7.

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 `typedef` statements use the `ADI_DCB_` prefix, and functions and global variables use the lowercase `adi_dcb_` equivalent.

Using the Deferred Callback Manager

The operation of the DCB manager comprises the following functions.

- 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

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, API calls to the DCB manager are the same: a queue is initialized with a call to `adi_dcb_Open`, and callbacks are 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 a 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, standalone operation requires the initialization of the interrupt manager prior to initializing the DCB manager. On the assumption that the sample application requires that only one interrupt handler be defined per IVG level, initialize the interrupt manager with 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 (in this case, five entries) and the desired IVG level at which the queue operates. This level is ignored when used in 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.
);
```

In the example above, `event` typically defines an event (for example, DMA completion) and `data` typically points to an appropriate location in memory that is meaningful within the context of the callback function. Within the DMA manager context, this argument is the address of an appropriate descriptor or data buffer.

For any reason, flushing the queue of entries for the above callback can be achieved in one of two ways: by calling the `adi_dcb_Remove` function directly or by calling 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 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 provided only 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 described in this section 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 coincides with the address of the queue server structure to which the callback is posted. This is ignored in standalone mode.
`IvgLevel`	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 depends 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 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	Interrupt level at which callbacks are dispatched
Dispatcher	Mandatory address of the adi_dcb_DispatchCallbacks function
hServer	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 critical regions are required within a callback function, you must be aware of any restrictions imposed by the underlying RTOS. For example, VDK-based applications are prohibited from calling `PushCriticalRegion/PopCriticalRegion` functions from within the 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 the 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, they effect critical regions directly by using the `cli()` and `sti()` built-in functions.

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 – 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

The `adi_dcb_Close()` 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_DCB_RESULT adi_dcb_Close(
      ADI_DCB_HANDLE hServer
);
```

Arguments

`hServer`	Handle of the required queue server to close

Return Value

`ADI_DCB_RESULT_SUCCESS`	Queue successfully closed.
`ADI_DCB_RESULT_NO_SUCH_QUEUE`	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

The `adi_dcb_Control()` function is used to configure/control a deferred callback queue server according to command-value pairs. For more information, see "ADI_DCB_COMMAND_PAIR" on page 5-22.

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.

```
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 5-23 for the complete list of commands and associated values.

Prototype

```
ADI_DCB_RESULT adi_dcb_Control(
        ADI_DCB_HANDLE hServer,
        ADI_DCB_COMMAND Command,
        void *Value
);
```

Arguments

`hServer`	Handle of the required queue server to close.
`Command`	`ADI_DCB_COMMAND` enumeration value specifying the meaning of the associated value argument. See "ADI_DCB_COMMAND" on page 5-23.
`Value`	Required value, a single value, a command-value pair, or a table of command-value pairs

Return Value

`ADI_DCB_RESULT_SUCCESS`	Function completed successfully.
`ADI_DCB_RESULT_NO_SUCH_QUEUE`	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,
        u32             *NumServers
        void            *hCriticalRegionData
);
```

Arguments

`ServerMemData`	Pointer to an area of memory used to hold the data associated with each registered queue server
`szServer`	Length in bytes of memory 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`	Handle to data area containing critical region data. This is passed to `adi_int_EnterCriticalRegion` where it is used internally by the module. See "Interrupt Manager" for further details.

Return Value

`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`	DCB 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.

ⓘ The interrupt manager must be initialized 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`	IVG level at which the DCB manager's dispatcher function operates. This value is ignored in the VDK version of the library.
`QueueMemData`	Pointer to an area of memory used to hold the data associated with the server's entry queue
`szQueue`	Length in bytes of memory 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 is used to uniquely identify the queue server in calls to other API functions within the SSL.

Return Value

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

adi_dcb_Post

Description

The `adi_dcb_Post()` 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 such that higher-priority callbacks run before lower-priority callbacks. 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,
      u32                   Priority;
      ADI_DCB_CALLBACK_FN   Callback,
      void                  *pHandle,
      u32                   u32Arg,
      void                  *pArg
);
```

Arguments

Table 5-1. adi_dcb_Post Arguments

Argument	Explanation
`hServer`	Handle of the required queue server
`Priority`	Priority level at which the callback runs; the lower the number, the higher the priority. There is no real limit on the value supplied.
`Callback`	Address of the client callback function queued
`pHandle`	`void*` address passed as the first argument to the callback function upon its deferred execution. Typically it is a handle address that is meaningful within the context of the callback function. For example, when used within the interrupt handler of the DMA manager, this argument is the `ClientHandle` value defined when the DMA channel was opened.
`u32Arg`	`u32` value passed as the second argument to the callback function upon its deferred execution. (See "ADI_DCB_CALLBACK_FN" on page 5-22.) Typically, it is a value that is meaningful within the context of the callback function. For example, when used within the interrupt handler of the DMA manager, this argument describes the nature of the event that has occurred.
`pArg`	`void*` value passed as the third argument to the callback function upon its deferred execution. (See "ADI_DCB_CALLBACK_FN" on page 5-22.) Typically, it is 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

`ADI_DCB_RESULT_SUCCESS`	Entry was successfully queued.
`ADI_DCB_RESULT_NO_MEMORY`	No vacant queue entry available.
`ADI_DCB_RESULT_NO_SUCH_QUEUE`	Handle provided does not represent a valid queue server.

adi_dcb_Remove

Description

The `adi_dcb_Remove()` function removes entries in the given queue that match 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`	Handle of the required queue server
`Callback`	Address of the client callback function removed. If NULL, then all entries in the queue are removed, otherwise all entries matching the given callback function address are removed.

Return Value

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

adi_dcb_Terminate

Description

The `adi_dcb_Terminate()` function terminates the DCB manager by dissociating the supplied memory (see "adi_dcb_Init" on page 5-14) and critical region data.

Prototype

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

Return Value

`ADI_DCB_RESULT_SUCCESS`	Function completed successfully.

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 data type 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

The ADI_DCB_COMMAND_PAIR 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 5-23.

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.

ADI_DCB_CMD_END	Defines the end of a table of command pairs.
ADI_DCB_CMD_PAIR	Tells adi_dcb_Control that a single command pair is being passed.
ADI_DCB_CMD_TABLE	Tells adi_dcb_Control that a table of command pairs is being passed.
ADI_DCB_CMD_FLUSH_QUEUE	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 (refer to "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 include the following.

`ADI_DCB_RESULT_SUCCESS`	Queue server was successfully initialized.
`ADI_DCB_RESULT_NO_MEMORY`	Insufficient memory for one queue entry was present.
`ADI_DCB_RESULT_QUEUE_IN_USE`	Queue server has already been opened for use by the specified IVG. See "ADI_DCB_COMMAND" on page 5-23.
`ADI_DCB_RESULT_CALL_IGNORED`	DCB manager has already been initialized for this processor core. See "ADI_DCB_COMMAND" on page 5-23.
`ADI_DCB_RESULT_NO_SUCH_QUEUE`	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 features of the direct memory access (DMA) manager and its 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 (MMRs) through the implementation of application programming interface (API) function calls.
- Place no limitations on the type of data transfer. All descriptor types are supported as well as single and circular buffers. Both one-dimensional (1-D) and two-dimensional (2-D) DMA can be used.
- Provide a simple interface to perform block copies of data between different memory locations using both 1-D and 2-D 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 user-supplied callback functions. For example, if an interrupt is raised on each inner loop of a circular 2-D 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 application requires.
- 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 or elsewhere.

  To this end, all enumeration values and `typedef` statements use the `ADI_DMA_` prefix, and functions and global variables use the lowercase `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 processor's DMA controller. The DMA manager supports peripheral DMA to move data to and from the various on-board peripherals and uses memory DMA to move data between the various memory spaces of the Blackfin processor.

The DMA manager is capable of controlling any number of DMA channels. You specify which channels the DMA manager controls. The application can use the remaining channels (those channels not under control of the DMA manager) for any purpose; that is, the channels are controlled independently of the DMA manager.

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

The DMA manager can be directed to notify the client (through the client's callback function) when data transfers complete. Additionally, the client's callback function is invoked when an unexpected event, such as a DMA error, 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.

DMA Manager Initialization

In order to use the DMA manager, the client must first initialize it. The DMA manager does not use static data, so the initialization step is used to give the DMA manager memory for use in managing the DMA controller.

The DMA manager requires a small, fixed amount of base memory and a variable amount of memory, depending the number of 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) required for the base and channel memory. These macros are `ADI_DMA_BASE_MEMORY` and `ADI_DMA_CHANNEL_MEMORY`.

For example, if the client wants to initialize the DMA manager and has at most four DMA channels and one memory DMA stream open simultaneously, the amount of required memory 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 to the DMA manager. The channel count can be tested to ensure that 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 which 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 is used to manage the open channel(s). After the DMA manager is initialized, DMA channels and memory streams can be opened for use.

Although it is possible to create multiple DMA managers in a single-core Blackfin system, there is no practical advantage in doing so.

DMA Manager 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.

ⓘ In many embedded systems, the DMA manager is never terminated.

Memory DMA and Peripheral DMA

As described in the Blackfin processor's *Hardware Reference*, the Blackfin processor's DMA controller supports both peripheral DMA and memory DMA. Regardless of whether peripheral DMA or memory DMA is being used, the client schedules DMA manager activity on a block-by-block basis, rather than a sample-by-sample basis. Though 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 to be performed. The term *buffer* is used throughout this document to represent the block of data.

Peripheral DMA moves 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 of) 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 processor's memory spaces. For example, due to the large 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 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 accomplished most easily using higher level memory streams. The `adi_dma_MemoryXXXX()` functions provide a simple, efficient method of transferring data between the various memory spaces by the Blackfin processor's DMA controller.

The overall sequence for using memory streams is to open the memory stream, schedule transfers as needed, and 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 controls the stream
- The stream ID (of type `ADI_DMA_STREAM_ID`) that identifies the 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 makes *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 one-dimensional and two-dimensional transfers, so a client can schedule a one-dimensional transfer on a given stream, and can then schedule a two-dimensional transfer on that same stream.

Note that a memory stream supports only one transfer at a time. If one 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 provided to the client during the `adi_dma_MemoryOpen()` function.
- The destination starting address into which data is copied
- The source starting address from which data is copied
- The width of each element (in bytes) to be copied. The DMA manager uses this value to schedule 8-, 16-, or 32-bit transfers.

- A count of the number of elements copied
- The address of the callback function to be called when the transfer is complete. The invocation of the callback function 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 (matrix) memory transfers are handled by calling the `adi_dma_MemoryCopy2D()` function. When calling this function, the client provides the following parameters:

- The stream handle. This is the value 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 is stored into the destination memory
- A pointer to a data structure (of type `ADI_DMA_2D_TRANSFER`) that defines how data is read from the source memory
- The width of each element (in bytes) 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. The invocation of the callback function depends on the callback service handle value 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.

The `ADI_DMA_2D_TRANSFER` data type structure holds the necessary values to specify a two-dimensional transfer. This data type contains the starting address in memory, an `XCount` value that defines the number of columns, a `YCount` value that defines the number of rows, and `XModify` and `YModify` values to describe 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 provided to the client during the `adi_dma_MemoryOpen` function.
- A flag to indicate whether the DMA manager should wait for the completion of any ongoing transfers on the stream 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 be opened first and then configured.

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 controls the channel
- The channel ID (of type `ADI_DMA_CHANNEL_ID`) that identifies the DMA channel to open
- A client handle that is passed back to the client's callback function. This is a client-supplied value, providing some meaning to the client, which is passed back to the client's callback function so the client can associate this value with the stream 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 moves data. Refer to the sections starting with "Single Transfers" on page 6-12.
- 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 makes 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 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 configured, buffers can be supplied to the channel, and so on. Note that the actual transfer of data does not begin with the `adi_dma_MemoryOpen` function. Dataflow must be enabled specifically via the `adi_dma_Control` function.

The DMA manager supports the following operational modes of the Blackfin processor's DMA controller:

- "Single Transfers"
- "Circular Transfers" on page 6-14
- "Large Descriptor Chaining Model" on page 6-16
- "Small Descriptor Chaining Model" on page 6-20

Single Transfers

The single transfer operating mode (`ADI_DMA_MODE_SINGLE`) transfers 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 the following parameters to the function:

- The channel handle. This is the value 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 is initially read (for outbound data) or the address in memory where data is initially stored (when the transfer is for inbound data).
- The configuration word for the transfer. This is a 16-bit value that represents the DMA configuration control 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 fields 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 call-back to the client when the transfer completes.

- The `XCount` value. For one-dimensional transfers, this value defines the number of elements to transfer. 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 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 auto-buffer 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, repeating indefinitely. Optionally, the client can 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 circular transfer mode, the client calls the `adi_dma_Buffer()` function with the following parameters:

- The channel handle. This is the value 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 is initially read from (when the transfer is for outbound data), or the address in memory where data is initially stored (when the transfer is for inbound data).
- The configuration word for the transfer. This 16-bit value 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 client provides values for the following fields within the configuration word.

`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. Set this parameter 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. Set this parameter 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. After enabling dataflow, if the client wants to change to a different circular buffer, the client must first disable dataflow on the channel, call the `adi_dma_Buffer()` function with the new buffer data, and then re-enable dataflow on the appropriate channel.

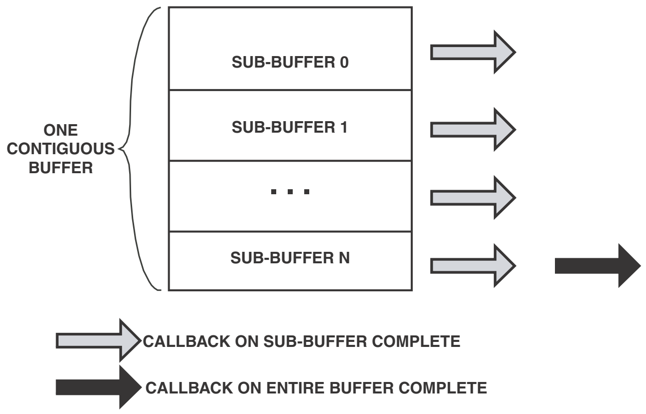
ONE CONTIGUOUS BUFFER
SUB-BUFFER 0
SUB-BUFFER 1
. . .
SUB-BUFFER N
CALLBACK ON SUB-BUFFER COMPLETE
CALLBACK ON ENTIRE BUFFER COMPLETE

Figure 6-1. Circular Buffer Usage in a Circular Transfer

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.

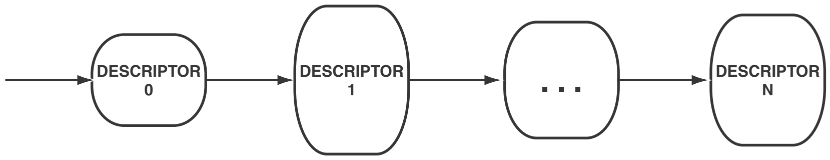

Figure 6-2. Descriptor Chain

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 are processed.

Both one-dimensional 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 call back.

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 provided to the client during the `adi_dma_Open()` function.
- A handle of the type `ADI_DMA_DESCRIPTOR_HANDLE` to a descriptor. Because 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`, which is a data type that defines a large model descriptor. When calling the `adi_dma_Queue()` function, the client can 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 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, 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 is initially read from (when the transfer is for outbound data), or the address in memory where data is initially stored (when the transfer is for inbound data).
- The configuration word for the transfer. This 16-bit value 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 client provides the following values to fields within the configuration word.

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 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.

The DMA manager does not constrain when descriptors can be provided to a channel. For DMA channels that process 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.

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 use 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 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 6-6 on page 6-74. In most cases, the client passes the following parameters to the `adi_dma_Control()` function:

- The channel handle. This is the value 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 configured.
- A command-specific value. The 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 provided to the client during the `adi_dma_Open()` function.
- A flag indicating whether 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, and 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 whether a transfer is in progress by using the `ADI_DMA_CMD_GET_TRANSFER_STATUS` command.

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 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 passes 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 that clients use to determine when a transfer has 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 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. 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 provided in the `adi_dma_Open()` function.
- Event ID. This value is `ADI_DMA_EVENT_INNER_LOOP_PROCESSED` when a sub-buffer has completed processing or `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 a callback is 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 provided in the `adi_dma_Open()` function.
- Event ID. This value is `ADI_DMA_EVENT_DESCRIPTOR_PROCESSED`.
- Starting address of the data

Descriptor-Based Sub-Modes

When using the small or large model descriptor-based transfers, two sub-modes (loopback and streaming) allow the client application greater flexibility in processing descriptors. Each of these sub-modes can be used independently or in combination. Each sub-mode is enabled or disabled via the `adi_dma_Control()` function. Clients that want to use these sub-modes must enable them prior to enabling dataflow on the channel. By default, both sub-modes are disabled.

Loopback Sub-Mode

The loopback sub-mode is controlled by the `ADI_DMA_CMD_SET_LOOPBACK` command.

When loopback sub-mode 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 provided to the channel. This effectively creates an infinite loop of descriptors as illustrated in Figure 6-3. For example, with loopback sub-mode, the client can provide the descriptors at initialization time, allow the DMA manager to process the descriptors, and never need 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.

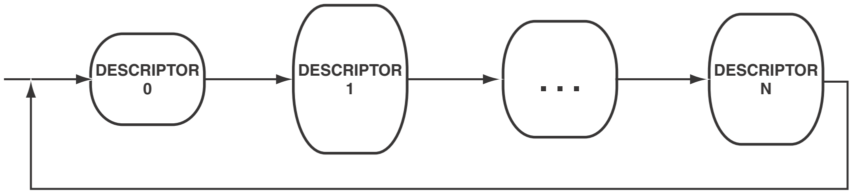

Figure 6-3. Descriptor Chain With Loopback

Streaming Sub-Mode

The streaming sub-mode is controlled by the `ADI_DMA_CMD_SET_STREAMING` command.

When not using streaming sub-mode, the DMA manager pauses the DMA controller after processing a descriptor that has been tagged to generate a callback has been processed. The DMA manager does this because the Blackfin processor's 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 sub-mode allows the client to alter this behavior. When the streaming sub-mode is enabled, the DMA manager never pauses the DMA controller; this allows 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 met fairly easily 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. Typically, however, the mappings for the memory DMA channels are fixed and cannot be changed.

The DMA manager provides two functions, `adi_dma_GetMapping()` and `adi_dma_SetMapping()`, 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 is 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 detected.
- Pointer to an `ADI_DMA_CHANNEL_ID` value. This value is the address of a location in memory into which the function stores 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 set.
- The channel ID. This value, an `ADI_DMA_CHANNEL_ID` value, enumerates the DMA channel to which the given peripheral is 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, and 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 client can alter the mapping of DMA channels to IVG levels 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, 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 that 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 no other open channels 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. This is especially useful in video 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 you 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) for processing. 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 process.

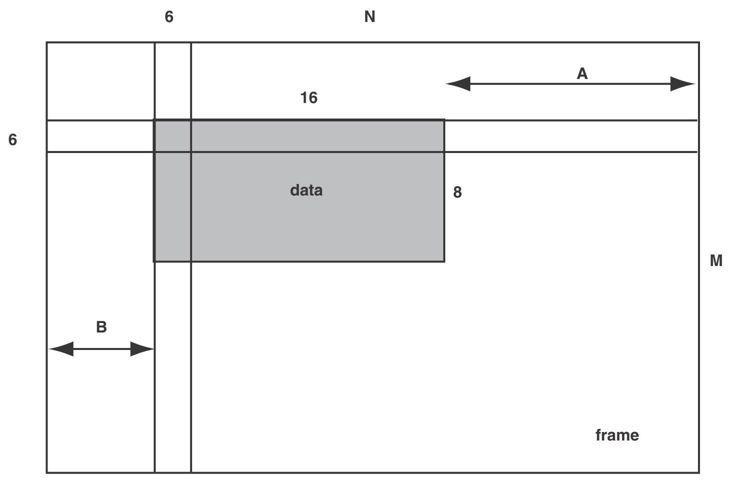

Figure 6-4. Selecting a 16 x 8 Block of Data From a Video Frame of Size N x M

To select each row of the 16 x 8 block, the inner loop of the required 2-D 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 is also 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.

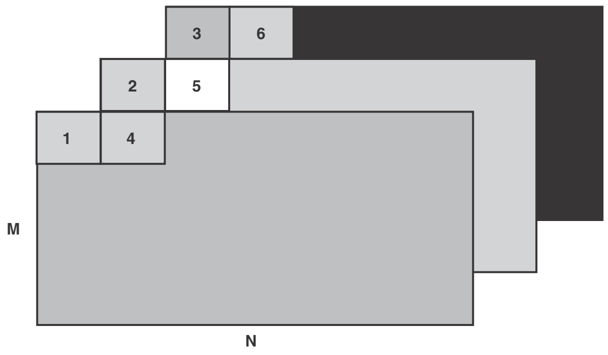

Figure 6-5. Capturing a Video Data Stream of (R,G,B Pixels) x (N x M Image Size)

In this case, the inner loop of the required 2-D DMA configuration has three values (XCOUNT=3) and a stride (XMODIFY) of N*M, chosen such that successive elements in each row (or RGB tuple) are 1 - 2 - 3, 4 - 5 - 6, and so on (see Figure 6-5).

The outer loop of the 2-D 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 Traffic Control

The traffic control period registers and the traffic control count registers can be controlled using a command to set a value and a command to sense a value.

A data structure called `ADI_DMA_TC_SET` is defined for setting a DMA traffic control parameter. It contains fields to specify which DMA controller the command is being issued for, which of the traffic control parameters to set (`DEB`, `DCB`, `DAB`), and the value to set it to.

A similar data structure called `ADI_DMA_TC_GET` is defined for sensing a DMA traffic control parameter.

Two commands called `ADI_DMA_CMD_GET_TC` and `ADI_DMA_CMD_SET_TC` are used to set and sense the traffic control parameters. For more details on setting and sensing traffic control parameters, see "Data Structures" on page 6-64 and Table 6-6 on page 6-74.

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 – 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 functions listed in Table 6-1.

Table 6-1. DMA Manager API Functions

Function	Description
Primary Functions	
`adi_dma_Buffer`	Provides a single or circular buffer. See "adi_dma_Buffer" on page 6-34.
`adi_dma_Close`	Closes a DMA channel. See "adi_dma_Close" on page 6-36.
`adi_dma_Control`	Controls/queries the operation of a DMA channel. See "adi_dma_Control" on page 6-37.
`adi_dma_Init`	Initializes a DMA manager. See "adi_dma_Init" on page 6-42.
`adi_dma_Open`	Opens a DMA channel for use. See "adi_dma_Open" on page 6-56.
`adi_dma_Queue`	Queues a descriptor chain. See "adi_dma_Queue" on page 6-58.
`adi_dma_Terminate`	Shuts down and terminates a DMA manager. See "adi_dma_Terminate" on page 6-61.
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_GetPeripheralInterruptID`	Gets the peripheral interrupt ID for a given DMA channel ID. See "adi_dma_GetPeripheralInterruptID" on page 6-41.
`adi_dma_SetConfigWord`	Sets the bits in the configuration word for a chain of descriptors. See "adi_dma_SetConfigWord" on page 6-59.
`adi_dma_SetMapping`	Sets the DMA Channel ID to which a peripheral is mapped. See "adi_dma_SetMapping" on page 6-60.

Table 6-1. DMA Manager API Functions (Cont'd)

Function	Description
Memory DMA Functions	
`adi_dma_MemoryOpen`	Opens a memory DMA stream for use. See "adi_dma_MemoryOpen" on page 6-48.
`adi_dma_MemoryClose`	Closes a memory DMA stream. See "adi_dma_MemoryClose" on page 6-43.
`adi_dma_MemoryCopy`	Copies memory in a linear, one-dimensional fashion. See "adi_dma_MemoryCopy" on page 6-44.
`adi_dma_MemoryCopy2D`	Copies memory in a two-dimensional fashion. See "adi_dma_MemoryCopy2D" on page 6-46.
Memory DMA Queue Functions	
`adi_dma_MemoryQueueControl`	Controls or configures a memory DMA stream. See "adi_dma_MemoryQueueControl" on page 6-53.
`adi_dma_MemoryQueueOpen`	Opens a memory DMA stream for queueing. See "adi_dma_MemoryQueueOpen" on page 6-54.
`adi_dma_MemoryQueueClose`	Closes a memory DMA stream that was opened for queueing. See "adi_dma_MemoryQueueClose" on page 6-52.
`adi_dma_MemoryQueue`	Queues memory DMA descriptor(s) to a stream. See "adi_dma_MemoryQueue" on page 6-50.

adi_dma_Buffer

Description

The `adi_dma_Buffer()` 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`	Uniquely identifies the DMA channel the buffer is assigned to and is the value returned when the DMA channel is opened
`StartAddress`	Location of the start of the filled or transmitted buffer
`Config`	DMA configuration control register for the transfer
`XCount`	Total number of words transferred in a one-dimensional buffer or the number of data elements per row in a two-dimensional buffer
`XModify`	Offset in bytes between each word transferred (1-D) or the offset in bytes between each row element (2-D)
`YCount`	Number of rows transferred
`YModify`	Offset in bytes between the last data element of one row and the first element of the next

Return Value

`ADI_DMA_RESULT_SUCCESS`	Buffer was assigned successfully.
`ADI_DMA_RESULT_BAD_HANDLE`	`ChannelHandle` does not contain a valid channel handle.
`ADI_DMA_RESULT_BAD_MODE`	DMA channel has not been opened for either single or circular buffer operation.
`ADI_DMA_RESULT_ALREADY_RUNNING`	DMA operation is in progress.

adi_dma_Close

Description

The `adi_dma_Close()` function closes a DMA channel and releases the configuration memory for further use. Depending on the value of the `WaitFlag` argument, the channel is closed immediately or is closed after ongoing transfers have completed.

Prototype

```
ADI_DMA_RESULT adi_dma_Close(
      ADI_DMA_CHANNEL_HANDLE      ChannelHandle,
      u32                         WaitFlag
);
```

Arguments

`ChannelHandle`	Uniquely identifies the DMA channel to close and is the value returned when the DMA channel is opened
`WaitFlag`	If set to `TRUE(1)`, instructs the DMA manager to wait for ongoing transfers to complete before closing the channel; otherwise, if set to `FALSE(0)`, the channel is 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`	Data handler and/or error handler cannot be unhooked.

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.

```
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(ChannelHan-
dle,ADI_DMA_CMD_VALUE_PAIR,&cmd);
```

- A table of `ADI_COMMAND_PAIR` structures is passed. The table must have the 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 6-74.

Prototype

```
ADI_DMA_RESULT adi_dma_Control(
        ADI_DMA_CHANNEL_HANDLE     ChannelHandle,
        ADI_DMA_CM                 Command,
        void                       *Value
);
```

Arguments

ChannelHandle	Uniquely identifies the DMA channel the buffer is assigned to and is the value returned when the DMA channel is opened.
Command	ADI_DMA_CMD enumeration value. See "DMA Commands" on page 6-74 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`	Function completed successfully.
`ADI_DMA_RESULT_BAD_COMMAND`	Command is invalid. Either a bad command or a specific command is not allowed in this context.
`ADI_DMA_RESULT_ALREADY_RUNNING`	Commands could not be performed as the channel is currently transferring data.

adi_dma_GetMapping

Description

The `adi_dma_GetMapping()` 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           pmap,
        ADI_DMA_CHANNEL_ID     *pChannelID
);
```

Arguments

`pmap`	Peripheral ID is queried.
`pChannelID`	Location where the DMA manager stores the channel ID to which the peripheral is mapped.

Return Value

`ADI_DMA_RESULT_SUCCESS`	Device is identified and DMA information is returned.
`ADI_DMA_RESULT_BAD_PERIPHERAL`	Bad peripheral value was encountered.
`ADI_DMA_RESULT_NOT_MAPPED`	No mapping was found for the device.

adi_dma_GetPeripheralInterruptID

Description

The `adi_dma_GetPeripheralInterruptID` function gets the peripheral interrupt ID for a given DMA channel ID.

Prototype

```
ADI_DMA_RESULT adi_dma_GetPeripheralInterruptID(
        ADI_DMA_CHANNEL_ID       ChannelID,
        ADI_INT_PERIPHERAL_ID    *pPeripheralID
);
```

Arguments

ChannelID	DMA channel ID
`pPeripheralID`	`ADI_INT_PERIPHERAL_ID` structure in which the peripheral ID will be stored.

Return Value

`ADI_DMA_RESULT_SUCCESS`	No errors encountered
`ADI_DMA_RESULT_BAD_CHANNEL_ID`	Invalid channel ID

adi_dma_Init

Description

The `adi_dma_Init()` function initializes a DMA manager.

Prototype

```
ADI_DMA_RESULT adi_dma_Init(
        void                        *pMemory,
        const size_t                MemorySize,
        u32                         *pMaxChannels
        ADI_DMA_MANAGER_HANDLE      *pManagerHandle,
        void                        *pCriticalRegionArg
);
```

Arguments

`pMemory`	Pointer to memory that the DMA can use
`MemorySize`	Size, in bytes, of the memory provided
`pMaxChannels`	Location in memory where the DMA manager stores the number of simultaneously open channels that can be supported given the memory provided
`pManagerHandle`	Location in memory where the DMA manager stores the handle to the DMA manager
`pCriticalRegionArg`	Parameter that the DMA manager passes to the `adi_int_EnterCriticalRegion()` function

Return Value

`ADI_DMA_RESULT_SUCCESS`	Function completed successfully.
`ADI_DMA_RESULT_NOMEMORY`	Insufficient memory is available to initialize the DMA manager.

adi_dma_MemoryClose

Description

The `adi_dma_MemoryClose()` function closes down a memory DMA stream, freeing up all resources used by the memory stream.

Prototype

```
ADI_DMA_RESULT adi_dma_MemoryClose(
    ADI_DMA_STREAM_HANDLE        StreamHandle,
    u32                          WaitFlag
);
```

Arguments

`StreamHandle`	Handle to the DMA memory stream
`WaitFlag`	If set to `TRUE(1)`, 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

`ADI_DMA_RESULT_SUCCESS`	Function completed successfully.
`ADI_DMA_RESULT_BAD_HANDLE`	`StreamHandle` parameter does not point to a valid memory stream.

adi_dma_MemoryCopy

Description

The `adi_dma_MemoryCopy()` 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                     ElementWidth,
    u16                     ElementCount,
    ADI_DCB_CALLBACK_FN     ClientCallback
);
```

Arguments

`StreamHandle`	Handle to the DMA memory stream
`pDest`	Starting address into which memory is copied
`pSrc`	Starting address from which memory is copied
`ElementCount`	Number of elements to transfer
`ElementWidth`	Width of an element (in bytes); allowed values are 1, 2, and 4.
`ClientCallback`	Callback function 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

`ADI_DMA_RESULT_SUCCESS`	Function completed successfully.
`ADI_DMA_RESULT_BAD_HANDLE`	`StreamHandle` parameter does not point to a valid memory stream.
`ADI_DMA_RESULT_IN_USE`	Memory stream already has a transfer in progress.

adi_dma_MemoryCopy2D

Description

The `adi_dma_MemoryCopy2D()` 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`	Handle to the DMA memory stream
`pDest`	Pointer to the structure that describes how and where the data is copied into memory
`pSrc`	Pointer to the structure that describes how and where the data is copied from memory
`ElementWidth`	Width of an element (in bytes); allowed values are 1, 2, and 4.
`ClientCallback`	Callback function 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

`ADI_DMA_RESULT_SUCCESS`	Function completed successfully.
`ADI_DMA_RESULT_BAD_HANDLE`	`StreamHandle` parameter does not point to a valid memory stream.
`ADI_DMA_RESULT_IN_USE`	Memory stream already has a transfer in progress.

adi_dma_MemoryOpen

Description

The `adi_dma_MemoryOpen()` 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,
    void                     *DCBHandle
);
```

Arguments

`ManagerHandle`	Handle to the DMA manager
`StreamID`	Memory stream ID that is opened.
`ClientHandle`	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`	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 memory stream includes this handle.
`DCBServiceHandle`	Handle to the deferred callback service 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

`ADI_DMA_RESULT_SUCCESS`	Function completed successfully.
`ADI_DMA_RESULT_ALL_IN_USE`	All channel memory is in use.
`ADI_DMA_RESULT_CANT_HOOK_INTERRUPT`	System cannot hook a DMA data or error interrupt.

adi_dma_MemoryQueue

Description

The `adi_dma_MemoryQueue()` function queues memory DMA descriptors(s) to a stream.

Prototype

```
ADI_DMA_RESULT adi_dma_MemoryQueue(
    ADI_DMA_STREAM_HANDLE        StreamHandle,
    ADI_DMA_DESCRIPTOR_LARGE     *pSourceDescriptor,
    ADI_DMA_DESCRIPTOR_LARGE     *pDestinationDescriptor
);
```

Arguments

`StreamHandle`	Handle to the DMA memory stream
`pSourceDescriptor`	Source descriptor handle
`pDestinationDescriptor`	Destination descriptor handle

Return Value

`ADI_DMA_RESULT_SUCCESS`	Function completed successfully.
`ADI_DMA_RESULT_BAD_HANDLE`	Invalid stream handle was passed.
`ADI_DMA_RESULT_BAD_DESCRIPTOR`	Invalid descriptor was passed.
`ADI_DMA_RESULT_ALIGNMENT_ERROR`	Parameters will cause an alignment error.
`ADI_DMA_RESULT_BAD_XCOUNT`	Invalid `XCount` value was supplied.
`ADI_DMA_RESULT_NULL_DESCRIPTOR`	A NULL descriptor was passed.
`ADI_DMA_RESULT_INCOMPATIBLE_TRANSFER_SIZE`	Source and destination have different transfer sizes.
`ADI_DMA_RESULT_INCOMPATIBLE_WDSIZE`	Source and destination have different `WDSIZE` values.
`ADI_DMA_RESULT_INCOMPATIBLE_CALLBACK`	Destination descriptor callback is not compatible with source descriptors.
`ADI_DMA_RESULT_NO_BUFFER`	Channel has no buffer.

adi_dma_MemoryQueueClose

Description

The `adi_dma_MemoryQueueClose()` function closes a memory DMA stream that was opened for queueing.

Prototype

```
ADI_DMA_RESULT adi_dma_MemoryQueueClose(
    ADI_DMA_STREAM_HANDLE     StreamHandle,
    u32                       WaitFlag
);
```

Arguments

`StreamHandle`	Handle to the DMA memory stream
`WaitFlag`	Wait for transfers to complete flag (TRUE/FALSE)

Return Value

`ADI_DMA_RESULT_SUCCESS`	Function completed successfully.
`ADI_DMA_RESULT_BAD_HANDLE`	Invalid stream handle was passed.

adi_dma_MemoryQueueControl

Description

The `adi_dma_MemoryQueueControl()` function controls or configures a memory DMA stream.

Prototype

```
ADI_DMA_RESULT adi_dma_MemoryQueueControl(
    ADI_DMA_STREAM_HANDLE   StreamHandle,
    ADI_DMA_CMD             Command,
    void                    *Value
);
```

Arguments

`StreamHandle`	Handle to the DMA memory stream
`Command`	Command ID
`Value`	Command-specific value

Return Value

`ADI_DMA_RESULT_SUCCESS`	Function completed successfully.
`ADI_DMA_RESULT_BAD_COMMAND`	Invalid command item was passed.
`ADI_DMA_RESULT_ALREADY_RUNNING`	Commands could not be performed as the channel is currently transferring data.
`ADI_DMA_RESULT_BAD_HANDLE`	Invalid stream handle was passed.

adi_dma_MemoryQueueOpen

Description

The `adi_dma_MemoryQueueOpen()` function opens a memory DMA stream for queueing.

Prototype

```
ADI_DMA_RESULT adi_dma_MemoryQueueOpen(
    ADI_DMA_MANAGER_HANDLE    ManagerHandle,
    ADI_DMA_STREAM_ID         StreamID,
    void                      *ClientHandle
    ADI_DMA_STREAM_HANDLE     *pStreamHandle,
    void                      *DCBHandle,
    ADI_DCB_CALLBACK_FN       ClientCallback
);
```

Arguments

`ManagerHandle`	Handle to the DMA manager
`StreamID`	Open memory stream ID
`ClientHandle`	`ClientHandle` argument passed in callbacks
`pStreamHandle`	Location where DMA `StreamHandle` is stored
`DCBHandle`	Deferred callback service handle
`ClientCallback`	Client callback function

Return Value

`ADI_DMA_RESULT_SUCCESS`	Function completed successfully.
`ADI_DMA_RESULT_BAD_HANDLE`	Invalid stream handle or manager handle was passed.
`ADI_DMA_RESULT_ALL_IN_USE`	All channel memory is in use.
`ADI_DMA_RESULT_CANT_HOOK_INTERRUPT`	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_MANAGER_HANDLE     ManagerHandle
   ADI_DMA_CHANNEL_ID         ChannelID
   void                       *ClientHandle,
   ADI_DMA_CHANNEL_HANDLE     *pChannelHandle,
   ADI_DMA_MODE               Mode,
   ADI_DCB_HANDLE             DCBHandle,
   ADI_DCB_CALLBACK_FN        ClientCallback
);
```

Arguments

Argument	Explanation
ManagerHandle	Handle to the DMA manager
ChannelID	ADI_DMA_CHANNEL_ID enumeration value. See "ADI_DMA_CHANNEL_ID" on page 6-67.
ClientHandle	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	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	ADI_DMA_MODE enumeration value specifying the data transfer mode used by the opened DMA channel. See "ADI_DMA_MODE" on page 6-68.
DCBServiceHandle	Handle to the deferred callback service 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	Address of a callback 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

ADI_DMA_RESULT_SUCCESS	Function completed successfully.
ADI_DMA_RESULT_ALL_IN_USE	All channel memory is in use.
ADI_DMA_RESULT_CANT_HOOK_INTERRUPT	System cannot hook a DMA data or error interrupt.

adi_dma_Queue

Description

The `adi_dma_Queue()` 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`	Uniquely identifies the DMA channel that the descriptor is queued on and is the value returned when the DMA channel is opened
`DescriptorHandle`	Pointer to the first descriptor in the chain

Return Value

`ADI_DMA_RESULT_SUCCESS`	Descriptor was queued successfully.
`ADI_DMA_RESULT_BAD_HANDLE`	`ChannelHandle` does not contain a valid channel handle.
`ADI_DMA_RESULT_BAD_DESCRIPTOR`	Descriptor handle is NULL.
`ADI_DMA_RESULT_ALREADY_RUNNING`	Cannot submit additional descriptors to a channel configured for a loopback with dataflow enabled.

adi_dma_SetConfigWord

Description

The `adi_dma_SetConfigWord()` function sets the bits in the configuration word for a chain of descriptors.

Prototype

```
ADI_DMA_RESULT adi_dma_SetConfigWord(
        ADI_DMA_CHANNEL_HANDLE        ChannelHandle,
        ADI_DMA_DESCRIPTOR_HANDLE     DescriptorHandle
);
```

Arguments

`ChannelHandle`	Channel handle
`DescriptorHandle`	Descriptor chain

Return Value

`ADI_DMA_RESULT_SUCCESS`	No errors encountered
`ADI_DMA_RESULT_BAD_HANDLE`	Channel handle is NULL.
`ADI_DMA_RESULT_BAD_DESCRIPTOR`	Descriptor chain is NULL.
`ADI_DMA_RESULT_NON_TERMINATED_CHAIN`	Chain is not NULL terminated.
`ADI_DMA_RESULT_BAD_DIRECTION`	The `WNR` bit is wrong.
`ADI_DMA_RESULT_CALLBACKS_DISALLOWED_ON_SOURCE`	No callbacks allowed

adi_dma_SetMapping

Description

The `adi_dma_SetMapping()` function maps the DMA channel ID to the given peripheral.

Prototype

```
ADI_DMA_RESULT adi_dma_SetMapping(
        ADI_DMA_PMAP            pmap,
        ADI_DMA_CHANNEL_ID      ChannelID
);
```

Arguments

`pmap`	Peripheral ID to which the DMA channel is mapped.
`ChannelID`	Channel ID that is mapped to the peripheral.

Return Value

`ADI_DMA_RESULT_SUCCESS`	Channel was successfully mapped.
`ADI_DMA_RESULT_BAD_PERIPHERAL`	Bad peripheral value was encountered.
`ADI_DMA_RESULT_ALREADY_RUNNING`	Mapping could not be performed as the channel is currently transferring data.

adi_dma_Terminate

Description

The `adi_dma_Terminate()` 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`	Handle to the DMA manager

Return Value

`ADI_DMA_RESULT_SUCCESS`	Process completed successfully.

Public Data Structures, Enumerations, and Macros

This section defines 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 `adi_dma.h` header file. All types have the `ADI_DMA_` prefix to avoid ambiguity with client developer's 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` data 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 and 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 a) a small descriptor chain, b) a large descriptor chain, or c) an array of descriptors. By using the handle/union, a single `adi_dma_Queue()` function is needed, instead of 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` data 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 operates. 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 control register are available in the public `adi_dma.h` header file. The field names follow the convention used in the *Hardware Reference* for the appropriate processor.

ADI_DMA_2D_TRANSFER

The `ADI_DMA_2D_TRANSFER` data structure defines the characteristics of 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;
    u16                           CallbackFlag;
} ADI_DMA_DESCRIPTOR_ARRAY;
```

Descriptor element `CallbackFlag` is defined as `u16`, but it should take only values 0 or 1 (`FALSE` or `TRUE`, respectively). Passing a value greater than 1 causes unpredictable results.

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;
    u16                                CallbackFlag;
} ADI_DMA_DESCRIPTOR_LARGE;
```

Descriptor element `CallbackFlag` is defined as `u16`, but it should take only values 0 or 1 (`FALSE` or `TRUE`, respectively). Passing a value greater than 1 causes unpredictable results.

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;
    u16                         CallbackFlag;
} ADI_DMA_DESCRIPTOR_SMALL;
```

Descriptor element `CallbackFlag` is defined as `u16`, but it should take only values 0 or 1 (`FALSE` or `TRUE`, respectively). Passing a value greater than 1 causes unpredictable results.

ADI_DMA_TC_SET

The `ADI_DMA_TC_SET` structure is used for setting a DMA traffic control parameter. The `ParameterID` field specifies what type of parameter to set and is defined using the `ADI_DMA_TC_PARAMETER` enumeration. The `ControllerID` specifies which DMA controller to set, where multiple controllers are available, the first controller starting at 0. The `Value` field specifies the value to write.

```
typedef struct ADI_DMA_TC_SET  {
    ADI_DMA_TC_PARAMETER     ParameterID;
    u16                      ControllerID;
    u16                      Value;
} ADI_DMA_TC_SET;
```

ADI_DMA_TC_GET

The ADI_DMA_TC_GET structure is used for sensing or getting a DMA traffic control parameter. The ParameterID field specifies what type of parameter to sense and is defined using the ADI_DMA_TC_PARAMETER enumeration. The ControllerID specifies which DMA controller to sense, where multiple controllers are available, the first controller starting at 0. The Value field specifies the location where the parameter value is stored.

```
typedef struct ADI_DMA_TC_GET  {
    ADI_DMA_TC_PARAMETER     ParameterID;
    u16                      ControllerID;
    u16                      *pValue;
} ADI_DMA_TC_GET;
```

General Enumerations

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 every DMA channel of the processor. This value is used in the adi_dma_Open() function to identify the channel to open. The specific enumeration values depend 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 (Table 6-2). Associated with the ADI_DMA_EVENT parameter is another parameter that points to the companion argument, pArg, for the event.

Table 6-2. ADI_DMA_EVENT

Value	Event	Companion Argument
`ADI_DMA_EVENT_DESCRIPTOR_PROCESSED`	Descriptor has completed processing or a memory stream has completed a memory copy operation.	The address of the descriptor just processed, or NULL when 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. This enumeration takes the values shown in Table 6-3.

Table 6-3. ADI_DMA_MODE

`ADI_DMA_DATA_MODE_UNDEFINED`	Undefined
`ADI_DMA_DATA_MODE_SINGLE`	Single one-shot buffer
`ADI_DMA_DATA_MODE_CIRCULAR`	Single circular buffer
`ADI_DMA_DATA_MODE_DESCRIPTOR_ARRAY`	Array of descriptors
`ADI_DMA_DATA_MODE_DESCRIPTOR_SMALL`	Chain of small descriptors
`ADI_DMA_DATA_MODE_DESCRIPTOR_LARGE`	Chain of large descriptors

ADI_DMA_PMAP

The `ADI_DMA_PMAP` enumeration defines each of the processor's DMA-supported on-chip peripherals. 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 shown in Table 6-4.

Table 6-4. ADI_DMA_RESULT

`ADI_DMA_RESULT_SUCCESS`	Generic success
`ADI_DMA_RESULT_FAILED`	Generic failure
`ADI_DMA_RESULT_NOT_SUPPORTED`	Function not supported
`ADI_DMA_RESULT_IN_USE`	Resource is already in use.
`ADI_DMA_RESULT_ALREADY_RUNNING`	DMA is already running.
`ADI_DMA_RESULT_NOT_MAPPED`	Peripheral is not mapped to a channel.
`ADI_DMA_RESULT_BAD_HANDLE`	Invalid channel handle
`ADI_DMA_RESULT_BAD_DESCRIPTOR`	Invalid descriptor
`ADI_DMA_RESULT_BAD_MODE`	Invalid channel mode
`ADI_DMA_RESULT_BAD_CHANNEL_ID`	No such channel ID
`ADI_DMA_RESULT_BAD_MEMORY_STREAM_ID`	No such memory stream ID
`ADI_DMA_RESULT_BAD_PERIPHERAL`	Invalid peripheral value
`ADI_DMA_RESULT_NO_BUFFER`	Channel has no buffer
`ADI_DMA_RESULT_ALL_IN_USE`	No free channel memory structures are available.

Table 6-4. ADI_DMA_RESULT (Cont'd)

ADI_DMA_RESULT_BAD_COMMAND	Invalid command item
ADI_DMA_RESULT_BAD_DATA_SIZE	Memory DMA source and destination conflict
ADI_DMA_RESULT_BAD_DATA_WIDTH	Data element width is not valid.
ADI_DMA_RESULT_NO_MEMORY	Cannot allocate memory to channel object
ADI_DMA_RESULT_CANT_HOOK_INTERRUPT	Cannot hook an interrupt
ADI_DMA_RESULT_CANT_UNHOOK_INTERRUPT	Cannot unhook an interrupt
ADI_DMA_RESULT_BAD_SEQUENCE	Invalid programming sequence
ADI_DMA_RESULT_BAD_CONFIG_REG	Invalid configuration register value
ADI_DMA_RESULT_BAD_ALIGNMENT_ERROR	Parameters will cause an alignment error.
ADI_DMA_RESULT_BAD_XCOUNT	Invalid XCount value was supplied.
ADI_DMA_RESULT_NON_TERMINATED_CHAIN	Descriptor chain is not NULL terminated.
ADI_DMA_RESULT_NO_CALLBACK_FUNCTION_ SUPPLIED	No callback function was provided to the open function.
ADI_DMA_RESULT_BAD_CONTROLLER_ID	Invalid controller ID specified
ADI_DMA_RESULT_BAD_TC_PARAMETER	Invalid traffic control parameter
ADI_DMA_RESULT_BAD_DIRECTION	Invalid XCount value was supplied.
ADI_DMA_RESULT_INCOMPATIBLE_WDSIZE	Source and destination have different WDSIZE values.
ADI_DMA_RESULT_INCOMPATIBLE_TRANSFER_ SIZE	Source and destination have different transfer sizes.
ADI_DMA_RESULT_NULL_DESCRIPTOR	NULL descriptor passed
ADI_DMA_RESULT_CALLBACKS_DISALLOWED_ON_ SOURCE	Callbacks are not allowed on source memory DMA descriptors.
ADI_DMA_RESULT_INCOMPATIBLE_CALLBACK	Destination descriptor callback incompatible with source descriptors

Table 6-4. ADI_DMA_RESULT (Cont'd)

ADI_DMA_RESULT_BAD_CHANNEL_MEMORY_SIZE	ADI_DMA_CHANNEL_MEMORY macro is invalid (internal error).
ADI_DMA_RESULT_INCOMPATIBLE_IVG_LEVEL	Function cannot be called from the current IVG level.

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_MemoryOpen() function to identify which stream to open. The specific enumeration values are dependent on the specific processor being targeted.

ADI_DMA_TC_PARAMETER

The ADI_DMA_TC_PARAMETER enumeration defines the DMA traffic control parameters that can be used in the ParameterID field of an ADI_DMA_TC_SET or ADI_DMA_TC_GET data structure when passing the ADI_DMA_CMD_GET_TC and ADI_DMA_CMD_SET_TC commands to the DMA manager. Possible values are shown in Table 6-5.

Table 6-5. ADI_DMA_TC_PARAMETER

ADI_DMA_TC_DCB	DMA core bus
ADI_DMA_TC_DEB	DMA access bus
ADI_DMA_TC_DAB	DMA external bus
ADI_DMA_TC_MDMA	MDMA round robin

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`	2-D DMA operation

ADI_DMA_DI_EN

`ADI_DMA_DI_EN_DISABLE`	Disables callbacks on completion.
`ADI_DMA_DI_EN_ENABLE`	Enables 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`	Disables DMA transfer on the channel.
`ADI_DMA_ENABLE`	Enables 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 6-6 describes the commands and values that can be issued via this function.

Table 6-6. 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 loop-back.
`ADI_DMA_CMD_SET_STREAMING`	TRUE/FALSE	Enables/disables streaming.
`ADI_DMA_CMD_SET_DATAFLOW`	TRUE/FALSE	Enables/disables data-flow.
`ADI_DMA_CMD_FLUSH`	n/a	Flushes all buffers and descriptors on a chan-nel.
`ADI_DMA_CMD_GET_TRANSFER_STATUS`	`u32 *`	Provides the transfer status, TRUE - in progress, FALSE - not in progress.
`ADI_DMA_CMD_SET_TC`	`ADI_DMA_TC_SET *`	Sets a traffic control (period) parameter.
`ADI_DMA_CMD_GET_TC`	`ADI_DMA_TC_GET *`	Senses a traffic control (count) parameter.

7 PROGRAMMABLE FLAG SERVICE

The programmable flag service within the system services library provides the application with an easy-to-use interface into the programmable flag (sometimes called general-purpose I/O, or GPIO) subsystem of the Blackfin processor.

This chapter contains the following sections:

Introduction

Using the capabilities of other system services, the flag service allows the client control over the direction of flags, values placed on or sensed from a flag pin, and notification of the client upon flag pin changes, via live or deferred callbacks.

The use of the flag service is dependent on the use of both the interrupt manager and the deferred callback (DCB) manager for its full operation. If callbacks are not deferred, but rather are live, the DCB manager is not required. If callbacks are not required, neither the interrupt manager nor DCB manager is required.

In order to reduce the pin count of devices, flag pins are sometimes muxed onto the same pins as other peripherals. The flag service does not provide arbitration functionality to control pin muxing. It is the responsibility of the client program to ensure that peripherals and the flag service do not use the same pin simultaneously. For the ADSP-BF531/532/533 and ADSP-BF561 processors, this entails ensuring that the relevant peripheral control registers are correctly set. For ADSP-BF534/536/537 (and future) processors, the flag service automatically invokes the port control service to effect any pin multiplexer changes. No user intervention is required.

For ADSP-BF531/532/533 and ADSP-BF561 processor cores, this entails ensuring that the relevant peripheral control registers are correctly set. For ADSP-BF534/536/537 (and future) processor cores, the port control registers are required to be set accordingly. The latter can be managed via the port control service within the system services library. (Note that device drivers for ADSP-BF534/536/537 (and future) processors automatically make the appropriate calls into the port control service without any user intervention.)

The flag service uses an unambiguous naming convention to safeguard against conflicts with other software libraries provided by ADI or elsewhere. To this end, all enumeration values and `typedef` statements use the

ADI_FLAG_ prefix, and functions and global variables use the lowercase adi_flag_ equivalent.

Each function within the flag service application program interface (API) returns an error code of the type ADI_FLAG_RESULT. Like all system services, a return value of 0 (ADI_FLAG_RESULT_SUCCESS) indicates no errors. A nonzero value indicates an error. Like all system services, flag service error codes are unique from all other system services. The adi_flag.h include file lists all error codes that the flag service returns.

Parameter checking in the debug versions of the system services library provides a more complete test of API function parameters and for conditions that may cause errors. ADI strongly recommends that development work be done using the debug versions of the system service library, and final test and deployment be done with the release version of the library.

Operation

This section describes the overall operation of the flag service. Details on the application program interface (API) can be found later in this chapter.

Initialization

Prior to using the flag service, the client must first initialize the service. In order to initialize the service, the client passes to the initialization function, adi_flag_Init, a parameter that is passed to the critical region function should the flag service need to protect a critical region of code, and optionally a contiguous piece of memory that the service can use for flag callbacks.

The flag service provides a facility whereby, if so directed by the client, a callback function supplied by the client can be invoked should conditions on a flag cause an interrupt event. See "Callbacks" on page 7-6 for more information on how callbacks operate.

In order to control and manage callbacks, the flag service needs a small amount of memory to store the necessary information about each callback. The exact amount of memory is defined by a macro, `ADI_FLAG_CALLBACK_MEMORY`. The client provides an amount of memory equal to `ADI_FLAG_CALLBACK_MEMORY` times the number of callbacks that are simultaneously installed at any point in time.

For example, if the client has simultaneously installed callbacks for two flags, the client provides `ADI_FLAG_CALLBACK_MEMORY * 2` bytes of memory. If flag callbacks are not used, no memory need be provided by the client.

Termination

When the client no longer requires the functionality of the flag service, the termination function, `adi_flag_Terminate`, is called. This function uninstalls any flag callbacks and returns any memory provided to the flag initialization function back to the client.

Flag IDs

All API functions within the flag service, other than the initialize and terminate functions, are passed a parameter that identifies the controlled flag. This parameter is of the type `ADI_FLAG_ID`. The include file for the flag service, `adi_flag.h`, defines flag IDs for each flag supported by the processor. Flag IDs are of the form `ADI_FLAG_x`, where x uniquely identifies the specific flag.

Flag Control Functions

The functions described in this section control operation of each flag.

adi_flag_Open

The `adi_flag_Open()` function is called prior to any of the individual flag control functions. Depending on the specific Blackfin device, this function initializes any hardware necessary for the operation of the flag. For example, on ADSP-BF534/536/537 (and future) processors, this function configures the port control logic, via the port control system service, for the flag used as a general-purpose I/O pin. On ADSP-BF561 and ADSP-BF531/532/533 processors, this function does nothing and simply returns to the caller. Refer also to "adi_flag_Open" on page 7-19.

adi_flag_Close

When a flag is no longer needed, the `adi_flag_Close()` function is called to close and shut down the flag. At present, for all Blackfin processors, this function does nothing but returns immediately to the caller. Future Blackfin devices may require this function to manipulate the hardware in some way when closing a flag. Refer also to "adi_flag_Close" on page 7-16.

adi_flag_SetDirection

Flags can be configured as inputs or outputs. The `adi_flag_SetDirection()` function is used to set the direction of a flag as either an input or an output. This function does not change the value of a flag. Refer also to "adi_flag_SetDirection" on page 7-24.

adi_flag_Set

When configured as an output, the `adi_flag_Set()` function sets the value of the flag to a logical 1, driving high. Refer also to "adi_flag_Set" on page 7-23.

adi_flag_Clear

When configured as an output, the `adi_flag_Clear()` function sets the value of the flag to a logical 0, driving low. Refer also to "adi_flag_Clear" on page 7-15.

adi_flag_Toggle

When configured as an output, the `adi_flag_Toggle()` function inverts the current value of the flag. If the flag was clear/low, this function changes the flag to set/high. If the flag was set/high, this function changes the flag to clear/low. Refer also to "adi_flag_Toggle" on page 7-21.

adi_flag_Sense

When configured as an input, the `adi_flag_Sense()` function senses the value of the flag and stores that value in the location provided by the client. If the flag is clear/low, then a value of `FALSE` is stored in the location. If the flag is set/high, then a value of `TRUE` is stored in the location. Refer also to "adi_flag_Sense" on page 7-20.

Callbacks

Like other system services, the flag service uses a callback mechanism in order to notify the client of, typically, asynchronous events.

The Blackfin processor's programmable flags can be configured to generate interrupts. The flag service provides an internal interrupt handler that is used to process interrupts from the flag hardware. This interrupt handler makes the appropriate callbacks into the client's application. When a client installs a flag callback, a parameter to the function dictates whether the callback should be made live or deferred. Live callbacks mean that the client's callback function is called at interrupt time. Deferred callbacks mean that callbacks are not made at interrupt time but rather deferred to a lower-priority using a specified deferred callback service.

When using the callback capability of the flag service, the client does not need to take any other action outside the flag service API. No calls to the interrupt manager or deferred callback service, other than initialization of those services, are required.

It is possible for clients to use all capabilities of the flag service and not use any of the callback capabilities. If callbacks are not used by the client, no memory need be provided to the flag service's initialization function.

adi_flag_InstallCallback

The `adi_flag_InstallCallback()` function is used to install a callback to a specified flag. In addition to the flag ID, the client provides the interrupt ID that the flag should generate, a wakeup flag, the type of trigger that generates the callback, the callback function address, a client handle, and deferred callback service handle. Refer also to "adi_flag_InstallCallback" on page 7-26.

Depending on the specific Blackfin processor, programmable flags can generate any one of several interrupts. (Sometimes the processor has constraints as to which flag can generate which interrupt. See the appropriate Blackfin processor *Hardware Reference* for details.) The peripheral ID enumerates which interrupt the flag generates.

The wakeup flag indicates whether the processor is woken up from a low power state should the flag event occur.

The trigger type describes the event that causes the callback to occur. The following trigger types (all enumerated in the `adi_flag.h` include file) are supported:

- Level high – callback generated when the level is high
- Level low – callback generated when the level is low
- Rising edge – callback generated on the rising edge

- Falling edge – callback generated on the falling edge
- Both edges – callback generated on both the rising and falling edge (**Note:** This trigger type is not supported by ADSP-BF54x hardware.)

The callback function address specifies a callback function of the type `ADI_DCB_CALLBACK_FN` (see the deferred callback service for more information on this data type). When invoked, the callback function is passed three parameters:

- `ClientHandle` – a value provided by the client when the callback was installed
- `ADI_FLAG_EVENT_CALLBACK` – indicates a flag callback event
- `FlagID` – the flag ID of the flag that generated the callback

When the deferred callback service handle parameter passed to the `adi_flag_InstallCallback` function is NULL, the callback is executed live, meaning it is invoked at interrupt time. If the deferred callback service handle parameter is non-NULL, the flag service uses the specified deferred callback service to invoke the callback.

A single callback function can be used and installed for any number of flags; the callback function can use the `FlagID` parameter to determine which flag generated the callback. Note, however, that only one callback should be installed for a given flag.

This function does not alter flag control, such as direction.

adi_flag_RemoveCallback

The `adi_flag_RemoveCallback()` function is used to remove a callback from a specified flag. This function disables interrupt generation for the flag and removes the callback from its internal tables. Unless reinstalled, no further callbacks occur for the specified flag. After calling this function, the memory freed by removing the callback is available for the flag service to use for the next callback that is installed. This function does not alter flag control, such as direction. Refer also to "adi_flag_RemoveCallback" on page 7-28.

adi_flag_SuspendCallbacks

The `adi_flag_SuspendCallbacks()` function is used to temporarily suspend callbacks for a given flag but does not uninstall the callback. This function is typically used in conjunction with the `adi_flag_ResumeCallbacks` function. Refer also to "adi_flag_SuspendCallbacks" on page 7-30.

adi_flag_ResumeCallbacks

The `adi_flag_ResumeCallbacks()` function is used to re-enable callbacks that were suspended by the `adi_flag_SuspendCallbacks` function. Refer also to "adi_flag_ResumeCallbacks" on page 7-29.

adi_flag_SetTrigger

The `adi_flag_SetTrigger()` function sets the condition on a flag that triggers a callback. This function is not typically called by clients as setting the trigger condition is taken care of automatically by the `adi_flag_InstallCallback` function. This function is provided as a convenience for users who want an extra measure of control on callbacks. Refer also to "adi_flag_SetTrigger" on page 7-25.

Coding Example

This section describes the code required to implement a simple example using the flag service. This example initializes the flag service, configures one flag for input, and configures another flag for output. The example illustrates how the output flag is controlled and then illustrates how a callback function is used to sense changes on a flag. All flag service functions return an error code. In practice, this error code should be checked to ensure the function completed successfully. For the purposes of this example only, the return value is not checked.

Initialization

Prior to using the flag service, it must be initialized. The following fragment initializes the service and provides the service with memory for one callback function.

```
static u8 FlagServiceData[ADI_FLAG_CALLBACK_MEMORY * 1];
                          // memory for service

ADI_FLAG_RESULT Result;   // return value
u32 ResponseCount;        // number of callbacks supported

Result = adi_flag_Init(FlagServiceData, sizeof(FlagServiceData),
                       &ResponseCount, NULL);
```

Upon completing this function, the flag service is initialized and is ready for use.

Opening a Flag

After the service is initialized, any flags used can be opened. In this example, two flags are used.

```
Result = adi_flag_Open(ADI_FLAG_PF0);

Result = adi_flag_Open(ADI_FLAG_PF1);
```

The `adi_flag_Open` function takes any action necessary to configure the processor hardware for use as a programmable flag.

Setting Flag Direction

After the flags are opened, they must be set to the proper direction. In this example, one flag is configured for input and one for output.

```
Result = adi_flag_SetDirection(ADI_FLAG_PF0,
                               ADI_FLAG_DIRECTION_INPUT);

Result = adi_flag_SetDirection(ADI_FLAG_PF1,
                               ADI_FLAG_DIRECTION_OUTPUT);
```

Once the flag direction has been established, the flag can be controlled.

Controlling an Output Flag

After a flag is configured for the output direction, its value can be set with any of the following functions.

```
Result = adi_flag_Set(ADI_FLAG_PF0);
                  // sets output value to logical high (1)

Result = adi_flag_Clear(ADI_FLAG_PF0);
                  // sets output value to logical low (0)

Result = adi_flag_Toggle(ADI_FLAG_PF0);
                  // toggles from current value
```

The first call sets the value of the flag to a logical high, and the second call sets the flag to a logical low. The third call toggles the current value of the flag—if logical low, it changes to a logical high value; if logical high, it changes to a logical low value.

Sensing the Value of a Flag

The application can sense the value of a flag, regardless of whether the flag has been configured as an input or an output. The following fragment illustrates how a flag value is sensed.

```
u32 Value;          // location where flag value is stored
Result = adi_flag_Sense(ADI_FLAG_PF0, &Value);
                    // senses the flag value
if (Value == TRUE) {
                    // flag is set to logical high
} else {
                    // flag is set to logical low }
```

The above fragment illustrates how a flag value can be sensed in a polled type method. Alternatively, a callback function can be used to alert the application when an event, such as a flag changing value, has occurred.

Installing a Callback Function

To avoid polling and instead invoke a callback function when a pin state changes, the application should install a callback function. The following fragment illustrates how to install a callback function and the actual callback function.

```
...
Result = adi_flag_InstallCallback(ADI_FLAG_PF1, ADI_INT_PFA,
                  ADI_FLAG_TRIGGER_LEVEL_HIGH,
                  TRUE, (void *)0x12345678, NULL, Callback);
...

void Callback(void *ClientHandle, u32 Event, void *pArg) {
     // ClientHandle = 0x12345678
     // Event = ADI_FLAG_EVENT_CALLBACK
     switch ((ADI_FLAG_ID)pArg) {
     case ADI_FLAG_PF1:
             // do processing when PF1 changes state break;
     }
}
```

When the callback function is invoked, the `ClientHandle` parameter is the value that is given when the callback is installed (in this case, `0x12345678`), the `Event` is the `ADI_FLAG_EVENT_CALLBACK` value, and the `pArg` parameter contains the flag ID that triggered the callback.

Suspending and Resuming Callbacks

If the application needs to temporarily suspend callback processing, the following fragment illustrates how to do it.

```
Result = adi_flag_SuspendCallbacks(ADI_FLAG_PF1);
```

The callback function for that flag is no longer called when the trigger condition occurs. The following fragment illustrates how to resume callback processing.

```
Result = adi_flag_ResumeCallbacks(ADI_FLAG_PF1);
```

Now the callback function will again be invoked when the trigger condition occurs.

Removing Callbacks

If an application no longer needs the callback, it removes the callback with the following call.

```
Result = adi_flag_RemoveCallback(ADI_FLAG_PF1);
```

The callback function is no longer invoked, and the callback function itself is removed from the flag service. The memory used to manage that callback is now available to the flag service to use for another callback function should another callback be installed.

Termination

When the functionality provided by the flag service is no longer required, the application terminates the service. The following fragment terminates the flag service.

```
ADI_FLAG_RESULT Result;        // return value

Result = adi_flag_Terminate();
```

After termination, any memory provided to the flag service during installation is freed up for reuse by the application.

Flag Service API Reference

This section provides the flag service API.

The information in this section was accurate at the time this document was created. However, always check the include file for the flag service, `adi_flag.h`, for the most up-to-date information.

Notation Conventions

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

Name – 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_flag_Clear

Description

The `adi_flag_Clear` function sets the value of the flag to a logical 0, driving low.

Prototype

```
ADI_FLAG_RESULT adi_flag_Clear(
        ADI_FLAG_ID        FlagID
);
```

Arguments

`FlagID`	Enumerator value that uniquely identifies the flag to close

Return Value

`ADI_FLAG_RESULT_SUCCESS`	Function completed successfully.
Any other value	Error has occurred. See "ADI_FLAG_RESULT" on page 7-33 for a list of return codes.

adi_flag_Close

Description

The `adi_flag_Close` function is called when a particular flag is no longer needed by the application. At present, this function does nothing but returns immediately to the caller. Future Blackfin devices may require this function manipulate the hardware in some way when closing a flag.

Prototype

```
ADI_FLAG_RESULT adi_flag_Close(
       ADI_FLAG_ID       FlagID
);
```

Arguments

`FlagID`	Enumerator value that uniquely identifies the flag to close

Return Value

`ADI_FLAG_RESULT_SUCCESS`	Function completed successfully.
Any other value	Error has occurred. See "ADI_FLAG_RESULT" on page 7-33 for a list of return codes.

adi_flag_Init

Description

The adi_flag_Init function provides and initializes memory for the flag service. This function should only be called once per core. If called by more than one core, provide separate memory areas.

Prototype

```
ADI_FLAG_RESULT adi_flag_Init(
        void                   *pMemory,
        const size_t           MemorySize,
        u32                    *pMaxEntries,
        void                   *pEnterCriticalArg
);
```

Arguments

pMemory	Pointer to an area of memory used to hold the data required by the flag service
MemorySize	Size, in bytes, of memory supplied for the flag service data
pMaxEntries	On return, holds the maximum number of simultaneously active callback functions that can be supported using the supplied memory
pEnterCriticalArg	Handle to data area containing critical region data. This is passed to adi_int_EnterCriticalRegion where it is used internally of the module. See "Interrupt Manager" on page 2-1 for further details.

Return Value

`ADI_FLAG_RESULT_SUCCESS`	Flag service was successfully initialized.
Any other value	Error has occurred. See "ADI_FLAG_RESULT" on page 7-33 for a list of return codes.

adi_flag_Open

Description

The `adi_flag_Open` function configures any hardware necessary for the specified flag to operate as a general-purpose I/O pin. Depending on the specific Blackfin device, this function initializes any hardware necessary for the operation of the flag. For example, on ADSP-BF534/536/537 (and future) processors, this function configures the port control logic, via the port control system service, for the flag to be used as a general-purpose I/O pin. On ADSP-BF531/532/533 and ADSP-BF561 processors, this function does nothing and simply returns to the caller.

Prototype

```
ADI_FLAG_RESULT adi_flag_Open(
        ADI_FLAG_ID      FlagID
);
```

Arguments

`FlagID`	Enumerator value that uniquely identifies the flag to open

Return Value

`ADI_FLAG_RESULT_SUCCESS`	Function completed successfully.
Any other value	Error has occurred. See "ADI_FLAG_RESULT" on page 7-33 for a list of return codes.

adi_flag_Sense

Description

The `adi_flag_Sense` function senses the value of a flag. The function stores the value `TRUE` in the provided location if the flag is a logical 1. Otherwise, the function stores the value `FALSE` in the provided location.

Prototype

```
ADI_FLAG_RESULT adi_flag_Sense(
        ADI_FLAG_ID            FlagID,
        u32                    *pValue
);
```

Arguments

`FlagID`	Enumerator value that uniquely identifies the flag to control
`pValue`	Pointer to location where the value of the flag is stored

Return Value

`ADI_FLAG_RESULT_SUCCESS`	Function completed successfully.
Any other value	Error has occurred. See "ADI_FLAG_RESULT" on page 7-33 for a list of return codes.

adi_flag_Toggle

Description

The `adi_flag_Toggle` function inverts the current value of the flag. If the flag is a logical 1 (driving high), this function makes the flag a logical 0 (driving low). If the flag is a logical (0 driving low), this function makes the flag a logical 1 (driving high).

Prototype

```
ADI_FLAG_RESULT adi_flag_Toggle(
        ADI_FLAG_ID             FlagID
);
```

Arguments

`FlagID`	Enumerator value that uniquely identifies the flag to control

Return Value

`ADI_FLAG_RESULT_SUCCESS`	Function completed successfully.
Any other value	Error has occurred. See "ADI_FLAG_RESULT" on page 7-33 for a list of return codes.

adi_flag_Terminate

Description

The `adi_flag_Terminate` function closes the flag service. Any installed callbacks are removed, and all memory provided at initialization is returned. Once terminated, the initialization function must be called again before using any of the flag service functions.

Prototype

```
ADI_FLAG_RESULT adi_flag_Terminate(void);
```

Arguments

The function takes no arguments.

Return Value

`ADI_FLAG_RESULT_SUCCESS`	Function completed successfully.
Any other value	Error has occurred. See "ADI_FLAG_RESULT" on page 7-33 for a list of return codes.

adi_flag_Set

Description

The `adi_flag_Set` function sets the value of the flag to a logical 1, driving high.

Prototype

```
ADI_FLAG_RESULT adi_flag_Set(
        ADI_FLAG_ID         FlagID
);
```

Arguments

`FlagID`	Enumerator value that uniquely identifies the flag to control

Return Value

`ADI_FLAG_RESULT_SUCCESS`	Function completed successfully.
Any other value	Error has occurred. See "ADI_FLAG_RESULT" on page 7-33 for a list of return codes.

Operation

adi_flag_SetDirection

Description

The `adi_flag_SetDirection` function sets the flag direction as an input or an output. If set as an input, any on-chip input buffer for the flag is also enabled. If set as an output, any on-chip input buffer for the flag is disabled.

Prototype

```
ADI_FLAG_RESULT adi_flag_SetDirection(
        ADI_FLAG_ID                 FlagID,
        ADI_FLAG_DIRECTION          Direction
);
```

Arguments

`FlagID`	Enumerator value that uniquely identifies the flag to control
`Direction`	Direction to which the flag is configured

Return Value

`ADI_FLAG_RESULT_SUCCESS`	Function completed successfully.
Any other value	Error has occurred. See "ADI_FLAG_RESULT" on page 7-33 for a list of return codes.

adi_flag_SetTrigger

Description

The `adi_flag_SetTrigger` function sets the trigger condition that generates a callback. This function is not typically called by clients, as setting the trigger condition is taken care of automatically by the `adi_flag_InstallCallback` function.

This function is provided as a convenience for users who want an extra measure of control on callbacks. The function can also be used to change the trigger conditions for a callback without removing and then reinstalling the callback.

Prototype

```
ADI_FLAG_RESULT adi_flag_SetTrigger(
        ADI_FLAG_ID                 FlagID,
        ADI_FLAG_TRIGGER            Trigger
);
```

Arguments

`FlagID`	Enumerator value that uniquely identifies the flag whose callbacks are resumed
`Trigger`	Trigger condition that generates the callback

Return Value

`ADI_FLAG_RESULT_SUCCESS`	Function completed successfully.
Any other value	Error has occurred. See "ADI_FLAG_RESULT" on page 7-33 for a list of return codes.

adi_flag_InstallCallback

Description

The `adi_flag_InstallCallback` function installs a callback function that is invoked should the specified trigger condition for the given flag occur. Note that the function provided by the caller is a callback function, not an interrupt handler. This function does not alter the flag values, direction, and so on.

Prototype

```
ADI_FLAG_RESULT adi_flag_InstallCallback(
        ADI_FLAG_ID                  FlagID,
        ADI_INT_PERIPHERAL_ID        PeripheralID,
        ADI_FLAG_TRIGGER             Trigger,
        u32                          WakeupFlag,
        void                         *ClientHandle,
        ADI_DCB_HANDLE               DCBHandle,
        ADI_DCB_CALLBACK_FN          ClientCallback
);
```

Arguments

`FlagID`	Enumerator value that uniquely identifies which flag the callback is assigned
`PeripheralID`	Peripheral ID that specifies the system interrupt for the flag to use (see interrupt manager and `adi_int.h`.)
`Trigger`	Trigger condition that generates the callback
`WakeupFlag`	Flag indicating if the processor has woken up from a low power state if the trigger occurs
`ClientHandle`	Identifier defined and supplied by the client. This value is passed to the callback function.
`DCBHandle`	Either NULL if using live callbacks or the handle to the deferred callback service that is used for callbacks
`ClientCallback`	Address of the client's callback function

Return Value

`ADI_FLAG_RESULT_SUCCESS`	Flag service was successfully initialized.
Any other value	Error has occurred. See "ADI_FLAG_RESULT" on page 7-33 for a list of return codes.

adi_flag_RemoveCallback

Description

The `adi_flag_RemoveCallback` function removes the callback from the specified flag and disables the generation of the interrupt that triggers the callback. This function does not alter the flag values, direction, and so on.

Calling `adi_flag_RemoveCallback` from within a callback routine is not supported and will result in undefined behavior.

Prototype

```
ADI_FLAG_RESULT adi_flag_RemoveCallback(
        ADI_FLAG_ID                FlagID
);
```

Arguments

`FlagID`	Enumerator value that uniquely identifies the flag whose callback is removed

Return Value

`ADI_FLAG_RESULT_SUCCESS`	Function completed successfully.
Any other value	Error has occurred. See "ADI_FLAG_RESULT" on page 7-33 for a list of return codes.

adi_flag_ResumeCallbacks

Description

The adi_flag_ResumeCallbacks function resumes callback generation that was temporarily suspended by the adi_flag_SuspendCallbacks function. This function simply re-enables the interrupt that causes the callback to occur.

Prototype

```
ADI_FLAG_RESULT adi_flag_ResumeCallbacks(
        ADI_FLAG_ID                 FlagID,
        ADI_INT_PERIPHERAL_ID       PeripheralID
);
```

Arguments

FlagID	Enumerator value that uniquely identifies the flag whose callbacks are resumed
PeripheralID	Peripheral ID that specifies the system interrupt for the flag to use (see interrupt manager and adi_int.h.)

Return Value

ADI_FLAG_RESULT_SUCCESS	Function completed successfully.
Any other value	Error has occurred. See "ADI_FLAG_RESULT" on page 7-33 for a list of return codes.

adi_flag_SuspendCallbacks

Description

The adi_flag_SuspendCallbacks function temporarily suspends callbacks for a given flag but does not uninstall the callback. This function is typically used in conjunction with the adi_flag_ResumeCallbacks function. This function simply disables the interrupt that causes the callback to occur.

Prototype

```
ADI_FLAG_RESULT adi_flag_SuspendCallbacks(
        ADI_FLAG_ID                 FlagID,
        ADI_INT_PERIPHERAL_ID       PeripheralID
);
```

Arguments

FlagID	Enumerator value that uniquely identifies the flag whose callbacks are suspended
PeripheralID	Peripheral ID that specifies the system interrupt for the flag to use (see interrupt manager and adi_int.h.)

Return Value

ADI_FLAG_RESULT_SUCCESS	Function completed successfully.
Any other value	Error has occurred. See "ADI_FLAG_RESULT" on page 7-33 for a list of return codes.

Public Data Types, Enumerations, and Macros

This section defines the public data structures and enumerations used by the flag service.

Always check the include file for the flag service, `adi_flag.h`, for the most up-to-date information.

ADI_FLAG_ID

The `ADI_FLAG_ID` enumeration type uniquely defines each flag in the processor being targeted. To the client application, flag IDs are simply values that identify a specific flag; however, each flag ID actually consists of two pieces of information.

The upper 16 bits of the enumeration is the bit position within the port that corresponds to that flag. The lower 16 bits is the offset to the flags system registers that control the flag. A macro is provided that creates a flag ID, given a bit position and register offset. Macros are also provided to extract the bit position and register offset when given a flag ID.

Applications rarely, if ever, need access to these macros; however, they are provided in the `adi_flag.h` file for reference. Applications typically use only the completed flag ID value. The preset enumeration values are too numerous to list here, but take the form `ADI_FLAG_Pxy`, where “`x`” is the port ID and “`y`” is the index into the port for the flag. Refer to `adi_flag.h` for further details.

Associated Macros

These macros are defined for internal use by the flag service.

`ADI_FLAG_CREATE_FLAG_ID`	Creates a flag ID given a bit position and register offset
`ADI_FLAG_GET_BIT`	Gets the bit position given a flag ID
`ADI_FLAG_GET_OFFSET`	Gets the register offset given a flag ID
`ADI_FLAG_GET_MASK`	Creates a mask for a given flag ID that can be used to manipulate hardware control registers for the given flag.

ADI_FLAG_DIRECTION

The `ADI_FLAG_DIRECTION` enumeration defines the direction (input or output) for a flag pin.

`ADI_FLAG_DIRECTION_INPUT`	Flag is configured as an input.
`ADI_FLAG_DIRECTION_OUTPUT`	Flag is configured as an output.

ADI_FLAG_EVENT

The `ADI_FLAG_EVENT` enumeration defines the type of callback event that occurred. There is only one value, `ADI_FLAG_EVENT_CALLBACK`. This enumeration type is different from all other event types for system services—a single callback function can be used for any service, regardless of the event it processes. Event codes for the flag service begin with the value `ADI_FLAG_ENUMERATION_START`, for easy identification.

`ADI_FLAG_EVENT_CALLBACK`	The trigger condition for the specified flag occurred.

ADI_FLAG_RESULT

Each API function of the flag service returns an ADI_FLAG_RESULT enumeration as a return code. As with all system services, generic success is defined as 0 and generic failure is defined as 1. This allows the calling function to quickly evaluate the return code for a zero or nonzero value. All detailed result codes for the flag service begin with the value ADI_FLAG_ENUMERATION_START for easy identification.

ADI_FLAG_RESULT_SUCCESS=0	Generic success = 0
ADI_FLAG_RESULT_FAILED=1	Generic failure = 1
ADI_FLAG_RESULT_INVALID_FLAG_ID	(0x80001) invalid flag ID
ADI_FLAG_RESULT_INTERRUPT_MANAGER_ERROR	(0x80002) error returned from interrupt manager
ADI_FLAG_RESULT_ERROR_REMOVING_CALLBACK	(0x80003) no callback installed for given ID
ADI_FLAG_RESULT_ALL_IN_USE	(0x80004) all flag slots in use
ADI_FLAG_RESULT_PORT_CONTROL_ERROR	(0x80005) error within port control
ADI_FLAG_RESULT_NOT_CAPABLE	(0x80006) given flag not capable of function requested
ADI_FLAG_RESULT_TRIGGER_TYPE_NOT_SUPPORTED	(0x80007) trigger type is not supported
ADI_FLAG_RESULT_CANT_MAP_FLAG_TO_INTERRUPT	(0x80008) cannot map flag to given interrupt peripheral ID
ADI_FLAG_RESULT_NOT_MAPPED_TO_INTERRUPT	(0x80009) flag not mapped to interrupt
ADI_FLAG_RESULT_CALLBACK_NOT_INSTALLED	(0x8000a) no callback is installed for given flag
ADI_FLAG_RESULT_BAD_CALLBACK_MEMORY_SIZE	(0x8000b) ADI_FLAG_CALLBACK_MEMORY macro is invalid (internal error)

ADI_FLAG_TRIGGER

The `ADI_FLAG_TRIGGER` enumeration type is used to specify the condition that, when triggered, causes the application's callback function to be invoked.

`ADI_FLAG_TRIGGER_LEVEL_HIGH`	Flag set when voltage on pin is at recognized 'digital' high level
`ADI_FLAG_TRIGGER_LEVEL_LOW`	Flag set when voltage on pin is at recognized 'digital' low level
`ADI_FLAG_TRIGGER_RISING_EDGE`	Flag set when voltage on pin rises from low to high level (rising edge)
`ADI_FLAG_TRIGGER_FALLING_EDGE`	Flag set when voltage on pin falls from high to low level (falling edge)
`ADI_FLAG_TRIGGER_BOTH_EDGE`	Flag set on both rising and falling edges

8 TIMER SERVICE

The timer service, within the system services library, provides the application with an easy-to-use interface into the core timers, watchdog timers, and general-purpose timers of the Blackfin processor.

This chapter contains the following sections:

Introduction

Using the capabilities of other system services, the timer service allows the client to control and coordinate the all timers in a consistent fashion, regardless of processor derivative. The service also provides the means for clients to install callback functions that are notified upon timer expirations.

Use of the timer service is dependent on the use of both the interrupt manager and the deferred callback (DCB) manager for its full operation. If callbacks are not deferred, but rather are live, the DCB manager is not required. If callbacks are not required, neither the interrupt manager nor DCB manager is required.

In order to reduce the pin count of devices, timer pins are sometimes muxed onto the same pins as other peripherals. The timer service does not provide arbitration functionality to control pin muxing. It is the responsibility of the client program to ensure that peripherals and the timer service do not use the same pin simultaneously. For ADSP-BF531/532/533 and ADSP-BF561 processors, this entails ensuring that the relevant peripheral control registers are correctly set. For ADSP-BF534/536/537 (and future) processors, the timer service automatically invokes the port control service to affect any pin multiplexer changes. No user intervention is required.

The timer service uses an unambiguous naming convention to safeguard against conflicts with other software libraries provided by Analog Devices or elsewhere. All enumeration values and `typedef` statements use the `ADI_TMR_` prefix, and functions and global variables use the lowercase `adi_tmr_` equivalent.

Each function within the timer service application program interface (API) returns an error code of the type `ADI_TMR_RESULT`. Like all system services, a return value of 0 (`ADI_TMR_RESULT_SUCCESS`) indicates no errors. A nonzero value indicates an error. Like all system services, timer service error codes are unique from all other system services. The `adi_tmr.h` include file lists all error codes that the timer service returns.

Parameter checking in the debug versions of the system services library provides a more complete test of API function parameters and for conditions that may cause errors. Analog Devices strongly recommends that development work be done using the debug versions of the system service library, while final test and deployment be done with the release version of the library.

Operation

This section describes the overall operation of the timer service. Details on the application programming interface (API) can be found later in this chapter.

Initialization

Prior to using the timer service, the client must first initialize the service. In order to initialize the service, the client passes to the initialization function, `adi_tmr_Init()`, a parameter is passed to the critical region function should the timer service need to protect a critical region of code, and optionally a contiguous piece of memory that the service can use for flag callbacks.

The timer service provides a facility where, if so directed by the client, a callback function supplied by the client can be invoked should a timer expire. See the "Callbacks" on page 8-7 for more information on how callbacks operate. Unlike some other services in which the client provides memory to the service for use by the service, the timer service requires no additional memory.

Termination

When the client no longer requires the functionality of the timer service, the termination function, `adi_tmr_Terminate()`, is called. This function uninstalls any installed timer callbacks and closes any open timers.

Timer IDs

All API functions within the timer service, other than the initialize and terminate functions, are passed a parameter that identifies the timer(s) to be controlled. The include file for the timer service, `adi_tmr.h`, defines the timer IDs for each timer supported by the processor. The timer ID parameter is defined as a `u32` type, but it is not a simple enumeration value. The timer ID is actually a complex value that contains information specific to the timer and also allows them to be OR'ed together so that multiple timers can be enabled and disabled simultaneously.

Basic Timer Functions

The functions described in this section are common to all types of timers: general-purpose timers, core timers, and watchdog timers. Any individual timer ID, regardless whether it is a general-purpose timer ID, core timer ID, or watchdog timer ID, can be passed to these functions.

adi_tmr_Open

The `adi_tmr_Open()` function is called to open the timer. Depending on the specific Blackfin device, this function initializes the hardware necessary for the operation of the timer. This function also resets the timer to its default settings. Refer also to "adi_tmr_Open" on page 8-17.

adi_tmr_Close

When a timer is no longer needed, the `adi_tmr_Close()` function is called to close and shut down the timer. Presently for all Blackfin processors, this function does nothing but returns immediately to the caller. Future Blackfin processors may require this function to manipulate the hardware in some way when closing a timer. Refer also to "adi_tmr_Close" on page 8-19.

adi_tmr_Reset

Should an application need to reset the timer to the default settings at a time other than when it is opened, use the `adi_tmr_Reset()` function. The configuration register for the timer is reset to its power-up value, the error status is cleared, and so on. Refer also to "adi_tmr_Reset" on page 8-20.

General-Purpose Timer Functions

The functions described in this section are for general-purpose timers only. These functions return an error if they are passed other types of timer IDs, such as core timer IDs or watchdog timer IDs.

adi_tmr_GPControl

The `adi_tmr_GPControl()` function configures a general-purpose timer. This function is passed the timer ID of the timer being controlled, a command ID specifying the parameter of the function being addressed, and a command-specific parameter. The list of command IDs applicable to general-purpose timers and the corresponding command-specific parameters are described in "ADI_TMR_GP_CMD" on page 8-36. Refer also to "adi_tmr_GPControl" on page 8-23.

adi_tmr_GPGroupEnable

The `adi_tmr_GPGroupEnable()` function enables or disables a single general-purpose timer or a group of general-purpose timers. This function is passed a parameter that is either a single general-purpose timer ID or an OR'ing of any number of general-purpose timer IDs, and a flag that indicates whether the group of timers is enabled (a value of `TRUE`) or disabled (a value of `FALSE`).

The function uses best efforts to simultaneously enable or disable the group of timers. If the underlying hardware of the specific Blackfin device allows the timers to be controlled simultaneously, the function takes the necessary action to simultaneously enable or disable the timers. If the underlying hardware does not allow the specified timers to be controlled simultaneously, the function uses best efforts to enable or disable the timers as quickly as possible. Refer also to "adi_tmr_GPGroupEnable" on page 8-24.

Core Timer Functions

The functions described in this section are used for the core timer only.

adi_tmr_CoreControl

The `adi_tmr_CoreControl()` function is used to configure the core timer. Analogous to the general-purpose timer control function, this function is passed a command ID specifying the parameter of the function that is being addressed, and a command-specific parameter. The list of command IDs applicable to the core timer and the corresponding command-specific parameters are described in "ADI_TMR_CORE_CMD" on page 8-33. Refer also to "adi_tmr_CoreControl" on page 8-21.

Watchdog Timer Functions

The functions described in this section are used for the watchdog timer only.

adi_tmr_WatchdogControl

The `adi_tmr_WatchdogControl()` function configures the watchdog timer. Analogous to the general-purpose timer and core timer control functions, this function is passed a command ID specifying the parameter of the function being addressed, and a command-specific parameter. Refer also to "adi_tmr_WatchdogControl" on page 8-22.

The list of command IDs applicable to the watchdog timer and the corresponding command-specific parameters are described in "ADI_TMR_WDOG_CMD" on page 8-34.

Peripheral Timer Functions

The functions described in this section are used for the general-purpose timers and watchdog timers only. Passing a core timer ID to these functions results in an error being returned to the caller.

adi_tmr_GetPeripheralID

The `adi_tmr_GetPeripheralID()` function can be called to identify the peripheral ID for the specified timer. While not normally required, this function may be useful if finer granularity of interrupt control logic than what is provided by the timer service is required. The peripheral ID value can be passed to functions in the interrupt service. Note that the core timer does not have an associated peripheral ID because the core timer mapping to an IVG level is fixed. Regardless, the core timer ID may still be passed to this function, and the function does not return an error. Refer also to "adi_tmr_GetPeripheralID" on page 8-29.

Callbacks

Like other system services, the timer service uses a callback mechanism in order to notify the client of asynchronous events, such as a timer expiring. Callbacks can be used on all types of timers: general-purpose timers, core timers, and watchdog timers.

The Blackfin processor's timers can be configured to generate interrupts. The timer service provides an internal interrupt handler that processes interrupts from the timer hardware. This interrupt handler makes the appropriate callbacks into the client's application. When a client installs a timer callback, a parameter to the function dictates if the callback is live or deferred. Live callbacks mean that the client's callback function is called at interrupt time. Deferred callbacks mean that callbacks are not made at interrupt time but rather deferred to a lower priority using a specified deferred callback service.

When using the callback capability of the timer service, the client does not need to take any other action outside the timer service API. No calls to the interrupt manager or deferred callback service, other than initialization of those services, are required.

Note that it is possible for clients to use all capabilities of the timer service and not use any of the callback capabilities.

adi_tmr_InstallCallback

The `adi_tmr_InstallCallback()` function is used to install a callback to a specified timer. In addition to the timer ID, the client provides a wakeup flag, the callback function address, a client handle, and a deferred callback service handle.

The wakeup flag indicates whether the processor is awake from a low-power state, should the timer event occur.

The callback function address specifies a callback function of the type `ADI_DCB_CALLBACK_FN` (see "Deferred Callback Manager" for more information). When invoked, the callback function is passed the following three parameters:

- `ClientHandle` – a value provided by the client when the callback is installed
- `ADI_TMR_EVENT_TIMER_EXPIRED` – indicates a timer callback event
- `TimerID` – the timer ID of the timer that generates the callback

When the deferred callback service handle parameter passed to the `adi_tmr_InstallCallback` function is NULL, the callback is executed live, meaning it is invoked at interrupt time. If the deferred callback service handle parameter is non-NULL, the timer service uses the specified deferred callback service to invoke the callback.

A single callback function can be used and installed for any number of timers; the callback function can use the `TimerID` parameter to determine which timer generated the callback. Note, however, that only one callback should be installed for a given timer.

This function does not alter timer control (such as direction) at all. Refer also to "adi_tmr_InstallCallback" on page 8-26.

adi_tmr_RemoveCallback

The `adi_tmr_RemoveCallback()` function is used to remove a callback from a specified timer. This function disables interrupt generation for the timer and removes the callback from its internal tables. Unless reinstalled, no further callbacks occur for the specified timer. This function does not alter timer control in any way. Refer also to "adi_tmr_RemoveCallback" on page 8-28.

Coding Example

This section provides code samples, illustrating how to access and use the functionality provided by the timer service. This example initializes the timer service, configures a couple of general-purpose timers, the core timer, and the watchdog timer. The use of callbacks is also illustrated. All timer service functions return an error code. In practice, check this error code to ensure the function completed successfully. For the purposes of this example only, the return value is not checked.

Initialization

Prior to using the timer service, it must be initialized. The following fragment initializes the service.

```
ADI_TMR_RESULT Result;              // return value

Result = adi_tmr_Init(NULL);
```

Upon completion of this function, the timer service is initialized and ready for use. This function does not alter the timers in any way but simply initialized internal data structures.

Opening a Timer

After the service is initialized, any timers that you need to use can be opened. In this example, two general-purpose timers, the core timer, and watchdog timer are opened

```
Result = adi_tmr_Open(ADI_TMR_GP_TIMER_0);
Result = adi_tmr_Open(ADI_TMR_GP_TIMER_1);
Result = adi_tmr_Open(ADI_TMR_CORE_TIMER);
Result = adi_tmr_Open(ADI_TMR_WDOG_TIMER);
```

The open function opens the timer for use and resets the timer to its power-up values, clearing any pending status, and so on.

Configuring a Timer

After the timer is opened, the timer can be configured. The `adi_tmr_GPControl()`, `adi_tmr_CoreControl()` and `adi_tmr_WatchdogControl()` functions are used to configure general-purpose timers, core timers, and the watchdog timers, respectively.

Each of these functions are provided with a command ID, typically specifying the parameter to control and a value for the parameter. (Note that the general-purpose control function also is passed a timer ID specifying the timer being controlled.) Commands to timers can be given individually or collectively as a table.

The following fragment illustrates both methods.

```
ADI_TMR_CORE_CMD_VALUE_PAIR CoreTable [] = {
      {ADI_TMR_CORE_CMD_SET_COUNT,        (void *)0x12345678 },
      { ADI_TMR_CORE_CMD_SET_PERIOD,      (void *)0xabcdef },
      { ADI_TMR_CORE_CMD_SET_SCALE,       (void *)0x10 },
      { ADI_TMR_CORE_CMD_SET_ACTIVE_MODE, (void *)TRUE },
      { ADI_TMR_CORE_CMD_END,             NULL  },
};
Result = adi_tmr_CoreControl(ADI_TMR_CORE_CMD_TABLE, CoreTable);

Result = adi_tmr_GPControl(ADI_TMR_GP_TIMER_0,
              ADI_TMR_GP_CMD_SET_PERIOD,     (void *)0x800000);
Result = adi_tmr_GPControl(ADI_TMR_GP_TIMER_0,
              ADI_TMR_GP_CMD_SET_WIDTH,      (void *)0x400000);
Result = adi_tmr_GPControl(ADI_TMR_GP_TIMER_0,
              ADI_TMR_GP_CMD_SET_TIMER_MODE,  (void *)0x1);

Result = adi_tmr_GPControl(ADI_TMR_GP_TIMER_1,
              ADI_TMR_GP_CMD_SET_PERIOD,     (void *)0x800000);
Result = adi_tmr_GPControl(ADI_TMR_GP_TIMER_1,
              ADI_TMR_GP_CMD_SET_WIDTH,      (void *)0x400000);
Result = adi_tmr_GPControl(ADI_TMR_GP_TIMER_1,
              ADI_TMR_GP_CMD_SET_TIMER_MODE,  (void *)0x1);

Result = adi_tmr_WatchdogControl
              (ADI_TMR_WDOG_CMD_EVENT_SELECT, (void *)0x0);
Result = adi_tmr_WatchdogControl
              (ADI_TMR_WDOG_CMD_SET_COUNT,  (void *)0x12345678);
```

Note in the above fragment that the core timer was enabled immediately after being configured and the watchdog and general-purpose timers were not enabled. Any timer can be enabled via a command table, typically the last entry in the table. For illustrative purposes, enabling the watchdog and general-purpose timers is shown separately in "Enabling and Disabling Timers".

Enabling and Disabling Timers

After the timer is configured, it can be enabled. When using a command table, the timer can be enabled as a command in the table as shown in "Configuring a Timer" on page 8-10. Typically, the command to enable the timer is the last entry in the table. Alternatively, timers can be enabled by a separate call to the appropriate control function. Further, general-purpose timers can be simultaneously enabled and disabled as a group.

The following fragment illustrates how to enable the watchdog timer and then simultaneously enable general-purpose timers 0 and 1.

```
Result = adi_tmr_WatchdogControl
            (ADI_TMR_WDOG_CMD_ENABLE_TIMER, (void *)TRUE);

Result = adi_tmr_GPGroupEnable
            (ADI_TMR_GP_TIMER_0 | ADI_TMR_GP_TIMER_1, TRUE);
```

When a timer is disabled, it can be disabled as part of a command table (though this is unlikely). More often, timers are disabled via a single control function call. As with the enabling of general-purpose timers, timers can be disabled simultaneously. The following code fragment illustrates how to disable the watchdog timer and simultaneously disable general-purpose timers 0 and 1.

```
Result = adi_tmr_WatchdogControl
             (ADI_TMR_WDOG_CMD_ENABLE_TIMER, (void *)FALSE);

Result = adi_tmr_GPGroupEnable
             (ADI_TMR_GP_TIMER_0 | ADI_TMR_GP_TIMER_1, FALSE);
```

Installing a Callback Function

While applications can install hardware interrupt service routines (ISRs) directly to process interrupts from timers (see "Interrupt Manager"), the timer service provides a simple, easy-to-use callback mechanism that provides equivalent functionality.

The following code fragment illustrates how to install a callback function. Different callback functions can be used for each timer, or a single callback function can be used for any number of timers. The fragment shows installation of a single callback function for two general-purpose timers, the core timer, and a watchdog timer. The switch statement within the callback function identifies which timer generated the callback.

```
...
Result = adi_tmr_InstallCallback
  (ADI_TMR_GP_TIMER_0, TRUE, (void *)0x00000000, NULL, Callback);
Result = adi_tmr_InstallCallback
  (ADI_TMR_GP_TIMER_1, TRUE, (void *)0x11111111, NULL, Callback);
Result = adi_tmr_InstallCallback
  (ADI_TMR_CORE_TIMER, TRUE, (void *)0x22222222, NULL, Callback);
Result = adi_tmr_InstallCallback
  (ADI_TMR_WDOG_TIMER, TRUE, (void *)0x33333333, NULL, Callback);
...
void Callback(void *ClientHandle, u32 Event, void *pArg) {
         // Event = ADI_TMR_EVENT_TIMER_EXPIRED
         switch ((u32)pArg) {
         case ADI_TMR_GP_TIMER_0:
                  // do processing when gp timer 0 expires
                  // ClientHandle = 0x00000000
```

```
                break;
        case ADI_TMR_GP_TIMER_1:
                // do processing when gp timer 1 expires
                // ClientHandle = 0x11111111
                break;
        case ADI_TMR_CORE_TIMER:
                // do processing when core timer expires
                // ClientHandle = 0x22222222
                break;
        case ADI_TMR_WDOG_TIMER:
                // do processing when watchdog timer expires
                // ClientHandle = 0x33333333
        break;
        }
}
```

When the callback function is invoked, the `ClientHandle` parameter is the value given when the callback was installed, the `Event` is the `ADI_TMR_EVENT_TIMER_EXPIRED` value, and the `pArg` parameter contains the timer ID that triggered the callback. This example passes in a NULL for the deferred callback service handle, so callbacks are live.

Removing Callbacks

Should an application no longer need the callback, it can remove the callback without affecting the other timer settings. The following fragment illustrates how to remove the callbacks.

```
Result = adi_tmr_RemoveCallback(ADI_TMR_GP_TIMER_0);

Result = adi_tmr_RemoveCallback(ADI_TMR_GP_TIMER_1);

Result = adi_tmr_RemoveCallback(ADI_TMR_CORE_TIMER);

Result = adi_tmr_RemoveCallback(ADI_TMR_WDOG_TIMER);
```

The callback functions are no longer invoked and the callback functions themselves are removed from the timer service.

Termination

When the functionality provided by the timer service is no longer required, the application terminates the service. The following fragment terminates the timer service.

```
Result = adi_tmr_Terminate();
```

After termination, the timer service must be reinitialized before using any of the timer service function.

Timer Service API Reference

This section provides the timer service application programming interface (API).

ⓘ The information in this section was accurate at the time this document was created. However, the include file for the timer service, `adi_tmr.h`, should be checked for current information.

Notation Conventions

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

Name – 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_tmr_Init

Description

The `adi_tmr_Init()` function initializes internal data structures of the timer service. This function should be called only once per core.

Prototype

```
ADI_TMR_RESULT adi_tmr_Init(
       void      *pCriticalRegionArg
);
```

Arguments

`pCriticalRegionArg`	Handle to data area containing critical region data. This is passed to `adi_int_EnterCriticalRegion` where it is used internally of the module. See "Interrupt Manager" on page 2-1 for further details.

Return Value

`ADI_TMR_RESULT_SUCCESS`	Timer service was successfully initialized.
Any other value	Error has occurred. See "ADI_TMR_RESULT" on page 8-32 for a list of return codes.

adi_tmr_Open

Description

The `adi_tmr_Open()` function opens a timer. The timer's registers are reset to the power-up values, status conditions are cleared, and so on. Future Blackfin devices may require this function take additional action to manipulate the hardware in some way when opening a timer.

On multi-core Blackfin devices, each core has its own core timer. However, the watchdog timers and general-purpose timers are shared between the cores. When running the timer service on multi-core devices, ensure that multiple cores do not attempt to simultaneously use the same watchdog and general-purpose timers.

Prototype

```
ADI_TMR_RESULT adi_tmr_Open(
        u32          TimerID
);
```

Arguments

`TimerID`	Enumerator value that uniquely identifies the timer to open

Return Value

`ADI_TMR_RESULT_SUCCESS`	Operation was successful.
Any other value	Error has occurred. See "ADI_TMR_RESULT" on page 8-32 for a list of return codes.

adi_tmr_Terminate

Description

The `adi_tmr_Terminate()` function closes the timer service. Any installed callbacks are removed. Once terminated, the initialization function must be called again before using any of the timer service functions.

Prototype

```
ADI_TMR_RESULT adi_tmr_Terminate(void);
```

Arguments

The function takes no arguments.

Return Value

`ADI_TMR_RESULT_SUCCESS`	Operation was successful.
Any other value	Error has occurred. See “ADI_TMR_RESULT” on page 8-32 for a list of return codes.

adi_tmr_Close

Description

The `adi_tmr_Close()` function is called when the application no longer requires the service of a timer. At present, this function does nothing but returns immediately to the caller. Future Blackfin devices may require that this function manipulate the hardware in some way when closing a timer.

Prototype

```
ADI_TMR_RESULT adi_tmr_Close(
        u32       TimerID
);
```

Arguments

`TimerID`	Enumerator value that uniquely identifies the timer to close

Return Value

`ADI_TMR_RESULT_SUCCESS`	Operation was successful.
Any other value	Error has occurred. See "ADI_TMR_RESULT" on page 8-32 for a list of return codes.

Operation

adi_tmr_Reset

Description

The `adi_tmr_Reset()` function resets the timer's registers to the power-up values. Any pending status indications (interrupts, and so on) are cleared. As this function is called from within the `adi_tmr_Open` function, there is rarely a need for an application to call this function.

Prototype

```
ADI_TMR_RESULT adi_tmr_Reset(
        u32       TimerID
);
```

Arguments

`FlagID`	Enumerator value that uniquely identifies the timer to reset

Return Value

`ADI_TMR_RESULT_SUCCESS`	Function completed successfully.
Any other value	Error has occurred. See "ADI_TMR_RESULT" on page 8-32 for a list of return codes.

adi_tmr_CoreControl

Description

The `adi_tmr_CoreControl()` function inverts the current value of the flag. If the flag is a logical 1 (driving high), this function makes the flag a logical 0 (driving low). If the flag is a logical 0 (driving low), this function makes the flag a logical 1 (driving high).

Prototype

```
ADI_TMR_RESULT adi_tmr_CoreControl(
        ADI_TMR_CORE_CMD       Command,
        void                   *Value
);
```

Arguments

`Command`	Identifier specifying the timer parameter that is addressed. See "ADI_TMR_CORE_CMD" on page 8-33 for a list of all core timer command identifiers.
`Value`	A command-specific value that is typically the value of the parameter being set or a location into which a value read from the timer is stored

Return Value

`ADI_TMR_RESULT_SUCCESS`	Function completed successfully.
Any other value	Error has occurred. See "ADI_TMR_RESULT" on page 8-32 for a list of return codes.

adi_tmr_WatchdogControl

Description

The `adi_tmr_WatchdogControl()` function configures and controls the settings of the watchdog timer.

Prototype

```
ADI_TMR_RESULT adi_tmr_WatchdogControl(
        ADI_TMR_WDOG_CMD           Command,
        void                       *Value
);
```

Arguments

`Command`	Identifier specifying the timer parameter that is addressed. See "ADI_TMR_WDOG_CMD" on page 8-34 for a list of all watchdog timer command identifiers.
`Value`	A command-specific value that is typically the value of the parameter being set or a location into which a value read from the timer is stored

Return Value

`ADI_TMR_RESULT_SUCCESS`	Function completed successfully.
Any other value	Error has occurred. See "ADI_TMR_RESULT" on page 8-32 for a list of return codes.

adi_tmr_GPControl

Description

The `adi_tmr_GPControl()` function configures and controls the settings of general-purpose timers.

Prototype

```
ADI_TMR_RESULT adi_tmr_GPControl(
        u32                  TimerID,
        ADI_TMR_GP_CMD       Command,
        void                 *Value
);
```

Arguments

`TimerID`	Enumerator value that uniquely identifies the timer to control
`Command`	Identifier specifying the timer parameter that is addressed. See "ADI_TMR_GP_CMD" on page 8-36 for a list of all general-purpose timer command identifiers.
`Value`	A command-specific value that is typically the value of the parameter being set or a location into which a value read from the timer is stored

Return Value

`ADI_TMR_RESULT_SUCCESS`	Function completed successfully.
Any other value	Error has occurred. See "ADI_TMR_RESULT" on page 8-32 for a list of return codes.

adi_tmr_GPGroupEnable

Description

The `adi_tmr_GPGroupEnable()` function simultaneously enables or disables a group of general-purpose timers. The function uses best efforts to simultaneously enable or disable the group of timers. If the underlying hardware of the specific Blackfin device allows the specified timers to be controlled simultaneously, the function takes the necessary action to simultaneously enable or disable the timers. If the underlying hardware does not allow the specified timers to be controlled simultaneously, the function uses best efforts to enable or disable the timers as quickly as possible.

Note that depending on the specific Blackfin device, when enabling the timer(s), this function may additionally configure the port muxing, based on the configuration settings for the timer.

For example, on ADSP-BF534/536/537 (and future) processors, if a general-purpose timer is configured as a PWM timer with the output pin active, when the timer is enabled, this function configures the port control logic, via the port control system service, to enable the `TMRx` pin. On ADSP-BF531/532/533 and ADSP-BF561 processors, no port control logic is necessary. No further user action with the port control service is required.

Prototype

```
ADI_TMR_RESULT adi_tmr_GPGroupEnable(
        u32        TimerID,
        u32        EnableFlag
);
```

Arguments

`TimerIDs`	OR'ing of all general-purpose timer IDs that are simultaneously controlled
`EnableFlag`	A value of `TRUE` indicates the timers are enabled. A value of `FALSE` indicates the timers are disabled.

Return Value

`ADI_TMR_RESULT_SUCCESS`	Function completed successfully.
Any other value	Error has occurred. See "ADI_TMR_RESULT" on page 8-32 for a list of return codes.

Operation

adi_tmr_InstallCallback

Description

The adi_tmr_InstallCallback() function installs a callback function that is invoked when a timer expires. Note that the function provided by the caller is a callback function, not an interrupt handler. This function does not alter timer configurations, values, or other settings in any way.

Prototype

```
ADI_TMR_RESULT adi_tmr_InstallCallback(
        u32                         TimerID,
        u32                         WakeupFlag,
        void                        *ClientHandle,
        ADI_DCB_HANDLE              DCBHandle,
        ADI_DCB_CALLBACK_FN         ClientCallback
);
```

Arguments

TimerID	Enumerator value that uniquely identifies the timer to which the callback is assigned
WakeupFlag	If the trigger occurs, flag indicating the processor is awake from a low-power state
ClientHandle	Identifier defined and supplied by the client. This value is passed to the callback function.
DCBHandle	Either NULL if using live callbacks or the handle to the deferred callback service that is used for callbacks
ClientCallback	Address of the client's callback function

Return Value

`ADI_TMR_RESULT_SUCCESS`	Flag service was successfully initialized.
Any other value	Error has occurred. See "ADI_TMR_RESULT" on page 8-32 for a list of return codes.

adi_tmr_RemoveCallback

Description

The `adi_tmr_RemoveCallback()` function removes the callback from the specified timer. This function does not alter timer configurations, values, or other settings in any way.

ⓘ Calling `adi_tmr_RemoveCallback` from within a callback routine is not supported and will result in undefined behavior.

Prototype

```
ADI_TMR_RESULT adi_tmr_RemoveCallback(
        u32         TimerID
);
```

Arguments

`TimerID`	Enumerator value that uniquely identifies the timer whose callback is removed

Return Value

`ADI_TMR_RESULT_SUCCESS`	Function completed successfully.
Any other value	Error has occurred. See "ADI_TMR_RESULT" on page 8-32 for a list of return codes.

adi_tmr_GetPeripheralID

Description

The `adi_tmr_GetPeripheralID()` function can be called to identify the peripheral ID for the specified timer. Though not normally required, this function may be useful should finer granularity of interrupt control logic than what is provided by the timer service be required. The peripheral ID value can be passed to functions in the interrupt service. Note that the core timer does not have a peripheral ID associated with it as the core timer mapping to an IVG level is fixed. Regardless, the core timer ID may still be passed to this function and the function will not return an error.

Prototype

```
ADI_TMR_RESULT adi_tmr_GetPeripheralID(
        u32                         TimerID,
        ADI_INT_PERIPHERAL_ID       *pPeripheralID
);
```

Arguments

`TimerID`	Enumerator value that uniquely identifies the timer to control
`pPeripheralID`	Pointer to location where the peripheral ID for the specified timer is stored

Return Value

`ADI_TMR_RESULT_SUCCESS`	Function completed successfully.
Any other value	Error has occurred. See "ADI_TMR_RESULT" on page 8-32 for a list of return codes.

Public Data Types, Enumerations, and Macros

This section defines the public data structures and enumerations used by the timer service.

Always check the include file for the timer service, `adi_tmr.h`, for the most up-to-date information.

Timer IDs

The timer service provides a unique identifier for each timer. Timer IDs, which are 32-bit values, are not a simple index; instead, they are a combination of two pieces of information. Bits 27 through 31 of the timer ID are an index that enumerates each timer in the system, including general-purpose, core, and watchdog timers. Bits 0 through 26, which are used for general-purpose timers only, form a mask used by the timer service to enable and disable multiple general-purpose timers simultaneously.

Macros are provided in the `adi_tmr.h` file to create timer IDs, to access the values held in bits 0 through 26, and to access the values in bits 27 through 31. These macros are used internally by the timer service and are not typically used by applications. However, should the application need to iterate through all general-purpose timers, the timer IDs can be created by the `ADI_TMR_CREATE_GP_TIMER_ID(x)` macro, where “`x`” is in the range of 0 to (but not including) the value `ADI_TMR_GP_TIMER_COUNT`.

For example, the following code fragment illustrates how to open all general-purpose timers.

```
u32 i, TimerID;
ADI_TMR_RESULT Result;

for (i = 0; i < ADI_TMR_GP_TIMER_COUNT; i++) {
            TimerID = ADI_TMR_CREATE_GP_TIMER_ID(i);
            Result  = adi_tmr_Open(TimerID); }
```

For most functions in the timer service API, a single timer ID value is passed to the function to identify the timer being addressed. However, the `adi_tmr_GPGroupEnable()` function can take a single timer ID value or a logical OR'ing of multiple timer ID values as a parameter. The structure of the timer ID value allows the single passed-in parameter to identify multiple general-purpose timers to the function.

Associated Macros

These macros are defined for internal use by the timer service.

`ADI_TMR_CREATE_GP_TIMER_ID`	Creates a timer ID given a general-purpose timer index.
`ADI_TMR_CREATE_CORE_TIMER_ID`	Creates a timer ID given a core timer index.
`ADI_TMR_CREATE_WDOG_TIMER_ID`	Creates a timer ID given a watchdog timer index.
`ADI_TMR_GET_TIMER_INDEX`	Gets the timer index given a timer ID.
`ADI_TMR_GET_GP_TIMER_MASK`	Gets the mask for a general-purpose timer(s) given a single timer ID or logical OR'ing of multiple timer IDs.

ADI_TMR_RESULT

Each API function of the timer service returns an ADI_TMR_RESULT enumeration as a return code. Similar to all system services, generic success is defined as 0 and generic failure is defined as 1. This allows the calling function to quickly evaluate the return code for a zero or nonzero value.

All detailed result codes for the timer service begin with the value ADI_TMR_ENUMERATION_START, for easy identification.

Result Code	Description
ADI_TMR_RESULT_SUCCESS	Function executed correctly.
ADI_TMR_RESULT_FAILED	Function execution not completed.
ADI_TMR_RESULT_NOT_SUPPORTED	Operation is not supported.
ADI_TMR_RESULT_BAD_TIMER_ID	(0x70002) TimerID value is invalid.
ADI_TMR_RESULT_BAD_TIMER_IDS	(0x70003) Timer ID values are invalid.
ADI_TMR_RESULT_BAD_TIMER_TYPE	Operation is not appropriate to the timer ID supplied.
ADI_TMR_RESULT_BAD_COMMAND	Invalid command.
ADI_FLAG_RESULT_INTERRUPT_MANAGER_ERROR	Interrupt manager service returned an error.
ADI_TMR_RESULT_CALLBACK_ALREADY_INSTALLED	Callback is already installed on the given timer.

ADI_TMR_EVENT

The ADI_TMR_EVENT enumeration defines the type of callback event that occurred. There is only one value, ADI_TMR_EVENT_TIMER_EXPIRED. This enumeration type is different from all other event types for system services—a single callback function can be used for any service, regardless of the events it processes. Event codes for the timer service begin with the value ADI_TMR_ENUMERATION_START for easy identification.

Event Code	Description
ADI_TMR_EVENT_TIMER_EXPIRED	Given timer expired.

ADI_TMR_CORE_CMD

Table 8-1 lists the commands that can be executed for the core timer. These command IDs are passed as a parameter to the adi_tmr_CoreCommand() function. In addition to the command ID, the Value parameter (a void * type) is also passed to the function. The meaning of the Value parameter depends on the command ID being passed. Table 8-1 also describes the Value parameter for each command ID.

Table 8-1. Commands Executed for Core Timer

Command ID	Value	Description
ADI_TMR_CORE_CMD_TABLE	ADI_TMR_CORE_CMD_VALUE_PAIR *	Start of command table
ADI_TMR_CORE_CMD_END	Ignored	End of command table
ADI_TMR_CORE_CMD_PAIR	ADI_TMR_CORE_CMD_VALUE_PAIR *	Command pair
ADI_TMR_CORE_CMD_SET_ACTIVE_MODE	TRUE – active mode FALSE – low power	Sets active or low power mode of timer.

Table 8-1. Commands Executed for Core Timer (Cont'd)

Command ID	Value	Description
`ADI_TMR_CORE_CMD_ENABLE_TIMER`	`TRUE` – enabled `FALSE` – disabled	Enables or disables the timer.
`ADI_TMR_CORE_CMD_SET_AUTO_RELOAD`	`TRUE` – auto reload `FALSE` – no reload	Enables or disables automatic reloading of timer.
`ADI_TMR_CORE_CMD_HAS_INTERRUPT_OCCURRED`	`u32 *` `TRUE` – enabled `FALSE` – disabled	Indicates if the timer interrupt has occurred.
`ADI_TMR_CORE_CMD_RESET_INTERRUPT_OCCURRED`	Ignored	Clears the indication that the timer interrupt has occurred.
`ADI_TMR_CORE_CMD_SET_COUNT`	`u32`	Sets the count value for the timer.
`ADI_TMR_CORE_CMD_SET_PERIOD`	`u32`	Sets the period value for the timer.
`ADI_TMR_CORE_CMD_SET_SCALE`	`u32`	Sets the scale value for the timer.

ADI_TMR_WDOG_CMD

Table 8-2 lists the commands that can be executed for the watchdog timer. These command IDs are passed as a parameter to the `adi_tmr_WatchdogCommand()` function. In addition to the command ID, the `Value` parameter (a `void *` type) is also passed to the function. The meaning of the `Value` parameter depends on the command ID being passed. Table 8-2 also describes the `Value` parameter for each command ID.

Table 8-2. Commands Executed for Watchdog Timer

Command ID	Value	Description
ADI_TMR_WDOG_CMD_TABLE	ADI_TMR_WDOG_CMD_VALUE_PAIR *	Start of command table
ADI_TMR_WDOG_CMD_END	Ignored	End of command table
ADI_TMR_WDOG_CMD_PAIR	ADI_TMR_WDOG_CMD_VALUE_PAIR *	Command pair
ADI_TMR_WDOG_CMD_EVENT_SELECT	0 – reset 1 – NMI 2 – GP interrupt 3 – no event	Sets the event type that occurs upon expiration of the watchdog timer.
ADI_TMR_WDOG_CMD_ENABLE_TIMER	TRUE – enabled FALSE – disabled	Enables or disables the timer.
ADI_TMR_WDOG_CMD_HAS_EXPIRED	u32 * TRUE – enabled FALSE – disabled	Indicates if the timer has expired.
ADI_TMR_WDOG_CMD_RESET_EXPIRED	Ignored	Clears the indication that the timer has expired.
ADI_TMR_WDOG_CMD_GET_STATUS	u32 *	Stores the current count value into the specified location.
ADI_TMR_WDOG_CMD_SET_COUNT	u32	Sets the current count value.
ADI_TMR_WDOG_CMD_RELOAD_STATUS	u32 *	Reloads the status register from the count register.

ADI_TMR_GP_CMD

Table 8-3 lists the commands that can be executed for general-purpose timers. These command IDs are passed as a parameter to the `adi_tmr_GPCommand()` function. In addition to the command ID, the `Value` parameter (a `void *` type) is also passed to the function. The meaning of the `Value` parameter depends on the command ID being passed. Table 8-3 also describes the `Value` parameter for each command ID.

Table 8-3. Commands Executed for General-Purpose Timers

Command ID	Value	Description
`ADI_TMR_GP_CMD_TABLE`	`ADI_TMR_GP_CMD_VALUE_PAIR *`	Start of command table
`ADI_TMR_GP_CMD_END`	Ignored	End of command table
`ADI_TMR_GP_CMD_PAIR`	`ADI_TMR_GP_CMD_VALUE_PAIR *`	Command pair
`ADI_TMR_GP_CMD_SET_PERIOD`	`u32`	Sets the period value for the timer.
`ADI_TMR_GP_CMD_GET_PERIOD`	`u32 *`	Stores the current period value for the timer in the specified location.
`ADI_TMR_GP_CMD_SET_WIDTH`	`u32`	Sets the width value for the timer.
`ADI_TMR_GP_CMD_GET_WIDTH`	`u32 *`	Stores the current width value for the timer in the specified location.
`ADI_TMR_GP_CMD_GET_COUNTER`	`u32 *`	Stores the counter value for the timer in the specified location.
`ADI_TMR_GP_CMD_SET_TIMER_MODE`	0 – reserved 1 – PWM 2 – WDTH_CAP 3 – EXT_CLK	Sets the operating mode of the timer.

Table 8-3. Commands Executed for General-Purpose Timers (Cont'd)

Command ID	Value	Description
`ADI_TMR_GP_CMD_SET_PULSE_HI`	`TRUE` – positive action pulse `FALSE` – negative action pulse	Sets the pulse action of the timer.
`ADI_TMR_GP_CMD_SET_COUNT_METHOD`	`TRUE` – count to end of period `FALSE` – count to end of width	Sets the count method.
`ADI_TMR_GP_CMD_SET_INTERRUPT_ENABLE`	`TRUE` – enables interrupt generation `FALSE` – disables interrupt generation	Enables or disables interrupt generation for the timer.
`ADI_TMR_GP_CMD_SET_INPUT_SELECT`	`TRUE` – UART_RX or PPI_CLK `FALSE` – TMRx or PF1	Selects the timer input.
`ADI_TMR_GP_CMD_SET_OUTPUT_PAD_DISABLE`	`TRUE` – output pad disabled `FALSE` – output pad enabled	Enables or disables the TMRx pin.
`ADI_TMR_GP_CMD_SET_CLOCK_SELECT`	`TRUE` – PWM_CLK `FALSE` – SCLK	Selects the input clock for the timer.
`ADI_TMR_GP_CMD_SET_TOGGLE_HI`	`TRUE` – PULSE_HI alternated each period `FALSE` – use programmed state	Sets the toggle mode.
`ADI_TMR_GP_CMD_RUN_DURING_EMULATION`	`TRUE` – run during emulation `FALSE` – do not run during emulation	Enables or disables the timer when the device is servicing emulator interrupts.

Table 8-3. Commands Executed for General-Purpose Timers (Cont'd)

Command ID	Value	Description
ADI_TMR_GP_CMD_GET_ERROR_TYPE	u32 * 0 – no error 1 – counter overflow 2 – period register error 3 – width register error	Stores the error type in the specified location.
ADI_TMR_GP_CMD_IS_INTERRUPT_ASSERTED	u32 * TRUE – asserted FALSE – not asserted	Stores the interrupt assertion status in the specified location.
ADI_TMR_GP_CMD_CLEAR_INTERRUPT	Ignored	Clears the timer's interrupt.
ADI_TMR_GP_CMD_IS_ERROR	u32 * TRUE – error FALSE – no error	Stores the error status in the specified location.
ADI_TMR_GP_CMD_CLEAR_ERROR	Ignored	Clears the error status of the timer.
ADI_TMR_GP_CMD_IS_SLAVE_ENABLED	u32 * TRUE – enabled FALSE – disabled	Stores the slave enable status in the specified location.
ADI_TMR_GP_CMD_IMMEDIATE_HALT	Ignored	Immediately stops timer in PWM mode.
ADI_TMR_GP_CMD_ENABLE_TIMER	TRUE – enabled FALSE – disabled	Enables or disables the timer.
ADI_TMR_GP_CMD_SET_ENABLE_DELAY	u32	Number of SCLK cycles to delay between the enabling of each timer

9 PORT CONTROL SERVICE

This chapter describes the port control manager service. This service is available for all Blackfin processors with general-purpose ports starting with the ADSP-BF534/6/7 processors.

This chapter contains the following sections:

Introduction

The port control manager service, within the system services library, provides the client applications' developer with a means of assigning the general-purpose input/output (GPIO) pins to various functions. For instance, the various data, clock, and framing signals required for `SPORT0` usage can be set up with a single call to the `adi_ports_EnableSPORT` function.

Where necessary, the memory-mapped registers for the appropriate peripherals are queried to determine behavior. For example, the port control manager can interrogate the `PPI_CONTROL` register to determine whether two internal frame syncs are required. However, it is the responsibility of the client program to configure the PPI control registers prior to enabling the required flag pins. This is the usual practice within the device driver model, where port control and the setting of flag values are done at the point of enabling dataflow.

The port control service is applicable on processors that support port control, which is standard on all Blackfin processors beginning with (and subsequent to) the ADSP-BF534, ADSP-BF536, and ADSP-BF537 Blackfin processor class.

All supported Blackfin processors share a common basic port control service API, including `adi_ports_Init()` and `adi_ports_Terminate()`.

Initial port control (on the ADSP-BF534/6/7 processors) also included a dedicated `adi_ports_EnableXxx()` function call for each of the respective devices (`Xxx` being PPI, SPI, SPORT, and so on). These enable functions operated on a fixed set of resources and are still supported for compatibility.

As more diverse built-in hardware resources and enhanced port multiplexing capabilities appeared on newer Blackfin architectures, the port control enable functions `(adi_ports_EnableXxx()`) were moved into the respective device driver code as a more generic `adi_ports_Configure()`

API that supports more robust and dynamic hardware and multiplexing management.

Whether using the legacy `adi_ports_EnableXxx()` API (still supported) or the newer `adi_ports_Configure()` API, the port control manager uses an unambiguous naming convention to safeguard against conflicts with other software libraries provided by Analog Devices or elsewhere. All enumeration values and `typedef` statements use the `ADI_PORTS_` prefix, and functions and global variables use the lowercase `adi_ports_` equivalent.

Both APIs are described in the "Port Control Manager API Reference" section of this chapter. Example usages are provided in the "Using the Port Control Manager" section.

Using the Port Control Manager

Depending on the processor family, one or the other API is used to manage the device and port configurations: `adi_ports_EnableXxx()` for legacy parts (ADSP-BF534/6/7) or `adi_ports_Configure()` for more recent parts. One example for each is illustrated in this section.

Legacy adi_ports_EnableXxx() API Usage

To demonstrate the use of the port control manager, an example is presented that configures the PPI for use with two internal frame syncs.

The port control manager is initialized as follows.

```
adi_ports_Init(     // Initialize Port Control Manager

          NULL      // No special data area for critical
                    // regions required

);
```

To enable the flag pins for the required PPI use, the `adi_ports_EnablePPI` function is called with an array of directives that determine how to configure the register.

```
// Configure the PPI_CONTROL register
ADI_PPI_CONTROL_REG ppi_control;
ppi_control.port_en  = 0;    // Disable until ready
ppi_control.port_dir = 0;    // Receive mode
ppi_control.xfr_type = 3;    // Non ITU-R 656 mode
ppi_control.port_cfg = 1;    // two or three internal frame syncs
ppi_control.dlen     = 7;    // 16 Bits data length
ppi_control.polc     = 0;    // Do not invert PPI_CLK
ppi_control.pols     = 0;    // Do not invert PPI_FS1 & PPI_FS2

// set PPI_COUNT to 1 to sample 2 16-bit words.
u16 ppi_count = 1;
u16 ppi_frame = 1;

ADI_DEV_CMD_VALUE_PAIR PPI_config[] = {
{ ADI_PPI_CMD_SET_CONTROL_REG, (void*)(*(u16*)&ppi_control)   },
{ ADI_PPI_CMD_SET_TRANSFER_COUNT_REG, (void*)(*(u16*)&ppi_count)
},
{ ADI_PPI_CMD_SET_LINES_PER_FRAME_REG,
(void*)(*(u16*)&ppi_frame)},
{ ADI_DEV_CMD_END, 0 }
};

// Program PPI peripheral
adi_dev_Control(
        ppiHandle,
        ADI_DEV_CMD_TABLE,
        (void*)PPI_config );

// other configuration code for PPI, eg Timers, etc  :
// The following would be elsewhere in the client code
// Configure pins for PPI use
u32 ppi_config[] = { ADI_PORTS_DIR_PPI_BASE };
adi_ports_EnablePPI(
        ppi_config,                       // Array of directives
        sizeof(ppi_config)/sizeof(u32),  // Number of directives
        1                                 // Enable
);

// Enable Data Flow
adi_dev_Control(ppiHandle, ADI_DEV_CMD_SET_DATAFLOW, 1);
```

Finally, when the port control manager is no longer required, the service is terminated with a call to `adi_ports_Terminate3`:

```
adi_ports_Terminate();
```

Newer adi_ports_Configure() API Usage

The common `adi_ports_Init()` and `adi_ports_Terminate()` calls are still used. Within this context, the following example illustrates configuration of a similar PPI setup using `adi_ports_Configure()`. (This particular code snippet is from the PPI driver implementation for the ADSP-BF518 Blackfin processor.)

```
static u32 ppiSetPortControl(
   ADI_PPI      *pDevice,
   u32          OpenFlag
)
{

   /* Number of directives to be passed */
   u32 nDirectives;
   /* Return code */
   u32 eResult;
```

```
/* Directives to enable PPI Clock and Data ports */
ADI_PORTS_DIRECTIVE     aePpiClkDataDirectives [] =
{
     ADI_PORTS_DIRECTIVE_PPI_CLK_MUX2,
     ADI_PORTS_DIRECTIVE_PPI_D0,
     ADI_PORTS_DIRECTIVE_PPI_D1,
     ADI_PORTS_DIRECTIVE_PPI_D2,
     ADI_PORTS_DIRECTIVE_PPI_D3,
     ADI_PORTS_DIRECTIVE_PPI_D4,
     ADI_PORTS_DIRECTIVE_PPI_D5,
     ADI_PORTS_DIRECTIVE_PPI_D6,
     ADI_PORTS_DIRECTIVE_PPI_D7,
     ADI_PORTS_DIRECTIVE_PPI_D8,
     ADI_PORTS_DIRECTIVE_PPI_D9,
     ADI_PORTS_DIRECTIVE_PPI_D10,
     ADI_PORTS_DIRECTIVE_PPI_D11,
     ADI_PORTS_DIRECTIVE_PPI_D12,
     ADI_PORTS_DIRECTIVE_PPI_D13,
     ADI_PORTS_DIRECTIVE_PPI_D14,
     ADI_PORTS_DIRECTIVE_PPI_D15,
};

/* Directives to enable PPI Frame sync ports */
ADI_PORTS_DIRECTIVE     aePpiFsDirectives [] =
{
     ADI_PORTS_DIRECTIVE_PPI_FS1_MUX2,
     ADI_PORTS_DIRECTIVE_PPI_FS2_MUX2,
     ADI_PORTS_DIRECTIVE_PPI_FS3
};

/* PPI device needs Clock & Data pins 0 to 7 by default */
 nDirectives = 9;
```

```
/* IF (PPI Data length is more than 8 bits) */
if (pDevice->PPIControl->dlen > 0)
{
     /* Enable rest of the data pins depending on PPI data
          length */
     /* PPI does not support 9-bit data length, so increase
         directive count by data length + 1 */
     nDirectives += (pDevice->PPIControl->dlen + 1);
}

/* Call port control to enable PPI Clock and data pins */
eResult = adi_ports_Configure(aePpiClkDataDirectives,
    nDirectives);

/* IF (Successfully enabled PPI Clock and data pins) */
if (eResult == ADI_PORTS_RESULT_SUCCESS)
{
     /* IF (PPI configured to use Frame syncs) */
     if (pDevice->PPIControl->xfr_type == 3)
     {
        /* assume no FS directives required to be sent */
        nDirectives = 0;

        /* IF (PPI configured in transmit mode) */
        if (pDevice->PPIControl->port_dir)
        {
             /* IF (Use 2 or 3 frame syncs) */
             if ((pDevice->PPIControl->port_cfg == 1) ||
                  (pDevice->PPIControl->port_cfg == 3))
             {
                  /* Enable FS2 & FS3 */
                  nDirectives = 2;
             }
        }
```

```
            /* ELSE (PPI configured in receive mode) */
            else
            {
                /* IF (Use 2 or 3 frame syncs) */
                if ((pDevice->PPIControl->port_cfg == 1) ||
                    (pDevice->PPIControl->port_cfg == 2))
                {
                    /* Enable FS2 & FS3 */
                    nDirectives = 2;/
                }
            }

            /* Call port control to enable PPI FS pins */
            eResult = adi_ports_Configure(aePpiFsDirectives,
                 nDirectives);
          }
      }
      /* return */
      return (eResult);
  }
```

Virtual Devices and Device Indexing

Some processors support mapping the same peripheral device to multiple ports to maximize the combinations of peripheral mapping and facilitate system design. When there is more than a one-to-one mapping of peripherals-to-ports, the port service employs virtual entries in the device table to accommodate the extra configurations.

The term “virtual” can also be thought of as “secondary”. But the important thing to remember is the device number is a zero-based array index that spans all “primary” device mappings plus any “virtual” (or secondary) mappings.

For example, the ADSP-BF526/7 processors have two distinct built-in SPORT devices (SPORT0 and SPORT1). Physical mapping of the SPORT to the appropriate port pins is managed in the SPORT driver code (`adi_sport.c`) via a set of processor-specific pin-mapping defines (`adi_ports_bf52x.h`) such as `ADI_PORTS_DIRECTIVE_SPORT0F_DRPRI`. The driver code selects the appropriate pin-mapping based on the device index, as specified in the `adi_dev_open()` call `DevNumber` parameter.

In the case at hand, SPORT0 and SPORT1 have primary mapping to different pins on PORTF, corresponding to device index 0 and 1. But SPORT0 also offers a secondary mapping via PORTG as device index 2, implying a third virtual device exists, where in fact, only two physical SPORTs exist.

Virtual device table entries (and indexing) are used whenever a peripheral may be mapped to multiple locations. It is an error to map the same physical device to multiple locations simultaneously; only one physical mapping is allowed, regardless of whether it's a primary or a secondary (virtual) mapping. For the ADSP-BF526/7, only two SPORTS are available: SPORT0 may be mapped to PORTF (device index 0) or to PORTG (device index 2), not both, whereas SPORT1 is limited to a singular mapping by device index 1.

Port Control Manager API Reference

This section documents the port control manager service application programming interface (API).

Legacy APIs (`adi_ports_EnableXxx()`) are described in detail in this section. The newer API (`adi_ports_Configure()`) is documented in general in this section, and also in further detail within the respective device driver documents located in the …`/Blackfin/docs` subdirectory.

Notation Conventions

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

Name – 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_ports_Init

Description

The `adi_ports_Init` function initializes the port control manager.

This API is common to all Blackfin processors, including both legacy processors and all newer processors.

Prototype

```
ADI_PORTS_RESULT adi_ports_Init(
        void                   *pCriticalRegionArg
);
```

Arguments

`pEnterCriticalArg`	Handle to a user-defined data area to store critical region data. This is passed to `adi_int_EnterCriticalRegion` where it is used internally by the module to protect against multiple access during critical port control register manipulations which must be atomic. See "Interrupt Manager" on page 2-1 for further details.

Return Value

`ADI_PORTS_RESULT_SUCCESS`	Port control manager was successfully initialized.

adi_ports_Terminate

Description

The `adi_ports_Terminate` function terminates the port control manager.

ⓘ This API is common to all Blackfin processors, including both legacy processors and all newer processors.

Prototype

```
ADI_PORTS_RESULT adi_ports_Terminate(void);
```

Arguments

None

Return Value

`ADI_PORTS_RESULT_SUCCESS`	Function completed successfully.

adi_ports_Configure

Description

The `adi_ports_Configure` function is used to perform the appropriate pin multiplexing to configure the GPIO pins for GPIO or peripheral use. The function accepts a table of port control directives. Directives are specific to each processor family and are defined in the include file named for that processor family, for example, `adi_ports_bf2x.h` found in the VisualDSP++ include path `\include\services\ports\`.

Prototype

```
ADI_PORTS_RESULT adi_ports_Configure(
   ADI_PORTS_DIRECTIVE     *pDirectives,
   u32                     nDirectives,
);
```

Arguments

`ADI_PORTS_DIRECTIVE`	Pointer to an array of directives
`u32`	Number of directives

Return Value

`ADI_PORTS_RESULT_SUCCESS`	Function completed successfully.
`ADI_PORTS_RESULT_FAILED`	Function failed.
`ADI_PORTS_RESULT_BAD_DIRECTIVE`	Invalid directive value has been passed.
`ADI_PORTS_RESULT_NULL_ARRAY`	Address of the `Directives` array is NULL.

adi_ports_EnablePPI

Description

(*Legacy function – use on ADSP-BF534/6/7 processors only.*) The `adi_ports_EnablePPI` function configures the port control registers to enable the use of the required PPI channel.

Prototype

```
ADI_PORTS_RESULT adi_ports_EnablePPI(
        u32          *Directives,
        u32          nDirectives,
        u32          Enable
);
```

Arguments

`Directives`	Address of an array of directives describing how the to configure the PPI flags. See "Legacy API Enumeration Values" on page 9-25.
`nDirectives`	Number of entries in `Directives` array
`Enable`	Flag determining whether the functionality is enabled (1) or disabled (0)

Return Value

`ADI_PORTS_RESULT_SUCCESS`	Function completed successfully.
`ADI_PORTS_RESULT_NULL_ARRAY`	Address of the `Directives` array is NULL.
`ADI_PORTS_RESULT_BAD_DIRECTIVE`	Invalid directive value has been passed.

adi_ports_EnableSPI

Description

(*Legacy function – use on ADSP-BF534/6/7 processors only.*) The `adi_ports_EnableSPI` function configures the port control registers to enable the use of the required SPI channel.

Prototype

```
ADI_PORTS_RESULT adi_ports_EnableSPI(
        u32          *Directives,
        u32          nDirectives,
        u32          Enable
);
```

Arguments

`Directives`	Address of an array of directives describing how to configure the SPI flags. See "Legacy API Enumeration Values" on page 9-25.
`nDirectives`	Number of entries in `Directives` array
`Enable`	Flag determining whether the functionality is enabled (1) or disabled (0)

Return Value

`ADI_PORTS_RESULT_SUCCESS`	Function completed successfully.
`ADI_PORTS_RESULT_NULL_ARRAY`	Address of the `Directives` array is NULL.
`ADI_PORTS_RESULT_BAD_DIRECTIVE`	Invalid directive value has been passed.

adi_ports_EnableSPORT

Description

(*Legacy function – use on ADSP-BF534/6/7 processors only.*) The `adi_ports_EnableSPORT` function configures the port control registers to enable the use of the required SPORT channel.

Prototype

```
ADI_PORTS_RESULT adi_ports_EnableSPORT(
        u32          *Directives,
        u32          nDirectives,
        u32          Enable
);
```

Arguments

`Directives`	Address of an array of directives describing how the to configure the SPORT flags. See "Legacy API Enumeration Values" on page 9-25.
`nDirectives`	Number of entries in `Directives` array
`Enable`	Flag determining whether the functionality is enabled (1) or disabled (0)

Return Value

`ADI_PORTS_RESULT_SUCCESS`	Function completed successfully.
`ADI_PORTS_RESULT_NULL_ARRAY`	Address of the `Directives` array is NULL.
`ADI_PORTS_RESULT_BAD_DIRECTIVE`	Invalid directive value has been passed.

adi_ports_EnableUART

Description

(*Legacy function – use on ADSP-BF534/6/7 processors only.*) The `adi_ports_EnableUART` function configures the port control registers to enable the use of the required UART channel.

Prototype

```
ADI_PORTS_RESULT adi_ports_EnableUART(
        u32          *Directives,
        u32          nDirectives,
        u32          Enable
);
```

Arguments

`Directives`	Address of an array of directives describing how the to configure the UART flags. See "Legacy API Enumeration Values" on page 9-25.
`nDirectives`	Number of entries in `Directives` array
`Enable`	Flag determining whether the functionality is enabled (1) or disabled (0)

Return Value

`ADI_PORTS_RESULT_SUCCESS`	Function completed successfully.
`ADI_PORTS_RESULT_NULL_ARRAY`	Address of the `Directives` array is NULL.
`ADI_PORTS_RESULT_BAD_DIRECTIVE`	Invalid directive value has been passed.

adi_ports_EnableCAN

Description

(*Legacy function – use on ADSP-BF534/6/7 processors only.*) The `adi_ports_EnableCAN` function configures the port control registers to enable the use of the required CAN channel.

Prototype

```
ADI_PORTS_RESULT adi_ports_EnableCAN(
        u32          *Directives,
        u32          nDirectives,
        u32          Enable
);
```

Arguments

`Directives`	Address of an array of directives describing how the to configure the CAN flags. See "Legacy API Enumeration Values" on page 9-25.
`nDirectives`	Number of entries in `Directives` array
`Enable`	Flag determining whether the functionality is enabled (1) or disabled (0)

Return Value

`ADI_PORTS_RESULT_SUCCESS`	Function completed successfully.
`ADI_PORTS_RESULT_NULL_ARRAY`	Address of the `Directives` array is NULL.
`ADI_PORTS_RESULT_BAD_DIRECTIVE`	Invalid directive value has been passed.

adi_ports_EnableTimer

Description

(*Legacy function – use on ADSP-BF534/6/7 processors only.*) The `adi_ports_EnableTimer` function configures the port control registers to enable the appropriate flag pins for the output of general-purpose (GP) timer clock signals, the timer clock input (used mainly for PPI clock), alternate timer clock inputs[1], and for bit rate detection on CAN and UART inputs.[2]

Any number of pins as required can be assigned in the one call to `adi_ports_EnableTimer`.

Prototype

```
ADI_PORTS_RESULT adi_ports_EnableTimer(
        u32          *Directives,
        u32          nDirectives,
        u32          Enable
);
```

Arguments

`Directives`	Address of an array of directives describing the timers for which the flags are configured. See "Legacy API Enumeration Values" on page 9-25.
`nDirectives`	Number of entries in `Directives` array
`Enable`	Flag determining whether the functionality is enabled (1) or disabled (0)

[1] Alternatively, the `TACLKx` flag pins can provide the clock signal to the general-purpose timers in `PWM_OUT` mode. For details, see "Timer Service" on page 8-1.

[2] Timers must be configured for `WDTH_CAP` mode. For details, see "Timer Service" on page 8-1.

Return Value

`ADI_PORTS_RESULT_SUCCESS`	Function completed successfully.
`ADI_PORTS_RESULT_NULL_ARRAY`	Address of the `Directives` array is NULL.
`ADI_PORTS_RESULT_BAD_DIRECTIVE`	Invalid directive value has been passed.
`ADI_PORTS_RESULT_PIN_ALREADY_IN_USE`	One of the required pins has already been assigned a different functionality.

adi_ports_EnableGPIO

Description

(*Legacy function – use on ADSP-BF534/6/7 processors only.*) The `adi_ports_EnableGPIO()` function configures the port control registers to enable any number of flag pins for GPIO use.

By default, GPIO use is enabled upon system reset.

Prototype

```
ADI_PORTS_RESULT adi_ports_EnableGPIO(
        u32          *Directives,
        u32          nDirectives,
        u32          Enable
);
```

Arguments

`Directives`	Address of an array of directives describing which pins are configured for GPIO use. See "Legacy API Enumeration Values" on page 9-25.
`nDirectives`	Number of entries in `Directives` array
`Enable`	Flag determining whether the functionality is enabled (1) or disabled (0)

Return Value

`ADI_PORTS_RESULT_SUCCESS`	Function completed successfully.
`ADI_PORTS_RESULT_NULL_ARRAY`	Address of the `Directives` array is NULL.
`ADI_PORTS_RESULT_BAD_DIRECTIVE`	Invalid directive value has been passed.
`ADI_PORTS_RESULT_PIN_ALREADY_IN_USE`	One of the required pins has already been assigned a different functionality.

adi_ports_ClearProfile

Description

(*Legacy function – use on ADSP-BF534/6/7 processors only.*) The `adi_ports_ClearProfile()` function clears the specified profile structure passed.

Prototype

```
ADI_PORTS_RESULT adi_ports_ClearProfile(
        ADI_PORTS_PROFILE     *profile
);
```

Arguments

`Profile`	Data structure containing the profile to clear

Return Value

`ADI_PORTS_RESULT_SUCCESS`	Function completed successfully.
`ADI_PORTS_RESULT_FAILED`	Profile structure is not a valid address.

adi_ports_GetProfile

Description

(*Legacy function – use on ADSP-BF534/6/7 processors only.*) The `adi_ports_GetProfile()` function retrieves the profile information into the given data structure.

Prototype

```
ADI_PORTS_RESULT adi_ports_GetProfile(
        ADI_PORTS_PROFILE     *profile
);
```

Arguments

`Profile`	`ADI_PORTS_PROFILE` data structure

Return Value

`ADI_PORTS_RESULT_SUCCESS`	Function completed successfully.
`ADI_PORTS_RESULT_FAILED`	Profile structure is not a valid address.

adi_ports_SetProfile

Description

(*Legacy function – use on ADSP-BF534/6/7 processors only.*) The `adi_ports_SetProfile()` function applies the profile specified by the values in the given data structure.

Prototype

```
ADI_PORTS_RESULT adi_ports_SetProfile(
        ADI_PORTS_PROFILE     *profile
);
```

Arguments

`Profile`	`ADI_PORTS_PROFILE` data structure containing the given profile

Return Value

`ADI_PORTS_RESULT_SUCCESS`	Function completed successfully.
`ADI_PORTS_RESULT_FAILED`	Profile structure is not a valid address.

Public Data Types, Enumerations, and Macros

This section defines the legacy public data structures and enumerations used by the port control manager service `adi_ports_EnableXxx()` API.

Always check the include file for the port control manager, `adi_ports.h`, or the respective device driver include file (for newer processors) for the most up-to-date information.

Newer APIs (`adi_ports_Configure()`) are documented in the respective device driver documentation located in the …`/Blackfin/docs` subdirectory.

ADI_PORTS_RESULT

These values have been defined in the context of the relevant function call. The complete list is found in Table 9-1.

Table 9-1. ADI_PORTS_RESULT Values

Result Code	Numerical Value	Description
`ADI_PORTS_RESULT_SUCCESS`	0	Function executed correctly.
`ADI_PORTS_RESULT_FAILED`	1	Function execution not completed.
`ADI_PORTS_RESULT_BAD_ DIRECTIVE`	0x90001	Invalid directive value has been passed.
`ADI_PORTS_RESULT_NULL_ARRAY`	0x90002	Address of the `Directives` array is NULL.

Legacy API Enumeration Values

The `adi_ports_EnableXxx()` legacy API directives are described by anonymous enumeration types as shown in Table 9-2.

Table 9-2. Port Control Manager Enumeration Types

Enumeration Type	Description
PPI Operation	
`ADI_PORTS_DIR_PPI_BASE`	Enables flag pins for basic PPI operation.
`ADI_PORTS_DIR_PPI_FS3`	Enables flag pin for 3rd PPI frame sync.
SPI Operation	
`ADI_PORTS_DIR_SPI_BASE`	Enables flag pins for basic PPI operation.
`ADI_PORTS_DIR_SPI_SLAVE_SELECT_1`	Enables flag pins for SPI SlaveSelect 1.

Table 9-2. Port Control Manager Enumeration Types (Cont'd)

Enumeration Type	Description
ADI_PORTS_DIR_SPI_SLAVE_SELECT_2	Enables flag pins for SPI SlaveSelect 2.
ADI_PORTS_DIR_SPI_SLAVE_SELECT_3	Enables flag pins for SPI SlaveSelect 3.
ADI_PORTS_DIR_SPI_SLAVE_SELECT_4	Enables flag pins for SPI SlaveSelect 4.
ADI_PORTS_DIR_SPI_SLAVE_SELECT_5	Enables flag pins for SPI SlaveSelect 5.
ADI_PORTS_DIR_SPI_SLAVE_SELECT_6	Enables flag pins for SPI SlaveSelect 6.
ADI_PORTS_DIR_SPI_SLAVE_SELECT_7	Enables flag pins for SPI SlaveSelect 7.
SPORT Operation	
ADI_PORTS_DIR_SPORT0_BASE_RX	Enables flag pins for basic SPORT receive operation.
ADI_PORTS_DIR_SPORT0_BASE_TX	Enables flag pins for basic SPORT transmit operation.
ADI_PORTS_DIR_SPORT0_RXSE	Enables flag pin for SPORT secondary data receive.
ADI_PORTS_DIR_SPORT0_TXSE	Enables flag pin for SPORT secondary data transmit.
ADI_PORTS_DIR_SPORT1_BASE_RX	Enables flag pins for basic SPORT receive operation.
ADI_PORTS_DIR_SPORT1_BASE_TX	Enables flag pins for basic SPORT transmit operation.
ADI_PORTS_DIR_SPORT1_RXSE	Enables flag pin for SPORT secondary data receive.
ADI_PORTS_DIR_SPORT1_TXSE	Enables flag pin for SPORT secondary data transmit.

Table 9-2. Port Control Manager Enumeration Types (Cont'd)

Enumeration Type	Description
UART Operation	
ADI_PORTS_DIR_UART0_RX	Enables flag pins for basic UART receive operation.
ADI_PORTS_DIR_UART0_TX	Enables flag pins for basic UART transmit operation.
ADI_PORTS_DIR_UART1_RX	Enables flag pins for basic UART receive operation.
ADI_PORTS_DIR_UART1_TX	Enables flag pins for basic UART transmit operation.
CAN Operation	
ADI_PORTS_DIR_CAN_RX	Enables flag pins for basic CAN receive operation.
ADI_PORTS_DIR_CAN_TX	Enables flag pins for basic CAN transmit operation.
Timer Operation	
ADI_PORTS_DIR_TMR_CLK	Enables flag pin for Timer Input Clock use.
ADI_PORTS_DIR_TMR_0	Enables flag pin for GP Timer 0 use.
ADI_PORTS_DIR_TMR_1	Enables flag pin for GP Timer 1 use.
ADI_PORTS_DIR_TMR_2	Enables flag pin for GP Timer 2 use.
ADI_PORTS_DIR_TMR_3	Enables flag pin for GP Timer 3 use.
ADI_PORTS_DIR_TMR_4	Enables flag pin for GP Timer 4 use.
ADI_PORTS_DIR_TMR_5	Enables flag pin for GP Timer 5 use.
ADI_PORTS_DIR_TMR_6	Enables flag pin for GP Timer 6 use.
ADI_PORTS_DIR_TMR_7	Enables flag pin for GP Timer 7 use.
GPIO Operation	
ADI_PORTS_DIR_GPIO_PF0	Enables PF0 pin for GPIO use.
ADI_PORTS_DIR_GPIO_PF1	Enables PF1 pin for GPIO use.
ADI_PORTS_DIR_GPIO_PF2	Enables PF2 pin for GPIO use.
ADI_PORTS_DIR_GPIO_PF3	Enables PF3 pin for GPIO use.
ADI_PORTS_DIR_GPIO_PF4	Enables PF4 pin for GPIO use.
ADI_PORTS_DIR_GPIO_PF5	Enables PF5 pin for GPIO use.
ADI_PORTS_DIR_GPIO_PF6	Enables PF6 pin for GPIO use.

Table 9-2. Port Control Manager Enumeration Types (Cont'd)

Enumeration Type	Description
ADI_PORTS_DIR_GPIO_PF7	Enables PF7 pin for GPIO use.
ADI_PORTS_DIR_GPIO_PF8	Enables PF8 pin for GPIO use.
ADI_PORTS_DIR_GPIO_PF9	Enables PF9 pin for GPIO use.
ADI_PORTS_DIR_GPIO_PF10	Enables PF10 pin for GPIO use.
ADI_PORTS_DIR_GPIO_PF11	Enables PF11 pin for GPIO use.
ADI_PORTS_DIR_GPIO_PF12	Enables PF12 pin for GPIO use.
ADI_PORTS_DIR_GPIO_PF13	Enables PF13 pin for GPIO use.
ADI_PORTS_DIR_GPIO_PF14	Enables PF14 pin for GPIO use.
ADI_PORTS_DIR_GPIO_PF15	Enables PF15 pin for GPIO use.
ADI_PORTS_DIR_GPIO_PG0	Enables PG0 pin for GPIO use.
ADI_PORTS_DIR_GPIO_PG1	Enables PG1 pin for GPIO use.
ADI_PORTS_DIR_GPIO_PG2	Enables PG2 pin for GPIO use.
ADI_PORTS_DIR_GPIO_PG3	Enables PG3 pin for GPIO use.
ADI_PORTS_DIR_GPIO_PG4	Enables PG4 pin for GPIO use.
ADI_PORTS_DIR_GPIO_PG5	Enables PG5 pin for GPIO use.
ADI_PORTS_DIR_GPIO_PG6	Enables PG6 pin for GPIO use.
ADI_PORTS_DIR_GPIO_PG7	Enables PG7 pin for GPIO use.
ADI_PORTS_DIR_GPIO_PG8	Enables PG8 pin for GPIO use.
ADI_PORTS_DIR_GPIO_PG9	Enables PG9 pin for GPIO use.
ADI_PORTS_DIR_GPIO_PG10	Enables PG10 pin for GPIO use.
ADI_PORTS_DIR_GPIO_PG11	Enables PG11 pin for GPIO use.
ADI_PORTS_DIR_GPIO_PG12	Enables PG12 pin for GPIO use.
ADI_PORTS_DIR_GPIO_PG13	Enables PG13 pin for GPIO use.
ADI_PORTS_DIR_GPIO_PG14	Enables PG14 pin for GPIO use.
ADI_PORTS_DIR_GPIO_PG15	Enables PG15 pin for GPIO use.
ADI_PORTS_DIR_GPIO_PH0	Enables PH0 pin for GPIO use.

Table 9-2. Port Control Manager Enumeration Types (Cont'd)

Enumeration Type	Description
`ADI_PORTS_DIR_GPIO_PH1`	Enables PH1 pin for GPIO use.
`ADI_PORTS_DIR_GPIO_PH2`	Enables PH2 pin for GPIO use.
`ADI_PORTS_DIR_GPIO_PH3`	Enables PH3 pin for GPIO use.
`ADI_PORTS_DIR_GPIO_PH4`	Enables PH4 pin for GPIO use.
`ADI_PORTS_DIR_GPIO_PH5`	Enables PH5 pin for GPIO use.
`ADI_PORTS_DIR_GPIO_PH6`	Enables PH6 pin for GPIO use.
`ADI_PORTS_DIR_GPIO_PH7`	Enables PH7 pin for GPIO use.
`ADI_PORTS_DIR_GPIO_PH8`	Enables PH8 pin for GPIO use.
`ADI_PORTS_DIR_GPIO_PH9`	Enables PH9 pin for GPIO use.
`ADI_PORTS_DIR_GPIO_PH10`	Enables PH10 pin for GPIO use.
`ADI_PORTS_DIR_GPIO_PH11`	Enables PH11 pin for GPIO use.
`ADI_PORTS_DIR_GPIO_PH12`	Enables PH12 pin for GPIO use.
`ADI_PORTS_DIR_GPIO_PH13`	Enables PH13 pin for GPIO use.
`ADI_PORTS_DIR_GPIO_PH14`	Enables PH14 pin for GPIO use.
`ADI_PORTS_DIR_GPIO_PH15`	Enables PH15 pin for GPIO use.

10 DEVICE DRIVER MANAGER

This chapter describes the Analog Devices device driver model. Use the device driver model to control devices, both internal and external, to Analog Devices 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 the following sections:

- "Device Driver Model Overview" on page 10-3
  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 10-6
  Describes how applications invoke and interact with the device driver model including an explanation of the different dataflow methods supported in the model.

- "Device Manager Design" on page 10-30
  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 10-45
  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 10-56
  Describes the API functions of the device driver manager.
- “Device Manager Public Data Types and Enumerations” on page 10-66
  Provides tables for all the device manager data structures and enumerations.
- “Physical Driver API Reference” on page 10-80
  Describes the API used between the device driver manager and each physical driver.
- “Examples” on page 10-87
  Provides PPI driver and UART driver code examples.

ⓘ For detailed information regarding the use of specific device drivers, including command IDs, event codes, return codes, example code, and so on, refer to the individual device driver documents located in the `Blackfin/docs/drivers` directory of your VisualDSP++ installation.

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 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 works 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.

The terms *device manager* and *physical driver* refer to the respective software components. 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 10-1 illustrates the various components of the system design.

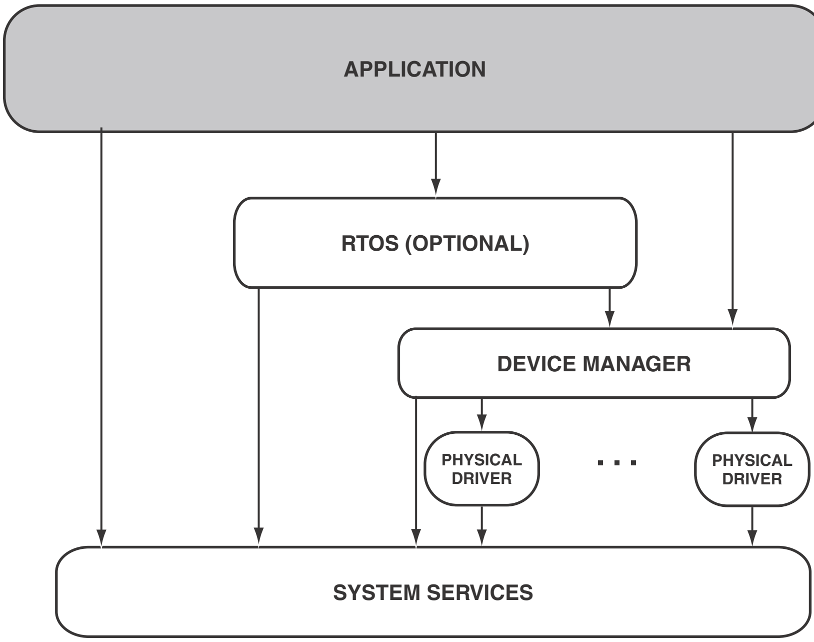

Figure 10-1. System Design and Hierarchy

The components shown in the figure 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 real-time operating systems 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 calls 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 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.

Device Manager Overview

The device manager API consists of these 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 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 is processed) and another device that moves the processed data out of the system. Often, multiple devices run 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 the underlying implementation details of 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:

- One-dimensional buffers called `ADI_DEV_1D_BUFFER`
- Two-dimensional buffers called `ADI_DEV_2D_BUFFER`
- Circular (autobuffer type) buffers called `ADI_DEV_CIRCULAR_BUFFER`

Because the physical movement of data uses valuable computing resources and has very little benefit, only the pointers to buffers are typically 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 a circular buffer. Though each of these types of buffers is processed differently, where no significant difference in processing exists, 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 and then made available again to the application. The `adi_dev_Write` function provides the device with buffers that contain data sent out through the device. The `adi_dev_Read` function provides the device with buffers, which are filled with data that is inbound from the device.

Buffers are processed in the order in which they are received. Buffers provided to a given device need not 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 more complex (see “Providing Buffers to a Device” on page 10-14).

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 and 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 the device manager; the more memory provided, the more physical devices can be simultaneously opened.

The device manager requires a contiguous block of memory that can be thought of two parts—one part is base memory required for the device manager to instantiate itself, and the other part is memory that is required to support n number of simultaneously opened device drivers. Macros (`ADI_DEV_BASE_MEMORY` and `ADI_DEV_DEVICE_MEMORY`) are provided to define the amount of memory (in bytes) required for the base memory and incremental device driver memory, respectively.

For example, if the client wants 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 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_EnterCriticalRegion()` function. Refer to "adi_int_Init" on page 2-19.

Device Manager Termination

When the device manager is no longer needed, the client can terminate it by means of 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 client can reuse the memory that was supplied to the device manager via the `adi_dev_Init()` function. Once terminated, the device manager must be reinitialized in order to be used again.

ⓘ 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 the device driver it wants to open (the `pEntryPoint` parameter), the instance of the device it wants to open (the `DevNumber` parameter), the direction it wants data to flow (inbound, outbound, or both), and so on. The client also passes in the handle to the DMA service the device manager and physical drivers to 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. They are also callback-related. The `DCBHandle` parameter is a handle to the deferred callback service that is used by the device driver 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 `DCBHandle` is NULL, all callbacks are live; meaning 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 completing the processing of a buffer, and some events may be unexpected, such as a 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.

Dataflow through a device does not start with the `adi_dev_Open()` function. This function simply opens the device for use; 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, little or no application configuration is required. At other times, the default settings may not be appropriate for an application, so the device needs some configuration. 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. This function takes as parameters the `DeviceHandle` (described in "Opening a Device"), a command ID that identifies the parameter to set or detect, and a pointer to the memory location that contains the value of the parameter to set (or where the value of the parameter 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 the physical driver documentation to determine which parameters are configurable and the configuration choices.

Regardless of whether the client needs to make configuration changes, the client must 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 inform the device manager of the dataflow method to use by calling the `adi_dev_Control()`

function with the `ADI_DEV_CMD_SET_DATAFLOW_METHOD` command. After the dataflow method has been defined, the client can provide inbound buffers (via the `adi_dev_Read` function) or outbound buffers (via the `adi_devWrite` function) to the device.

As shown in Figure 10-2, 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.

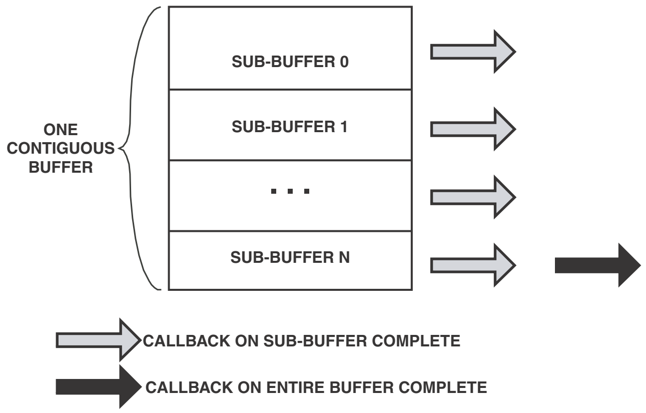
ONE CONTIGUOUS BUFFER
SUB-BUFFER 0
SUB-BUFFER 1
. . .
SUB-BUFFER N
CALLBACK ON SUB-BUFFER COMPLETE
CALLBACK ON ENTIRE BUFFER COMPLETE

Figure 10-2. Circular Buffer Operation

When providing the device manager with the circular buffer, the application informs the device manager of 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, callback after processing each sub-buffer, and callback after processing the entire buffer. The device manager begins processing at the start of the buffer. When 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 10-3, 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 buffer, 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.

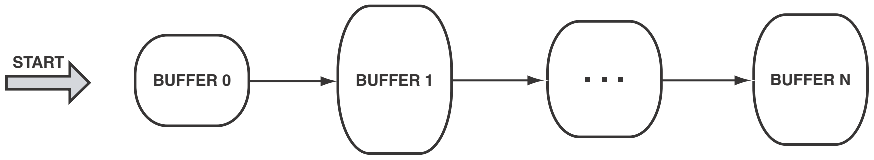

Figure 10-3. Chained Buffers

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 also supports synchronous operation.

As shown in Figure 10-4, the chained with loopback method is similar to the chained dataflow method except that after processing the last buffer, the device manager automatically loops back to the 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, allow the device manager to process the buffers, and never have to resupply the device manager with additional buffers. As with the chained dataflow method, each buffer, any one buffer, none, or all buffers can be tagged to generate a callback to the application when they are processed.

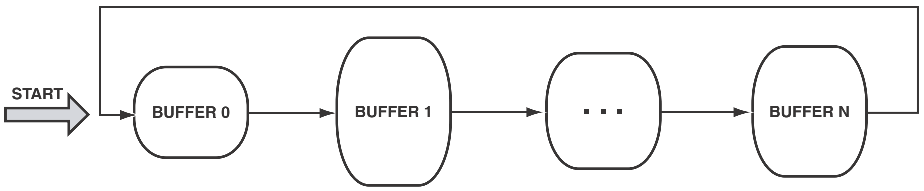

Figure 10-4. Chained Buffers With Loopback

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 enables 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 are opened for inbound or bidirectional data are 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 chosen.

When a device is 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 provided buffer points to a contiguous piece of memory corresponding to however many sub-buffers the application wants to use.

For example, assume 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 use 512 bytes of the buffer while the application uses the other 512 bytes simultaneously.

Another example is an application that wants to process a standard NTSC video frame (525 lines with 1716 bytes per line). The data buffer 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 is 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 is 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 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 be provided while dataflow is disabled.

Closing a Device

When the device is no longer needed by the client, the device is 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 used by the device driver. Should the application need to reuse the device after it is closed, it can be reopened via the `adi_dev_Open` function.

Callbacks

The device manager calls the application's callback function to notify the client of occurring events. Events include expected events (such as completion of buffer processing) 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 system services, it 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 used). As such, it is good practice to initialize and open the system services before opening any device drivers. See "Initialization" on page 1-11 and "Termination" on page 1-27 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 Lite board contains an AD1836 audio codec. This 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 is developed in which the AD1836 codec is the only device that would ever be connected to the processor, a single physical device driver could be written to control and manage the SPI and the SPORT.

Alternatively, the SPI port could hierarchically sit above the SPI and SPORT drivers, making calls into those physical drivers as necessary. It is especially true if other peripherals are to share the SPI port (for example, separate SPI and SPORT drivers could be controlled by an AD1836 driver). In this stackable fashion, it is possible to create more complex drivers such as the AD1836 driver or a TCP/IP stack driver that sit on top of an Ethernet controller.

Deciding on a Dataflow Method

When using device drivers, you should give thoughtful consideration when choosing the dataflow method to be used for each device driver. Some types of data are better suited to one type of dataflow method, and other types of data may be more suitable for another dataflow method. As a rough guideline, you may want to consider the following situations when selecting a dataflow method to use for a device driver.

Chained Without Loopback

The chained-without-loopback dataflow method is suitable for packet-based data that may be sent and/or received in a non-continuous or “bursty” manner. For example, UART data from a terminal, Ethernet data, and USB traffic frequently use the chained-without-loopback dataflow method.

Chained With Loopback

The chained-with-loopback dataflow method is suitable for steady, continuous dataflow. For example, streaming audio and video applications frequently use the chained-with-loopback dataflow method. With this method, buffers are provided at initialization time. Because loopback is used, the application never needs to re-supply the driver with additional buffers, since the driver continually loops through the same set of buffers.

Circular

When using streaming audio or video, the streaming sub-mode is highly recommended to avoid clicks and pops with audio data and glitches on video data.

The circular dataflow method is suitable for steady, continuous dataflow, where the entire data fits in a 64K byte maximum contiguous block of memory. Streaming audio data, assuming it fits within the 64 K byte block of memory, is sometimes appropriate for the circular dataflow method. This saves the overhead of having to create multiple buffers as is necessary using the chained-with-loopback method. Generally, video data is not appropriate for the circular dataflow method, since video data is frequently larger than the 64K byte maximum for circular dataflow.

Sequential With and Without Loopback

The sequential dataflow method, with and without loopback, is suitable only for devices that employ half-duplex, serial type communication protocols. For example, the two-wire interface (TWI) device driver uses the sequential dataflow methods to precisely schedule reads and writes in a specific order.

Creating One-Dimensional Buffers

The ADI_DEV_1D_BUFFER data structure is used to describe a linear array of data that a device driver processes. Applications populate the various fields of the buffer to completely describe the data to the device driver. For one-dimensional buffers, applications populate the following fields of the ADI_DEV_1D_BUFFER structure:

- Data – If the buffer is provided to the adi_dev_Write() function, this field contains the starting address of the data sent out through the device. If the buffer is provided to the adi_dev_Read() function, this field contains the starting address of where the device driver stores data received in from the device.
- ElementCount – This field indicates the number of elements pointed to by the data pointer.
- ElementWidth – This field indicates the width (in bytes) of each element sent out or read in.
- CallbackParameter – If this field is NULL, the device driver does not call back the application after the device driver processes the buffer. If non-NULL, the device driver invokes the application's callback function after the buffer is processed by the device driver, passing this value as the third parameter to the callback function.
- pNext – This field points to the next one-dimensional buffer in the chain, if any. If NULL, the given buffer is the only buffer provided to the adi_dev_Read() or adi_dev_Write() function.
  If non-NULL, this field contains the address of the next one-dimensional buffer in the chain of buffers passed to the adi_dev_Read() or adi_dev_Write() function.

- `pAdditionalInfo` – This field is a device driver-dependent value. This field is not used for most device drivers. For information describing whether this field is used by the particular device driver, refer to the documentation specific to the device driver, located in the `…/Blackfin/docs` subdirectory.
- `ProcessedFlag` – Some device drivers set this value to TRUE after processing the buffer. For information describing whether this field is used by the particular device driver, refer to the documentation specific to the device driver, located in the `…/Blackfin/docs` subdirectory.

When buffers are submitted to the device driver via the `adi_dev_Read()` or `adi_dev_Write()` functions, some device drivers do not require the following fields be populated:

- `ProcessedFlag` – Some device drivers set this value to TRUE after the device driver processes the buffer. For information describing whether this field is used by the particular device driver, refer to the documentation specific to the device driver, located in the `…/Blackfin/docs` subdirectory.
- `ProcessedElementCount` – Some device drivers set this value to the number of elements processed by the driver for the given buffer. For example, if a networking driver submitted a buffer describing 100 bytes of data to the `adi_dev_Read()` function for an incoming data packet that contains only 75 bytes of data, the driver may set this value to 75. This would indicate that although 100 bytes was requested, only 75 bytes were available. For information describing whether this field is used by the particular device driver, refer to the documentation specific to the device driver, located in the `…/Blackfin/docs` subdirectory.

For example, the following code fragment prepares and submits a single buffer of 128 16-bit elements to the `adi_dev_Read()` function. The driver calls back the application when the buffer has been processed, passing the value 0x12345678 as a parameter to the callback function.

```
#define SAMPLES_PER_BUFFER  (128)     // number of samples
                                      // in a data buffer

static u16 Data[SAMPLES_PER_BUFFER];  // storage for data
static ADI_DEV_1D_BUFFER Buffer;      // the actual buffer

            // create buffer for the driver to process
Buffer.Data              = Data;
Buffer.ElementCount      = SAMPLES_PER_BUFFER;
Buffer.ElementWidth      = 2;
Buffer.CallbackParameter = (void *)0x12345678;  // callback,
                                                // pArg = 0x12345678
Buffer.pNext              = NULL;  // only buffer in the chain

       // give the buffer to the driver to fill with data
       Result = adi_dev_Read(Handle, ADI_DEV_1D, (ADI_DEV_BUFFER
*)&Buffer);
```

The following code fragment prepares and submits a chain of four buffers, each describing 128 elements of 32-bit data, to the `adi_dev_Read()` function. The driver calls back the application after each buffer has been processed, passing the address of the buffer that just completed as a parameter to the callback function.

```
#define NUM_BUFFERS          (4)    // number of buffers to use
#define SAMPLES_PER_BUFFER (128)   // # of samples in data buffer
static u32 Data[NUM_BUFFERS*SAMPLES_PER_BUFFER];
                                     // storage for data
static ADI_DEV_1D_BUFFER   Buffer[NUM_BUFFERS];
                                     // the actual buffers
```

```
u32 i; // counter
     // create buffers for the driver to process
     for (i = 0; i < NUM_BUFFERS; i++) {
        Buffer[i].Data = &Data[i * SAMPLES_PER_BUFFER;]
        Buffer[i].ElementCoun        = SAMPLES_PER_BUFFER;
        Buffer[i].ElementWid         = 4;
        Buffer[i].CallbackParameter = &Buffer[i]; // gen call-
back, pArg = buffer address
        Buffer[i].pNext             = &Buffer[i+1];
                          // point to the next in chain
    }
     Buffer[NUM_BUFFERS - 1].pNext  = NULL;
                          // terminate the chain of buffers

    // give the buffers to the driver to fill with data
    Result = adi_dev_Read(Handle, ADI_DEV_1D, (ADI_DEV_BUFFER
*)Buffer);
```

Creating Two-Dimensional Buffers

Use the ADI_DEV_2D_BUFFER data structure to describe a two-dimensional array of data that a device driver processes. Applications populate the various fields of the buffer to completely describe the data to the device driver. For two-dimensional buffers, applications populate the following fields of the ADI_DEV_2D_BUFFER structure:

- Data – If the buffer is provided to the adi_dev_Write() function, this field contains the starting address of the data sent out through the device. If the buffer is provided to the adi_dev_Read() function, this field contains the starting address of where the device driver stores data received from the device.

- ElementWidth – This field indicates the width (in bytes) of each element sent out or read in.

- `XCount` – This field specifies the number of column elements.
- `XModify` – This field specifies the byte address increment (stride) after each column transfer.
- `YCount` – This field specifies the number of row elements.
- `YModify` – This field specifies the byte address increment (stride) after each row transfer.
- `CallbackParameter` – If this field is NULL, the device driver does not call back the application after the buffer is processed by the device driver. If non-NULL, the device driver invokes the application's callback function after the buffer is processed by the device driver, passing this value as the third parameter to the callback function.
- `pNext` – This field points to the next two-dimensional buffer in the chain, if any. If NULL, the given buffer is the only buffer provided to the `adi_dev_Read()` or `adi_dev_Write()` function.
  If non-NULL, this field contains the address of the next two-dimensional buffer in the chain of buffer passed to the `adi_dev_Read()` or `adi_dev_Write()` function.
- `pAdditionalInfo` – This field is a device driver-dependent value. This field is not used for most device drivers. For information describing whether this field is used by the particular device driver, refer to the documentation specific to the device driver, located in the `…/Blackfin/docs` subdirectory.

When buffers are submitted to the device driver via the `adi_dev_Read()` or `adi_dev_Write()` functions, some device drivers do not require the following fields be populated:

- `ProcessedFlag` – Some device drivers set this value to TRUE after the device driver processes the buffer. For information describing whether this field is used by the particular device driver, refer to the documentation specific to the device driver, located in the `…/Blackfin/docs` subdirectory.
- `ProcessedElementCount` – Some device drivers set this value to the number of elements processed by the driver for the given buffer. For example, if a networking driver submitted a buffer describing 100 bytes of data to the `adi_dev_Read()` function for an incoming data packet that contains only 75 bytes of data, the driver may set this value to 75. This would indicate that although 100 bytes was requested, only 75 bytes were available. For information describing whether this field is used by the particular device driver, refer to the documentation specific to the device driver, located in the `…/Blackfin/docs` subdirectory.

For example, the following code fragment prepares and submits a pair of two-dimensional buffers to the `adi_dev_Write()` function for transmission out through the driver. Each buffer describes an NTSC ITU-656 frame of data. Each frame consists of 525 rows, each containing 1716 bytes of data. The driver calls back the application when each buffer has been processed, passing the address of the buffer that just completed as a parameter to the callback function.

```
#define NUM_BUFFERS (2)
#define NUM_FRAMES  (2)
#define COLUMNS     (1716)
#define ROWS        (525)

static u8 Frames[NUM_FRAMES][COLUMNS * ROWS];
                  // storage for data
```

```
static ADI_DEV_2D_BUFFER Buffer[NUM_BUFFERS];
                  // the actual buffer

u32 i; // counter

// create buffers for the driver to process
    for (i = 0; i < NUM_BUFFERS; i++) {

        Buffer[i].Data              = &Frames[i][0];
        Buffer[i].ElementWidth      = 2;
        Buffer[i].XCount            = (COLUMNS >> 1);
        Buffer[i].XModify           = 2;
        Buffer[i].YCount            = ROWS;
        Buffer[i].YModify           = 2;
        Buffer[i].CallbackParameter = &Buffer[i];
                      // gen callback, pArg = buffer address
        Buffer[i].pNext             = &Buffer[i+1];
    }
    Buffer[NUM_BUFFERS - 1].pNext   = NULL;
                        // terminate the chain of buffers

    // give the buffers to the driver to send out
    Result = adi_dev_Write(Handle, ADI_DEV_2D, (ADI_DEV_BUFFER
*)Buffer);
```

Creating Circular Buffers

The `ADI_DEV_CIRCULAR_BUFFER` data structure is used to describe data that the driver processes in a circular manner. Applications populate the various fields of the buffer to completely describe the data to the device driver.

For circular buffers, applications populate the following fields of the ADI_DEV_CIRCULAR_BUFFER structure:

- Data – If the buffer is provided to the adi_dev_Write() function, this field contains the starting address of the data sent out through the device. If the buffer is provided to the adi_dev_Read() function, this field contains the starting address of where the device driver stores data received in from the device.
- ElementWidth – This field indicates the width (in bytes) of each element sent out or read in.
- SubBufferCount – This field specifies the number of sub-buffers into which the data is divided.
- SubBufferElementCount – This field specifies the number of elements in each sub-buffer.
- CallbackType – This field specifies the frequency of callbacks that the application desires and contains one of the following:
    - ADI_DEV_CIRC_NO_CALLBACK – No callbacks
    - ADI_DEV_CIRC_SUB_BUFFER – Call back after completion of each sub-buffer
    - ADI_DEV_CIRC_FULL_BUFFER – Call back only after completion of the entire buffer
- pAdditionalInfo – This field is a device driver-dependent value. This field is not used for most device drivers. For information describing whether this field is used by the particular device driver, refer to the documentation specific to the device driver, located in the …/Blackfin/docs subdirectory.

For example, the following code fragment prepares and submits a circular buffer for transmission out through the device driver via the adi_dev_Write() function. The buffer describes a contiguous block of 1024 bytes of memory that is divided into 8 sub-buffers of 128 bytes each. The driver calls the application back after each sub-buffer is processed. Once dataflow is enabled, the driver repeatedly processes the circular buffer until dataflow is terminated.

```
#define SUB_BUFFERS         (8)
#define SUB_BUFFER_ELEMENTS (128)

static u8 Data[SUB_BUFFERS * SUB_BUFFER_ELEMENTS];
                                        // storage for data
static ADI_DEV_CIRCULAR_BUFFER Buffer;  // the actual buffer

    // create buffer for the driver to process
    Buffer.Data                  = Data;
    Buffer.ElementWidth          = 1;
    Buffer.SubBufferCount        = SUB_BUFFERS;
    Buffer.SubBufferElementCount = SUB_BUFFER_ELEMENTS;
  Buffer.CallbackType          = ADI_DEV_CIRC_SUB_BUFFER;

    // give the buffer to the driver to send out
    Result = adi_dev_Write(Handle, ADI_DEV_CIRC, (ADI_DEV_BUFFER
*)&Buffer);
```

Creating Sequential One-Dimensional Buffers

The `ADI_DEV_SEQ_1D_BUFFER` data structure is used to describe a sequential, one-dimensional (linear) array of data that a device driver processes. Similar to the standard one-dimensional buffers, applications populate the various fields of the buffer to completely describe the data to the device driver. In fact, a sequential, one-dimensional buffer is a concatenation of a standard one-dimensional buffer with a direction field appended to the end of the standard buffer. For sequential, one-dimensional buffers, applications populate the following fields of the ADI_DEV_SEQ_1D_BUFFER structure:

- Buffer – The buffer entry, which is identical to the standard one-dimensional buffer, is populated exactly as the standard one-dimensional buffer. Refer to "Creating One-Dimensional Buffers" on page 10-19 for information on populating a standard one-dimensional buffer.
- Direction – This field is populated with one of the following choices:
    - ADI_DEV_DIRECTION_INBOUND – Populate the field with this value if the buffer is for data received in from the device.
    - ADI_DEV_DIRECTION_OUTBOUND – Populate the field with this value if the buffer is for data that is transmitted out through the device.

The following code fragment prepares and submits a chain of two buffers. The first and third buffers describe data sent out through the device, and the second and fourth buffers describe data that is read in from the device. The driver calls back the application after the last buffer in the chain is processed, passing the address of the buffer that just completed as a parameter to the callback function.

```
#define ELEMENT_WIDTH     (1)
                         // width of a data element

#define NUM_BUFFERS       (2)     // number of buffers

#define INBOUND_ELEMENTS  (64)
                         // number of elements to read in
```

```
#define OUTBOUND_ELEMENTS  (2)
                         // number of elements to write out


static u32 InboundData[INBOUND_ELEMENTS];  // inbound data

static u32 OutboundData[OUTBOUND_ELEMENTS];
                                          // outbound data

// the actual buffers

static ADI_DEV_SEQ_1D_BUFFER SeqBuffer[NUM_BUFFERS];

   // create outbound buffer for the driver to process

   SeqBuffer[0].Buffer.Data            = OutboundData;

   SeqBuffer[0].Buffer.ElementCount   = OUTBOUND_ELEMENTS;

   SeqBuffer[0].Buffer.ElementWidth   = ELEMENT_WIDTH;

   SeqBuffer[0].Buffer.CallbackParameter = NULL;
                // no callback

   SeqBuffer[0].Buffer.pNext = (ADI_DEV_1D_BUFFER
*)&SeqBuffer[1];

   SeqBuffer[0].Direction = ADI_DEV_DIRECTION_OUTBOUND;

    // create inbound buffer for the driver to process

   SeqBuffer[1].Buffer.Data     = InboundData;

   SeqBuffer[1].Buffer.ElementCount = INBOUND_ELEMENTS;

   SeqBuffer[1].Buffer.ElementWidth = ELEMENT_WIDTH;

   SeqBuffer[1].Buffer.CallbackParameter = &SeqBuffer[1];
               // callback
```

```
        SeqBuffer[1].Buffer.pNext = NULL;     // end of chain
        SeqBuffer[1].Direction = ADI_DEV_DIRECTION_INBOUND;
            // give the buffers to the driver
        Result = adi_dev_SequentialIO(Handle, ADI_DEV_SEQ_1D,
    (ADI_DEV_BUFFER *)SeqBuffer);
```

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.

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 a deeper understanding of the design. This section is particularly useful for writers of physical drivers who need 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 `adi_dev.h` file, which is located in the `Blackfin/Include/Drivers` directory.

The "Device Manager Public Data Types and Enumerations" section of this chapter contains tables for all the device manager data structures and enumerations.

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 is allocated to the device manager via the `adi_dev_Init` function.

The `ADI_DEV_BASE_MEMORY` macro defines the number of bytes that the device manager needs. The `ADI_DEV_DEVICE_MEMORY` macro defines the number of bytes the device manager needs to control each physical driver. When providing memory to the device manager, the client provides 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 opened simultaneously in the system.

Handles

Next in the `adi_dev.h` file are `typedef` statements for the various handle types used by the device manager. Typically, handles are pointers to data structures 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 dataflow direction. These enumerations are not extensible by physical drivers.

Command IDs

The next section in the `adi_dev.h` file 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 starting with the enumeration start value for the driver.

Also included in this section is a data structure defining a configuration command pair. This is provided as a convenience—it 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 "adi_dev_Control" on page 10-58).

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. Similar to command IDs, physical drivers can add their own callback events beginning with the enumeration start value for the driver.

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. Similar to command IDs and callback events, physical drivers can add their own return codes beginning with the enumeration start value for the driver.

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 completions, or make a callback on whole buffer completions.

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 that allows the device drivers access to a small amount of memory attached to each buffer. How (or if) the device driver uses this reserved area 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 ADI_DEV_BUFFER data structure, which represents a union of one-dimensional, two-dimensional, and circular buffers. This data type is used as a convenient method to refer to a buffer in a generic fashion, without knowing the specific type of buffer. The adi_dev_Read and adi_dev_Write API functions use the ADI_DEV_BUFFER data 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 structure `ADI_DEV_PDD_ENTRY_POINT` 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 "Device Manager API Reference" on page 10-56.

Device Manager Code

All code for the device manager is kept in the `adi_dev.c` file. This section describes the code of the device manager. This file is located in the `Blackfin/Lib/Src/Drivers` directory.

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 in sections that follow. 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`
- `adi_dev_Open`
- `adi_dev_Close`
- `adi_dev_Read`
- `adi_dev_Write`
- `adi_dev_Control`

Refer to "Device Manager API Reference" on page 10-56 for more API information.

adi_dev_Init Functional Description

The `adi_dev_Init` function is used to initialize the device manager. For detailed reference information, see "adi_dev_Init" on page 10-59.

Processing begins by checking to ensure enough memory is 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 Functional Description

The `adi_dev_Open` function is used to open a device for use. For detailed reference information, see "adi_dev_Open" on page 10-61.

Processing begins by finding a free `ADI_DEV_DEVICE` data structure 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 is 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 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 determine whether it is supported by peripheral DMA. The device manager saves this information in the `ADI_DEV_DEVICE` structure.

adi_dev_Close Functional Description

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 10-57.

After the device handle is validated, assuming error checking is enabled, the function calls the `adi_pdd_Control` function of the physical driver to terminate dataflow. Once dataflow is 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 to 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 later.

adi_dev_Read Functional Description

The adi_dev_Read function is called by the application to provide the device with buffers into which inbound data is stored. Assuming error checking is enabled, processing begins in this function by validating the device handle and ensuring that the device is open for inbound (or bidirectional) traffic and that the dataflow method is defined. If the dataflow method is not yet 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 10-63.

The pBuffer parameter passed into the function can point to a single buffer or a chain of buffers. Furthermore, if the device is supported by peripheral DMA, the reserved area within the buffer data structure must be configured appropriately. All of these details are taken care of in the PrepareBufferList static function (see "PrepareBufferList" on page 10-43 for more information on this function).

Once the buffer list is prepared, a check is made to see whether 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_Buffer 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 because 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) is 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 Functional Description

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 10-65.

adi_dev_Control Functional Description

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 10-58.

Processing within the `adi_dev_Control` function is based upon the command ID passed in as a parameter. Some commands can be processed entirely by the device manager, some commands are processed by the physical driver only, and other commands 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 the command needs to pass 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 allows each physical driver to extend the command IDs and allows 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 if the device supports two-dimensional data movement. On Blackfin processors, if a device is supported by peripheral DMA, 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 two-dimensional 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, the device manager takes no action other than noting 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, the default value used for the DMA configuration control register is updated with settings appropriate for the dataflow method. Further, once the dataflow method is 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 "SetDataflow" on page 10-44 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, as described in "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. Typically, this 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 the application guarantees devices that are opened for inbound or bidirectional dataflow always have a buffer in which to store data that is received, and 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 call-back 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, regardless of whether 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 in use, the invocation of the `PDDCallback` function in the device manager is 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 validated first. The function then determines the event that 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 entire 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 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 in use, the invocation of the `DMACallback` function in the device manager is 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 (when the device is supported by peripheral DMA) or the physical driver (when 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 differ, 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 is 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 DMA 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 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 of the buffers are initialized appropriately, 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` or `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 that 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, in which there is only a single device manager in the system, any number of physical drivers can be present in a system.

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. Although multiple physical drivers may exist simultaneously in a system, multiple physical drivers should never control the same device.

In general, a physical driver controls 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 is 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 are 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 is used for all interrupt processing. This simplifies the task of porting device drivers to different operating environments, toolchains, and operating systems.

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 the device is supported by peripheral DMA. If the physical driver responds in the affirmative, the device manager controls all DMA activity (such as initialization, providing data buffers, callback mechanisms, and so on) via the DMA manager API. 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 use the services of the deferred callback manager exclusively in order to post callbacks into the device manager. See “Deferred Callback Manager” on page 5-1 for more information.

Physical drivers have their own API, which is accessed by the device manager. The following sections describe the API and the functionality 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 device manager include file (`adi_dev.h`).

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 that follow. 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 in the sections that follow. 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 `adi_ppi.c` file, has a companion `adi_ppi.h` include file. The only contents of the include file are the extensible definitions that the physical driver makes 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.

Physical drivers can create any number of additional command IDs. Applications can issue these command IDs via the `adi_dev_Control` API function. When the `adi_dev_Control` function of the device manager sees an extended command ID, 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. Physical drivers can create any number of additional event IDs. 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 extended event ID, 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 controls 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. Physical drivers can create any number of additional return codes. These 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.

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 a high level of detail as to what caused the error.

The adi_dev.h file contains the starting enumeration values for each physical driver. Use this value as the starting value for all command IDs, event IDs, and return codes.

ADI_DEV_PDD_ENTRY_POINT

The physical driver's include function must include a declaration of the entry point into the driver. This declaration declares, 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 informs 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, are declared static so that they are not exposed to any other software components. The only global piece of code or data is the entry point address. The entry point is a simple structure that contains the addresses of the physical driver API functions in the following order.

```
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 is in support of the five physical driver API functions. These functions and the logic that they need to provide are described in sections that follow. 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 Functional Description

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 10-83.

The `adi_pdd_Open` function should first verify that the device being requested is available for use and supports the requested data direction. Appropriate error codes are returned if the device is unavailable or does not support the requested direction.

The device being controlled is initialized and flushed of any stray data or pending interrupts. Any interrupts that are required to be handled in support of the device are hooked. For devices that are supported with peripheral DMA, typically only the error interrupt need be hooked. The interrupt manager of the system services is used for all hooking of interrupts. Enabling/disabling of interrupts through the system interrupt controller (SIC) is also accomplished using the interrupt manager service calls.

The physical driver saves the handle to the callback service. If non-NULL (meaning that deferred callbacks are in use), the physical driver invokes all callbacks through the service identified by the callback service handle. If NULL (meaning all callbacks are live and not deferred), the physical driver calls the device manager's callback function directly when sending events.

The physical driver also saves 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 returns ADI_DEV_RESULT_SUCCESS if successful.

adi_pdd_Control Functional Description

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 validates the physical driver handle upon entry into the function. For detailed reference information, see "adi_pdd_Control" on page 10-82.

Processing within the adi_pdd_Control function is 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 turns 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 is also 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 is 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, have an entry in the statement. If the physical driver receives a command ID that it does not understand, it typically returns the `ADI_DEV_RESULT_NOT_SUPPORTED` return code.

adi_pdd_Read Functional Description

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 validates the physical driver handle upon entry into the function. For detailed reference information, see "adi_pdd_Read" on page 10-85.

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 a 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 sets, 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 callback, the physical driver does not notify the device manager when the buffer is processed. If, however, the buffer is flagged for a callback (once the buffer is processed), the physical driver is obligated to set the processed flag and processed size field in the buffer, and to 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 Functional Description

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 10-86.

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 must 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 process the buffers immediately, or provide the logic and functionality to queue or stage these buffers for transmission at a later 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 sets based on when it considers a buffer processed and how much data was transmitted out through the device. Unlike the adi_pdd_Read case, the entire contents of the buffer is expected to 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 call-back, the physical driver does not notify the device manager when the buffer is processed. However, if the buffer is flagged for a callback, once the buffer is 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 Functional Description

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 validates the physical driver handle is upon entry into the function. For detailed reference information, see "adi_pdd_Close" on page 10-81.

After validating the driver handle, the `adi_pdd_Close` function terminates all data transmission and reception if it is not already stopped, as it is possible for the application to call the `adi_dev_Close` function while dataflow is enabled.

The function idles the device and leaves the device in a state such that it can be opened again, should the application reopen the device at a later time. All resources that were allocated in support of the device are released. For example, if an error interrupt was hooked during the `adi_pdd_Open` function, it is released as part of the `adi_pdd_Close` function.

Device Manager API Reference

This section provides the device manager API. 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 – 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

The `adi_dev_Close()` 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

```
u32 adi_dev_Close(
        ADI_DEV_DEVICE_HANDLE    DeviceHandle
);
```

Arguments

`DeviceHandle`	Handle used to identify the device

Return Value

`ADI_DEV_RESULT_SUCCESS`	Device closed successfully.
`ADI_DEV_RESULT_BAD_DEVICE_HANDLE`	Device handle does not identify a valid device.
`ADI_DEV_RESULT_DMA_ERROR`	Error occurred while closing down DMA for the device.
`xxx`	Device-specific return code

adi_dev_Control

Description

The `adi_dev_Control()` function sets or detects a configuration parameter for a device.

Prototype

```
u32 adi_dev_Control(
        ADI_DEV_DEVICE_HANDLE         DeviceHandle,
        u32                           Command,
        void                          *pArg
);
```

Arguments

`DeviceHandle`	Handle used to identify the device
`Command`	Command identifier
`pArg`	Address of command-specific parameter

Return Value

`ADI_DEV_RESULT_SUCCESS`	Function completed successfully.
`ADI_DEV_RESULT_BAD_DEVICE_HANDLE`	Device handle does not identify a valid device.
`ADI_DEV_RESULT_DMA_ERROR`	Error was reported while configuring the DMA manager.
`ADI_DEV_RESULT_NOT_SUPPORTED`	Command is not supported.
`xxx`	Device-specific return code

adi_dev_Init

Description

The `adi_dev_Init()` function creates a device manager and initializes memory for the device manager. This function is typically called at initialization time.

Prototype

```
u32 adi_dev_Init(
        void                        *pMemory,
        size_t                      MemorySize,
        u32                         *pMaxDevices,
        ADI_DEV_MANAGER_HANDLE      *pManagerHandle,
        void                        *pEnterCriticalParam
);
```

Arguments

`pMemory`	Pointer to an area of static memory used by the device manager
`MemorySize`	Size, in bytes, of memory 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`	Pointer to memory location where the handle to the device manager is stored
`pEnterCriticalParam`	Parameter that is passed to the function that protects critical areas of code

Return Value

`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

The `adi_dev_Open()` 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

```
u32 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`	Handle to the device manager that controls the device
`pEntryPoint`	Address of the physical driver's entry point
`DeviceNumber`	Number representing an index into a “device table” of available devices within the system, specifying which device to open. (Please also refer to “Virtual Devices and Device Indexing” on page 9-8.)
`ClientHandle`	Identifier defined by the application. The device manager passes this value back to the client as an argument in the callback function.
`pDeviceHandle`	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`	Data direction for the device: inbound, outbound or bidirectional. (See "ADI_DEV_DIRECTION" on page 10-67.)
`DMAHandle`	Handle to the DMA manager service that is used for this device (can be NULL if DMA is not used)
`DCBHandle`	Handle to the deferred callback service that is used for this device. If NULL, all callbacks will be live and not deferred.
`ClientCallback`	Address of the client's callback function

Return Value

`ADI_DEV_RESULT_SUCCESS`	Device was opened successfully.
`ADI_DEV_RESULT_BAD_MANAGER_HANDLE`	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`	Device is already in use.
`xxx`	Device-specific return code

adi_dev_Read

Description

The `adi_dev_Read()` function reads data from a device or queues reception buffers to a device.

Prototype

```
u32 adi_dev_Read(
        ADI_DEV_DEVICE_HANDLE       DeviceHandle,
        ADI_DEV_BUFFER_TYPE         BufferType,
        ADI_DEV_BUFFER              *pBuffer
);
```

Arguments

`DeviceHandle`	Handle used to identify the device
`BufferType`	Identifies type of buffer: one-dimensional, two-dimensional, or circular. (See "ADI_DEV_BUFFER_TYPE" on page 10-66.)
`pBuffer`	Address of the buffer or first buffer in a chain of buffers. (See "ADI_DEV_BUFFER" on page 10-79.)

Return Value

`ADI_DEV_RESULT_SUCCESS`	Function completed successfully.
`ADI_DEV_RESULT_BAD_DEVICE_HANDLE`	Device handle does not identify a valid device.
`ADI_DEV_RESULT_DMA_ERROR`	Error was reported while configuring the DMA manager.
`ADI_DEV_RESULT_DATAFLOW_UNDEFINED`	Dataflow method has not yet been set.
`xxx`	Device-specific return code

adi_dev_Terminate

Description

The `adi_dev_Terminate()` function frees up all memory used by the device manager, stops data flow, closes all open device drivers, and terminates the device manager.

Prototype

```
u32 adi_dev_Terminate(
        ADI_DEV_MANAGER_HANDLE       ManagerHandle
);
```

Arguments

`ManagerHandle`	Handle to the device manager

Return Value

`ADI_DEV_RESULT_SUCCESS`	Function completed successfully.

adi_dev_Write

Description

The `adi_dev_Write()` function writes data to a device or queues transmission buffers to a device.

Prototype

```
u32 adi_dev_Write(
        ADI_DEV_DEVICE_HANDLE       DeviceHandle,
        ADI_DEV_BUFFER_TYPE         BufferType,
        ADI_DEV_BUFFER              *pBuffer
);
```

Arguments

`DeviceHandle`	Handle used to identify the device.
`BufferType`	Identifies type of buffer: one-dimensional, two-dimensional, or circular. (See "ADI_DEV_BUFFER_TYPE" on page 10-66.)
`pBuffer`	Address of the buffer or first buffer in a chain of buffers. (See "ADI_DEV_BUFFER" on page 10-79.)

Return Value

`ADI_DEV_RESULT_SUCCESS`	Function completed successfully.
`ADI_DEV_RESULT_BAD_DEVICE_HANDLE`	Device handle does not identify a valid device.
`ADI_DEV_RESULT_DMA_ERROR`	Error was reported while configuring the DMA manager.
`ADI_DEV_RESULT_DATAFLOW_UNDEFINED`	Dataflow method has not yet been set.
`xxx`	Device-specific return code

Device Manager Public Data Types and Enumerations

This section defines public data structures and enumerations used by the device manager service.

ADI_DEV_BUFFER_TYPE

The `ADI_DEV_BUFFER_TYPE` enumeration is used to specify the type of buffer provided to the driver. Table 10-1 describes the values for the `ADI_DEV_BUFFER_TYPE` enumeration.

Table 10-1. ADI_DEV_BUFFER_TYPE

Name	Description
`ADI_DEV_BUFFER_UNDEFINED`	Undefined type
`ADI_DEV_1D`	One-dimensional buffer
`ADI_DEV_2D`	Two-dimensional buffer
`ADI_DEV_CIRC`	Circular buffer
`ADI_DEV_SEQ_1D`	Sequential one-dimensional buffer
`ADI_DEV_SWITCH`	Switch buffer
`ADI_DEV_UPDATE_SWITCH`	Buffer to chain with the existing switch buffer
`ADI_DEV_BUFFER_SKIP`	Skip this buffer and DMA channel (for multi-DMA devices)
`ADI_DEV_BUFFER_TABLE`	Table of buffers
`ADI_DEV_BUFFER_END a`	End of buffer array

ADI_DEV_MODE

The ADI_DEV_MODE enumeration is used to specify the dataflow method. Table 10-2 describes the values for the ADI_DEV_MODE enumeration.

Table 10-2. ADI_DEV_MODE

Name	Description
ADI_DEV_MODE_UNDEFINED	Undefined mode
ADI_DEV_MODE_CIRCULAR	Circular buffer
ADI_DEV_MODE_CHAINED	Chained buffer
ADI_DEV_MODE_CHAINED_LOOPBACK	Chained buffer with loopback
ADI_DEV_MODE_SEQ_CHAINED	Sequential chained buffer
ADI_DEV_MODE_SEQ_CHAINED_LOOPBACK	Sequential chained buffer with loopback

ADI_DEV_DIRECTION

The ADI_DEV_DIRECTION enumeration is used to specify the data direction. Table 10-3 describes the values for ADI_DEV_DIRECTION enumeration.

Table 10-3. ADI_DEV_DIRECTION

Name	Description
ADI_DEV_DIRECTION_UNDEFINED	Undefined direction
ADI_DEV_DIRECTION_INBOUND	Inbound direction (read)
ADI_DEV_DIRECTION_OUTBOUND	Outbound direction (write)
ADI_DEV_DIRECTION_BIDIRECTIONAL	Both inbound and outbound (read and write)

CALLBACK EVENTS

The device manager header file lists enumerations for callback events which are extensible by the physical device driver (PDD).

The starting value for device manager specific enumerations is 0x40000000. This value is defined in the `services.h` file as `ADI_DEV_ENUMERATION_START`.

Table 10-4 lists enumeration values for callback events.

Table 10-4. Callback Events

Name	Description
`ADI_DEV_EVENT_START`	0x40000000 - starting point
`ADI_DEV_EVENT_BUFFER_PROCESSED`	0x0001 - buffer completed processing
`ADI_DEV_EVENT_SUB_BUFFER_PROCESSED`	0x0002 - a circular sub-buffer has been processed
`ADI_DEV_EVENT_DMA_ERROR_INTERRUPT`	0x0003 - DMA controller generated an error interrupt

RESULT CODES

The device manager header file `adi_dev.h` lists enumeration values for function return result codes, which are extensible by the physical device driver (PDD).

The starting value for device manager specific enumerations is 0x40000000. This value is defined in the file `services.h` as `ADI_DEV_ENUMERATION_START`.

Table 10-5 lists enumeration values for result codes.

Table 10-5. Result Codes

Name	Description
ADI_DEV_RESULT_SUCCESS	Generic success = 0
ADI_DEV_RESULT_FAILED	Generic failure = 1
ADI_DEV_RESULT_START	0x40000000 - starting point
ADI_DEV_RESULT_NOT_SUPPORTED	Functionality is not supported
ADI_DEV_RESULT_DEVICE_IN_USE	Device already in use
ADI_DEV_RESULT_NO_MEMORY	Insufficient memory for operation
ADI_DEV_RESULT_NOT_USED_1	No longer used
ADI_DEV_RESULT_BAD_DEVICE_NUMBER	Bad device number
ADI_DEV_RESULT_DIRECTION_NOT_SUPPORTED	Data direction not supported
ADI_DEV_RESULT_BAD_DEVICE_HANDLE	Bad device handle
ADI_DEV_RESULT_BAD_MANAGER_HANDLE	Bad device manager handle
ADI_DEV_RESULT_BAD_PDD_HANDLE	Bad physical driver handle
ADI_DEV_RESULT_INVALID_SEQUENCE	Invalid sequence of commands
ADI_DEV_RESULT_ATTEMPTED_READ_ON_OUTBOUND_DEVICE	Attempted read on outbound device
ADI_DEV_RESULT_ATTEMPTED_WRITE_ON_INBOUND_DEVICE	Attempted write on inbound device
ADI_DEV_RESULT_DATAFLOW_UNDEFINED	Dataflow method is undefined
ADI_DEV_RESULT_DATAFLOW_INCOMPATIBLE	Operation incompatible with the data-flow method
ADI_DEV_RESULT_BUFFER_TYPE_INCOMPATIBLE	Device does not support the given buf-fer type
ADI_DEV_RESULT_NOT_USED_2	No longer used
ADI_DEV_RESULT_CANT_HOOK_INTERRUPT	Cannot hook the interrupt
ADI_DEV_RESULT_CANT_UNHOOK_INTERRUPT	Cannot unhook the interrupt
ADI_DEV_RESULT_NON_TERMINATED_LIST	Non-NULL terminated buffer list

Table 10-5. Result Codes (Cont'd)

Name	Description
ADI_DEV_RESULT_NO_CALLBACK_FUNCTION_SUPPLIED	No callback function provided to Open function
ADI_DEV_RESULT_REQUIRES_UNIDIRECTIONAL_DEVICE	Requires unidirectional device
ADI_DEV_RESULT_REQUIRES_BIDIRECTIONAL_DEVICE	Requires bidirectional device
ADI_DEV_RESULT_TWI_LOCKED	TWI locked in other operation
ADI_DEV_RESULT_REQUIRES_TWI_CONFIG_TABLE	Requires configuration table for the TWI driver
ADI_DEV_RESULT_CMD_NOT_SUPPORTED	Command not supported
ADI_DEV_RESULT_INVALID_REG_ADDRESS	Accessing invalid device register address
ADI_DEV_RESULT_INVALID_REG_FIELD	Accessing invalid device register field location
ADI_DEV_RESULT_INVALID_REG_FIELD_DATA	Providing invalid device register field value
ADI_DEV_RESULT_ATTEMPT_TO_WRITE_READONLY_REG	Attempt to write a read-only register
ADI_DEV_RESULT_ATTEMPT_TO_ACCESS_RESERVE_AREA	Attempt to access a reserved location
ADI_DEV_RESULT_ACCESS_TYPE_NOT_SUPPORTED	Access type provided by the driver is not supported
ADI_DEV_RESULT_DATAFLOW_NOT_ENABLED	Sync mode- enable dataflow before providing buffers
ADI_DEV_RESULT_BAD_DIRECTION_FIELD	Sequential I/O mode- buffers provided with invalid direction
ADI_DEV_RESULT_BAD_IVG	Bad IVG number detected
ADI_DEV_RESULT_SWITCH_BUFFER_PAIR_INVALID	Invalid buffer pair provided with switch/update switch buffer type
ADI_DEV_RESULT_DMA_CHANNEL_UNAVAILABLE	No DMA channel available to process command/buffer
ADI_DEV_RESULT_ATTEMPTED_BUFFER_TABLE_NESTING	Buffer table nesting is not allowed

COMMAND IDs

The device manager header file `adi_dev.h` lists enumeration values for command IDs, which are extensible by the physical device driver (PDD).

The starting value for device manager specific enumerations is 0x40000000. This value is defined in the file `services.h` as `ADI_DEV_ENUMERATION_START`.

Table 10-6 lists enumeration values for Command IDs.

Table 10-6. Command IDs

Name	Description	Value
`ADI_DEV_CMD_START`	`ADI_DEV_ENUMERATION_START`	0x40000000
`ADI_DEV_CMD_UNDEFINED`	Undefined	
`ADI_DEV_CMD_END`	End of table	NULL
`ADI_DEV_CMD_PAIR`	Single command pair being passed	`ADI_DEV_CMD_VALUE_PAIR`
`ADI_DEV_CMD_TABLE`	Table of command pairs	`ADI_DEV_CMD_VALUE_PAIR`
`ADI_DEV_CMD_SET_DATAFLOW`	Enable/disable dataflow	`TRUE/FALSE`
`ADI_DEV_CMD_SET_DATAFLOW_METHOD`	Set dataflow method	`TRUE/FALSE`
`ADI_DEV_CMD_NOT_USED_1`	No longer used	
`ADI_DEV_CMD_SET_SYNCHRONOUS`	Set device to synchronous I/O	`TRUE/FALSE`
`ADI_DEV_CMD_NOT_USED_2`	No longer used	
`ADI_DEV_CMD_SET_STREAMING`	Set streaming mode	`TRUE/FALSE`
`ADI_DEV_CMD_GET_MAX_INBOUND_SIZE`	Get size of biggest inbound packet	`u32*`
`ADI_DEV_CMD_GET_MAX_OUTBOUND_SIZE`	Get size of biggest outbound packet	`u32*`

Table 10-6. Command IDs (Cont'd)

Name	Description	Value
ADI_DEV_CMD_GET_PERIPHERAL_DMA_SUPPORT	Query whether device supports processor DMA	u32*
ADI_DEV_CMD_GET_INBOUND_DMA_PMAP_ID	Get peripheral's DMA PMAP value for inbound data	ADI_DMA_PMAP*
ADI_DEV_CMD_GET_OUTBOUND_DMA_PMAP_ID	Get peripheral's DMA PMAP value for outbound data	ADI_DMA_PMAP*
ADI_DEV_CMD_SET_INBOUND_DMA_CHANNEL_ID	Set DMA channel ID for inbound DMA	ADI_DMA_CHANNEL_ID
ADI_DEV_CMD_GET_INBOUND_DMA_CHANNEL_ID	Get DMA channel ID for inbound DMA	ADI_DMA_CHANNEL_ID*
ADI_DEV_CMD_SET_OUTBOUND_DMA_CHANNEL_ID	Set DMA channel ID for outbound DMA	ADI_DMA_CHANNEL_ID
ADI_DEV_CMD_GET_OUTBOUND_DMA_CHANNEL_ID	Get DMA channel ID for outbound DMA	ADI_DMA_CHANNEL_ID*
ADI_DEV_CMD_GET_2D_SUPPORT	Query whether device supports 2D transfers	u32*
ADI_DEV_CMD_SET_ERROR_REPORTING	Enable/disable error reporting	TRUE/FALSE
ADI_DEV_CMD_FREQUENCY_CHANGE_PROLOG	Notification of impending core/system frequency change	ADI_DEV_FREQUENCIES*
ADI_DEV_CMD_FREQUENCY_CHANGE_EPILOG	Notification of new core/system frequency settings	ADI_DEV_FREQUENCIES*
ADI_DEV_CMD_REGISTER_READ	Read a single device register	ADI_DEV_ACCESS_REGISTER*
ADI_DEV_CMD_REGISTER_FIELD_READ	Read a specific device register field	ADI_DEV_ACCESS_REGISTER_FIELD *
ADI_DEV_CMD_REGISTER_TABLE_READ	Read a table of selective device registers	ADI_DEV_ACCESS_REGISTER*
ADI_DEV_CMD_REGISTER_FIELD_TABLE_READ	Read table of selective device register(s) field(s)	ADI_DEV_ACCESS_REGISTER_FIELD*
ADI_DEV_CMD_REGISTER_BLOCK_READ	Read a block of consecutive device registers	ADI_DEV_ACCESS_REGISTER_BLOCK*

Table 10-6. Command IDs (Cont'd)

Name	Description	Value
ADI_DEV_CMD_REGISTER_WRITE	Write to a single device register	ADI_DEV_ACCESS_REGISTER*
ADI_DEV_CMD_REGISTER_FIELD_WRITE	Write to a specific device register field	ADI_DEV_ACCESS_REGISTER_FIELD*
ADI_DEV_CMD_REGISTER_TABLE_WRITE	Write to a table of selective device registers	ADI_DEV_ACCESS_REGISTER
ADI_DEV_CMD_REGISTER_FIELD_TABLE_WRITE	Write to a table of selective device register(s) field(s)	ADI_DEV_ACCESS_REGISTER_FIELD*
ADI_DEV_CMD_REGISTER_BLOCK_WRITE	Write to a block of consecutive device registers	ADI_DEV_ACCESS_REGISTER_BLOCK*
ADI_DEV_CMD_UPDATE_1D_DATA_POINTER	Update the data pointer in a single 1D buffer	ADI_DEV_1D_BUFFER*
ADI_DEV_CMD_UPDATE_2D_DATA_POINTER	Update the data pointer in a single 2D buffer	ADI_DEV_2D_BUFFER*
ADI_DEV_CMD_UPDATE_SEQ_1D_DATA_POINTER	Update the data pointer in a single sequential 1D buffer	ADI_DEV_SEQ_1D_BUFFER*
ADI_DEV_CMD_GET_INBOUND_DMA_CURRENT_ADDRESS	Get current address register value of inbound DMA	u32*
ADI_DEV_CMD_GET_OUTBOUND_DMA_CURRENT_ADDRESS	Get current address register value of outbound DMA	u32*

ADI_DEV_1D_BUFFER

The data structure ADI_DEV_1D_BUFFER describes a normal one-dimensional buffer. Table 10-7 describes the elements of this data structure.

Table 10-7. ADI_DEV_1D_BUFFER

Name	Type	Description
Reserved	char[ADI_DEV_RESERVED_SIZE]	Reserved for physical device driver use
Data	void*	Pointer to data

Table 10-7. ADI_DEV_1D_BUFFER (Cont'd)

Name	Type	Description
`ElementCount`	`u32`	Data element count. Note that the total transfer size is 64 KB. For more information, see the *Hardware Reference Manual*.
`ElementWidth`	`u32`	Data element width (in bytes)
`CallbackParameter`	`void*`	Callback `flag`/`pArg` value
`ProcessedFlag`	`volatile u32`	Processed flag
`ProcessedElementCount`	`u32`	Number of bytes processed
`pNext`	`struct adi_dev_1d_buffer*`	Next buffer
`pAdditionalInfo`	`void*`	Device-specific pointer to additional information

ADI_DEV_2D_BUFFER

The data structure `ADI_DEV_2D_BUFFER` describes a normal one-dimensional buffer. Table 10-8 describes the elements of this data structure.

Table 10-8. ADI_DEV_2D_BUFFER

Name	Type	Description
`Reserved`	`char[ADI_DEV_RESERVED_SIZE]`	Reserved for physical device driver use
`Data`	`void*`	Pointer to data
`ElementWidth`	`u32`	Data element width (in bytes)
`XCount`	`u32`	`XCOUNT` value for 2D
`XModify`	`s32`	`XMODIFY` value for 2D
`YCount`	`u32`	`YCOUNT` value for 2D
`YModify`	`s32`	`YMODIFY` value for 2D
`CallbackParameter`	`void*`	Callback `flag`/`pArg` value

Table 10-8. ADI_DEV_2D_BUFFER (Cont'd)

Name	Type	Description
`ProcessedFlag`	`volatile u32`	Processed flag
`ProcessedElementCount`	`u32`	Number of bytes processed
`pNext`	`struct adi_dev_2d_buffer*`	Next buffer
`pAdditionalInfo`	`void*`	Device-specific pointer to additional information

ADI_DEV_CIRCULAR_BUFFER

The data structure `ADI_DEV_CIRCULAR_BUFFER` describes a normal circular buffer. Table 10-9 describes the elements of this room with data structure.

Table 10-9. ADI_DEV_CIRCULAR_BUFFER

Name	Type	Description
`Reserved`	`char[ADI_DEV_RESERVED_SIZE]`	Reserved for physical device driver use
`Data`	`void*`	Pointer to data
`SubBufferCount`	`u32`	Number of sub-buffers
`SubBufferElementCount`	`u32`	Number of data elements in one sub-buffer
`ElementWidth`	`u32`	Data element width (in bytes)
`CallbackType`	`ADI_DEV_CIRCULAR_CALLBACK`	Circular buffer callback switch
`pAdditionalInfo`	`void*`	Device-specific pointer to additional information

ADI_DEV_SEQ_1D_BUFFER

The data structure ADI_DEV_SEQ_1D_BUFFER describes a sequential one-dimensional buffer. Table 10-10 describes the elements of this data structure.

Table 10-10. ADI_DEV_SEQ_1D_BUFFER

Name	Type	Description
BufferType	ADI_DEV_BUFFER_TYPE	Buffer type
pDirection	union ADI_DEV_BUFFER	Pointer to a buffer of above type

ADI_DEV_BUFFER_PAIR

The data structure ADI_DEV_BUFFER_PAIR describes a sequential one-dimensional buffer. Table 10-11 describes the elements of this data structure.

Table 10-11. ADI_DEV_BUFFER_PAIR

Name	Type	Description
Buffer	ADI_DEV_1D_BUFFER	Buffer
Direction	ADI_DEV_DIRECTION	Direction

ADI_DEV_DMA_INFO

The data structure ADI_DEV_DMA_INFO holds peripheral DMA channel information. Table 10-12 describes the elements of this data structure.

Table 10-12. ADI_DEV_DMA_INFO

Name	Type	Description
MappingID	ADI_DMA_PMAP	DMA peripheral mapping ID
ChannelHandle	ADI_DMA_CHANNEL_HANDLE	Handle to this DMA channel

Table 10-12. ADI_DEV_DMA_INFO (Cont'd)

Name	Type	Description
SwitchModeFlag	u8	Switch mode status flag (TRUE when in switch mode)
pSwitchHead	ADI_DMA_DESCRIPTOR_UNION*	Head of switch buffer chain
pSwitchTail	ADI_DMA_DESCRIPTOR_UNION*	Tale of switch buffer chain
pNext	struct ADI_DEV_DMA_INFO*	Pointer to structure holding next DMA channel information

ADI_DEV_DMA_ACCESS

The data structure ADI_DEV_DMA_ACCESS accesses a peripheral's inbound and outbound DMA chain data. Table 10-13 describes the elements of this data structure.

Table 10-13. ADI_DEV_DMA_ACCESS

Name	Type	Description
DmaChannelCount	u8	Number of DMA channels to access and selected device
pData	void*	Start location of an array for the DMA-related data

ADI_DEV_FREQUENCIES

The data structure ADI_DEV_FREQUENCIES maintains information about the clock frequency changes. Table 10-14 describes the elements of this data structure.

Table 10-14. ADI_DEV_FREQUENCIES

Name	Type	Description
CoreClock	u32	Core clock (CCLK)
SystemClock	u32	System clock (SCLK)

ADI_DEV_ACCESS_REGISTER

The data type `ADI_DEV_ACCESS_REGISTER` is for accessing a single, off-chip device register. Table 10-15 describes the elements of this data structure.

Table 10-15. ADI_DEV_ACCESS_REGISTER

Name	Type	Description
`Address`	`u16`	Device register address
`Data`	`u16`	Data to be written to, or read from the register

ADI_DEV_ACCESS_REGISTER_BLOCK

The data type `ADI_DEV_ACCESS_REGISTER_BLOCK` is for accessing blocks of consecutive, off-chip device registers. Table 10-16 describes the elements of this data structure.

Table 10-16. ADI_DEV_ACCESS_REGISTER_BLOCK

Name	Type	Description
`Count`	`u32`	Number of registers to be accessed
`Address`	`u16`	Starting address of register block
`pData`	`u16*`	Pointer to an array of register data to be written to, or read from the register block

ADI_DEV_ACCESS_REGISTER_FIELD

The data type `ADI_DEV_ACCESS_REGISTER_FIELD` is for accessing fields within off-chip device registers. Table 10-17 describes the elements of this data structure.

Table 10-17. ADI_DEV_ACCESS_REGISTER_FIELD

Name	Type	Description
`Address`	`u16`	Address of register to be accessed
`Field`	`u16`	Register field to be accessed (see off-chip driver header file)
`Data`	`u16`	Data to be written to, or read from the register field

ADI_DEV_BUFFER

The union `ADI_DEV_BUFFER` describes a union of all the buffer types. Table 10-18 describes the elements of this union.

Table 10-18. ADI_DEV_BUFFER

Name	Type	Description
`OneD`	`ADI_DEV_1D_BUFFER`	One-dimensional buffer
`TwoD`	`ADI_DEV_2D_BUFFER`	Two-dimensional buffer
`Circular`	`ADI_DEV_CIRCULAR_BUFFER`	Circular buffer
`Seq1D`	`ADI_DEV_SEQ_1D_BUFFER`	One-dimensional sequential buffer
`BufferPair`	`ADI_DEV_BUFFER_PAIR`	Buffer pair

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 – 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

The `adi_pdd_Close()` 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

```
u32 adi_pdd_Close(
                ADI_PDD_DEVICE_HANDLE   PDDHandle
);
```

Arguments

`PDDHandle`	Handle used to identify the device

Return Value

`ADI_DEV_RESULT_SUCCESS`	Device closed successfully.
`xxx`	Device-specific code

adi_pdd_Control

Description

The `adi_pdd_Control()` function sets or detects a configuration parameter for a device.

Prototype

```
u32 adi_pdd_Control(
        ADI_DEV_PDD_HANDLE          PDDHandle,
        u32                         Command,
        void                        *pArg
);
```

Arguments

`PDDHandle`	Handle used to identify the device
`Command`	Command identifier
`pArg`	Address of command-specific parameter

Return Value

`ADI_DEV_RESULT_SUCCESS`	Function completed successfully.
`ADI_DEV_RESULT_NOT_SUPPORTED`	Command is not supported.
`xxx`	Device-specific return code

adi_pdd_Open

Description

The `adi_pdd_Open()` 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

```
u32 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`	Handle to the device manager that controls the physical driver
`DeviceNumber`	Number identifying which device is open. Device numbers begin with zero. For example, if there are four serial ports, they are numbered 0 through 3.
`DeviceHandle`	Device manager-supplied parameter that uniquely identifies the device to the device manager
`pPDDHandle`	Pointer to a location where the physical driver stores a handle that uniquely identifies the device to the physical driver

`Direction`	Data direction for the device: inbound, outbound, or bidirectional
`pEnterCriticalParam`	Parameter that is passed to the function that protects critical areas of code.
`DMAHandle`	Handle to the DMA manager service that is used for this device (can be NULL if DMA is not used)
`DCBHandle`	Handle to the deferred callback service that is used for this device. If NULL, all callbacks are live and not deferred.
`DMCallback`	Address of the device manager's callback function

Return Value

`ADI_DEV_RESULT_SUCCESS`	Device opened successfully.
`ADI_DEV_RESULT_DEVICE_IN_USE`	Device manager handle does not point to a device manager.
`xxx`	Device-specific return code

adi_pdd_Read

Description

The `adi_pdd_Read()` function provides buffers to a device for reception of inbound data. This function is never called for devices that are supported by peripheral DMA.

Prototype

```
u32 adi_pdd_Read(
        ADI_DEV_PDD_HANDLE           PDDHandle,
        ADI_DEV_BUFFER_TYPE          BufferType,
        ADI_DEV_BUFFER               *pBuffer
);
```

Arguments

`PDDHandle`	Handle used to identify the device
`BufferType`	Identifies type of buffer: one-dimensional, two-dimensional or circular
`pBuffer`	Address of the buffer or first buffer in a chain of buffers

Return Value

`ADI_DEV_RESULT_SUCCESS`	Function completed successfully.
`ADI_DEV_RESULT_BAD_PDD_HANDLE`	PDD handle does not identify a valid device.
`xxx`	Device-specific return code

adi_pdd_Write

Description

The `adi_pdd_Write()` 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

```
u32 adi_pdd_Write(
        ADI_DEV_PDD_HANDLE          PDDHandle,
        ADI_DEV_BUFFER_TYPE         BufferType,
        ADI_DEV_BUFFER              *pBuffer
);
```

Arguments

`PDDHandle`	Handle used to identify the device
`BufferType`	Identifies type of buffer: one-dimensional, two-dimensional, or circular
`pBuffer`	Address of the buffer or first buffer in a chain of buffers

Return Value

`ADI_DEV_RESULT_SUCCESS`	Function completed successfully.
`ADI_DEV_RESULT_BAD_PDD_HANDLE`	PDD handle does not identify a valid device.
`xxx`	Device-specific return code

Examples

Examples showing how to use the device driver model as well as Analog Devices device drivers are provided with the device driver and system services distribution media.

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.

11 REAL-TIME CLOCK SERVICE

This chapter describes how to use the real-time clock service within the system services library to enable the features of the real-time clock on Blackfin processors. The application programming interface (API) described in this chapter allows real-time clock events to be scheduled and serviced in a manner consistent with the other system services.

This chapter contains the following sections:

Introduction

The real-time clock service provides an easy-to-use interface to the real-time clock that is available on most Blackfin processors. In addition to the reading and writing of the date and time, the service allows an application to coordinate eight different real-time clock events in a consistent manner, regardless of the processor derivative. There are five periodic events that can be enabled to occur each second, each minute, each hour, each day at midnight, and each day at a specific time. There is an alarm that occurs once at a specific date and time. There is a stopwatch event that occurs when a given duration has elapsed. And there is an event which signals the completion of all pending register writes.

The sections in this chapter cover the basic operation of the real-time clock service, describing the application programming interface (API) and providing the necessary instructions for initializing and using the real-time clock service in a client application.

The debug version of the system services library provides parameter checking for a more complete test of the API function parameters and other error conditions. Analog Devices strongly recommends that development work be done using the debug versions of the system services library, and that final test and deployment be done with the release version of the library.

Operation

This section describes the overall operation of the real-time clock service. Details on the application programming interface (API) can be found later in this chapter.

Initialization

Prior to using the real-time clock service, the application must initialize the service by calling the initialization function, `adi_rtc_Init()`. Unlike some system services which require the application to provide memory to the service upon initialization, the real-time clock service requires no additional memory. The only parameter passed to `adi_rtc_Init()` is a critical region parameter that may be used later if the interrupt manager is called upon to protect critical regions of code. In the current implementation of the system services, a NULL pointer is used for the critical region parameter. For more information, refer to "Protecting Critical Code Regions" on page 2-13.

Before initializing the real-time clock service, the application should initialize the interrupt manager by calling `adi_int_Init()`. If callbacks are to be deferred, rather than "live", then the deferred callback (DCB) manager should also be initialized by calling `adi_dcb_Init()`.

Some of the real-time clock hardware memory-mapped registers (MMR) have certain limitations, in that they allow only one value to be written in any given 1 Hz cycle. A value written to the MMR does not take effect until the next 1 Hz tick, and any subsequent value written to the same register within that time period is discarded. The real-time clock service implements a register caching system that works around this issue, to ensure that none of the intended functionality is lost. The application should not attempt to access any of the memory-mapped registers directly, but should depend solely upon the API functions provided by the service.

When the application calls `adi_rtc_Init()`, the register caches are cleared. The prescaler bit is set so the real-time clock runs in 1 Hz mode, like a regular clock. The callback environment is initialized and the event flag register is cleared of any pending events. The real-time clock interrupt handler is hooked into the IVG chain, and the real-time clock service is ready to go. Real-time clock interrupts are initialized to also "wake up" the processor. If this is not the desired behavior, the application may call the API function `adi_rtc_DisableWakeup()`.

Termination

When the application no longer requires the real-time clock service, it calls the termination function, `adi_rtc_Terminate()`. This function removes any callbacks that were installed, cleaning up all statically-defined data structures.

Setting and Reading the Date and Time

To set the date and time of the real-time clock, the real-time clock service provides an API function called `adi_rtc_SetDateTime()`. To use this function, the application declares a structure of type `tm`, which is a C-standard time structure defined in the system include file, `time.h`. The application sets the fields of this structure with the month, day, year, hour, minute, and second and passes a pointer to the structure to `adi_rtc_SetDateTime()`. The function converts the data to 32-bit integer format and loads it into the `RTC_STAT` register, where the real-time clock hardware maintains the date and time. The `RTC_STAT` register is shown below. Each field of the `tm` structure is represented by a range of bits within the register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Day Counter (0–32767)															Hour (cont'd)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Hour (0–23)				Minutes (0–59)						Seconds (0–59)					

The `adi_rtc_SetDateTime()` function only needs to be called once, although it can be called multiple times. The date and time take effect on the next 1 Hz tick and are maintained in one-second increments for as long as the external battery power source is present. An application may subsequently call upon the real-time clock service to obtain the current date and time, or to install callbacks for the clock-driven events.

The real-time clock service provides an API function called `adi_rtc_GetDateTime()` which allows the application to retrieve the date and time from the memory-mapped `RTC_STAT` register. To use this function, the application declares a structure of type `tm`, which is a C-standard time structure defined in the system include file, `time.h`. The application passes a pointer to the `tm` structure. The function reads the date and time from the `RTC_STAT` register and returns the information in the `tm` structure.

The real-time clock service supports only the functionality of the hardware. It does not make adjustments to the date and time for daylight savings or time zones. The application can make these adjustments easily since the date and time are stored in a `tm` structure for compatibility with standard time functions found in the VisualDSP C/C++ Compiler Run-Time Library.

Real-Time Clock Events

The real-time clock service provides a mechanism for allowing certain predefined conditions, referred to as *events*, to be serviced at the interrupt level during regular program execution. Each event is identified by a unique `Event ID` that is defined in the include file, `adi_rtc.h`. The application provides a callback function to handle an event. The real-time clock service provides the mechanism for installing and removing callbacks, and for invoking a callback when the appropriate conditions are met. The real-time clock service supports eight different events. The application enables an event by calling the `adi_rtc_InstallCallback()` function, passing the appropriate `Event ID`.

One Second Periodic Event

The application may enable an interrupt to occur when the *seconds* counter in the `RTC_STAT` register advances. This interrupting event becomes enabled when the application calls the `adi_rtc_InstallCallback()` function, passing the argument `ADI_RTC_EVENT_SECONDS`. This event stays enabled and periodic interrupts continue to occur every second, until the application disables the event, by passing the argument `ADI_RTC_EVENT_SECONDS` to the `adi_rtc_RemoveCallback()` function.

One Minute Periodic Event

The application may enable a periodic interrupt to occur when the *minutes* counter of the `RTC_STAT` register advances. This interrupting event becomes enabled when the application calls the `adi_rtc_InstallCallback()` function, passing the argument `ADI_RTC_EVENT_MINUTES`. This event stays enabled and periodic interrupts continue to occur every minute, until the application disables the event by passing the argument `ADI_RTC_EVENT_MINUTES` to the `adi_rtc_RemoveCallback()` function.

Hourly Periodic Event

The application may enable a periodic interrupt to occur when the *hours* counter of the `RTC_STAT` register advances. This interrupting event becomes enabled when the application calls the `adi_rtc_InstallCallback()` function, passing the argument `ADI_RTC_EVENT_HOURS`. This event stays enabled and periodic interrupts continue to occur every hour until the application disables the event by passing the argument `ADI_RTC_EVENT_HOURS` to the `adi_rtc_RemoveCallback()` function.

Daily Periodic Event

The application may enable a periodic interrupt to occur at midnight when the *days* counter of the `RTC_STAT` register advances. This interrupting event becomes enabled when the application calls the `adi_rtc_InstallCallback()` function, passing the argument `ADI_RTC_EVENT_DAYS`. This event stays enabled and periodic interrupts continue to occur each day until the application disables the event by passing the argument `ADI_RTC_EVENT_DAYS` to the `adi_rtc_RemoveCallback()` function.

Periodic or One-Shot Stopwatch Event

The application may set up a *stopwatch* event to interrupt when a specified time duration has elapsed. The same stopwatch event can be either periodic or one shot. The event becomes enabled when the application passes the argument `ADI_RTC_EVENT_STOPWATCH` to the `adi_rtc_InstallCallback()` function, also passing the number of seconds of the stopwatch time duration. The stopwatch event occurs only once by default, as a "one-shot" stopwatch event. The callback function may then either disable the event by passing the argument `ADI_RTC_EVENT_STOPWATCH` to the `adi_rtc_RemoveCallback()` function, or it may re-enable the stopwatch event by calling the API function `adi_rtc_ResetStopwatch()`. If re-enabled, the stopwatch becomes a periodic event.

Once Only Alarm Event

The application may enable an alarm event that interrupts *once* on a given day at a specific time. This interrupting event becomes enabled when the application passes the argument `ADI_RTC_EVENT_ONCE_ALM` to the `adi_rtc_InstallCallback()` function, also passing a `tm` structure containing the date and time of the alarm. After the alarm event occurs, the callback may be removed by passing the argument `ADI_RTC_EVENT_ONCE_ALM` to the `adi_rtc_RemoveCallback()` function. Then, another such event can be scheduled by installing the callback again. After this event has been scheduled, and before it occurs, it may be canceled by removing the callback.

Each Day Alarm Event

The application may enable an alarm that interrupts at a specific time *each day* as long as the alarm is enabled. This interrupting event becomes enabled when the application passes the argument ADI_RTC_EVENT_EACH_DAY_ALM to the adi_rtc_InstallCallback() function, also passing a tm structure containing the time of day that the alarm should occur. The event stays enabled and the interrupt occurs each day at the specified time, until the application disables the event by calling the adi_rtc_RemoveCallback() function, passing the argument ADI_RTC_EVENT_EACH_DAY_ALM.

Pending Writes Complete Event

All *writes* to the memory-mapped registers RTC_STAT, RTC_ALARM, RTC_SWCNT, and RTC_ICTL are synchronized to the 1 Hz tick. The application may want to be alerted when the pending writes have completed. The application may enable an interrupt to occur when all pending writes to these registers have completed. This event becomes enabled when the application passes the argument ADI_RTC_EVENT_WRITES_COMPLETE to the adi_rtc_InstallCallback() function. The event stays enabled until the application disables the event by calling the adi_rtc_RemoveCallback() function, passing the argument ADI_RTC_EVENT_WRITES_COMPLETE.

Callbacks

The application may enable and process any of the real-time clock events by defining and installing a *callback* for that event. A callback is simply a function, associated with a specific event, or a group of events, that is executed whenever the associated event occurs. A separate callback function may be written for each event, or a single callback function may handle all the real-time clock events, perhaps within a case statement. Unless a callback is deferred, it is executed from within the interrupt service routine, by the real-time clock interrupt handler. The handler, part of the real-time clock service, is called from the interrupt manager whenever a real-time clock interrupt is triggered by one of the eight possible real-time clock events. The real-time clock service provides the function to install callbacks, the function to remove callbacks, and the interrupt handler which invokes the callbacks. The application provides the callback function, which controls how each event is processed, and also an optional data structure, comprised of event-specific information, that is passed to the callback function from the real-time clock service interrupt handler. A coding example appears later in this section which demonstrates how that data structure can be used.

The Callback List

To process callbacks, the real-time clock maintains a table with an entry for each of the eight events. Each entry contains a pointer to a callback function, or to NULL if no callback has been installed. When the application calls `adi_rtc_InstallCallback` passing an `Event ID`, the real-time clock service inserts a pointer to the callback function into the entry for the event. When the event occurs, the real-time clock interrupt handler uses the entry to find and execute the callback function designated for this event. When the application calls `adi_rtc_RemoveCallback` to disable an event, the callback pointer is removed from the callback list entry for that event.

Installing a Callback

To install callback functionality for one of the real-time clock events, the application simply calls the API function `adi_rtc_InstallCallback` passing the `Event ID`. For some events, such as alarms and stopwatch events, other information is passed to this function, such as the time that the alarm should occur, or the duration of a stopwatch period. The `adi_rtc_InstallCallback` function uses this information to configure the hardware registers to generate the interrupt at the appropriate time. An optional data structure called `ClientHandle` is also passed to the `adi_rtc_InstallCallback` function, containing information to be stored in the callback list and passed to the callback when it occurs.

Removing a Callback

The application may disable the processing of an event by passing the `Event ID` as a parameter to the function `adi_rtc_RemoveCallback`. This function removes the callback from the callback list entry for the event, and also configures the hardware so it no longer generates an interrupt for the event.

The Real-Time Clock Service Interrupt Handler

The real-time clock service has an interrupt handler that is "hooked" into the interrupt manager's "chain" for the associated interrupt vector group (IVG). Note the difference between the interrupt handler, which is defined in the real-time clock service to process real-time clock events, and the interrupt service routine (ISR), which is defined in the interrupt manager to handle any interrupting event of a given IVG level. The interrupt manager's ISR executes the real-time clock interrupt handler when a real-time clock interrupting event occurs. The real-time clock interrupt handler reads the event flags in the `RTC_ISTAT` register to determine which event triggered the interrupt. The handler then looks at the callback list entry for the event (see "The Callback List " on page 11-9). The entry will point to the callback function (see "Callbacks" on page 11-8) and to the `ClientHandle` parameter (see "Using the ClientHandle Parameter in a Callback" below). The handler executes the callback, passing the `ClientHandle` parameter. When the callback is complete, it returns to the real-time clock interrupt handler, which returns control to the interrupt manager.

Using the ClientHandle Parameter in a Callback

An optional data structure called `ClientHandle` is also passed as an argument to the `adi_rtc_InstallCallback` function. This user-defined structure contains event-specific information to be stored in the callback list entry. When the event occurs, the interrupt manager executes the real-time clock service interrupt handler. The interrupt handler uses the callback list to find the address of the callback and the ClientHandle information that it must pass to the callback.

Coding Example

A PeriodicStopwatch example is installed with VisualDSP++ 5.0. This example demonstrates the use of the functions adi_rtc_InstallCallback and adi_rtc_RemoveCallback for setting up a stopwatch event and servicing it within a callback. It demonstrates the use of the adi_rtc_ResetStopwatch function to re-enable the stopwatch event, and the use of the ClientHandle to creatively control event handling, so that a periodic stopwatch event occurs repeatedly every five seconds for twenty seconds, and then stops.

In the following code example, a data structure is defined, called STOPWATCH_INFO, which has three fields of information that are passed to the callback. The first field indicates that this event is periodic. The second field indicates the number of seconds that the callback function uses when it re-enables the stopwatch event, in this case, five. The third field tells the callback how many times to re-enable the stopwatch event, in this case, three. If this field is set to 0, the event recurs continuously. The ClientHandle argument is used to pass this data structure to the adi_rtc_InstallCallback function.

```
typedef struct
{
  u32 PeriodicFlag;    /* whether the callback shall be periodic
    or one-shot */
  u32 SecondsCounter;  /* value to place in the stopwatch
    count register when re-enabling the stopwatch */
  u32 NumRepetitions;  /* how many times to re-enable the
    stopwatch */
  u32 NumRepetitions;  /* how many times to re-enable the
    stopwatch */
} STOPWATCH_INFO;
```

In the example, a callback function is provided called `RTCCallback` which handles all the real-time clock events. (Only the processing of the `STOP-WATCH` event is shown here.) The callback function accepts three arguments:

```
void   *ClientHandle
u32    EventID
void   *pArg
```

The `ClientHandle` contains the structure that was passed as a parameter to the `adi_rtc_InstallCallback` function when the callback was installed. The interrupt handler passes this argument to the callback function when the interrupt occurs. The `EventID` specifies which of the eight possible events generated the callback, in this case, the stopwatch event. The `pArg` is a flexible argument, defined as type `void*` and reserved for the service to pass any pertinent information the callback needs to process the event. The callback performs a switch on the `EventID` and executes a case statement that proceeds according to the `EventID`. For the stopwatch event, the callback sees that the `periodic` flag is set, so it calls the `adi_rtc_ResetStopwatch` function to re-enable the event, using the `Num-Seconds` value to set the stopwatch counter.

Below is a code example from within the callback function where the stop-watch event is processed.

```
static void RTCCallback ( void *ClientHandle,
                          u32 Event,
                          void *pArg )

{
  STOPWATCH_INFO CountdownStruct =
    *(STOPWATCH_INFO*)ClientHandle ;
```

```
      switch ( (u32)Event )
        {
             /* stopwatch countdown timeout event */
             case ADI_RTC_EVENT_STOPWATCH:
          {
             /* increment the stopwatch event counter */
             SWEventCounter++;

             /* use the fields of the data structure
                to see whether we are done */
             /* if it's a one-shot ...*/
             if ( ( CountdownStruct.PeriodicFlag == 0 )

             /* or if it's periodic... */
             || ( (  CountdownStruct.PeriodicFlag == 1 )

             /* … but the stopwatch event counter reached
                  the number of repetitions  */
             && (SWEventCounter == CountdownStruct.NumRepetitions)

             /* If NumRepetitions were 0, we would repeat forever.
                If the counter wrapped to zero, we would not stop */
             && (CountdownStruct.NumRepetitions != 0 ) ) )

    {
          adi_rtc_RemoveCallback( ADI_RTC_EVENT_STOPWATCH );
          StopwatchCompleteFlag = 1;
          SWEventCounter = 0;
    }
    else
```

```
{
      /* the else condition means it is periodic, and either the
repetitions are unlimited or the count has not yet
reached number of repetitions, so we re-enable the event */
      adi_rtc_ResetStopwatch(  CountdownStruct.SecondsCounter );
}
 /* done */
 break;
}
```

In the main function of the example, a callback is installed for the stopwatch event. It takes a total of 20 seconds for the event to complete. The pertinent sections of the function are shown here.

```
main()
{
    int NumSeconds;
    int* pNumSeconds;

    /* to pass information when installing stopwatch callback */
    STOPWATCH_INFO StopwatchCallbackInfo;

    /* and a pointer to it */
    STOPWATCH_INFO* pStopwatchCallbackInfo;

    /* the example initializes system services including the real
        time clock service*/
    Result = adi_ssl_Init();

    /* zero this flag */
    StopwatchCompleteFlag = 0;

    /* initialize the static stopwatch event counter */
    SWEventCounter = 0;
    /* When re-enabled, the stopwatch will be set for 5 seconds*/
    StopwatchCallbackInfo.SecondsCounter = 5;

    /* stopwatch will happen periodically */
    StopwatchCallbackInfo.PeriodicFlag = 1;

    /* stopwatch will be re-enabled three times */
    StopwatchCallbackInfo.NumRepetitions = 3;

    /* point to the structure which contains the information for
         the callback */
    pStopwatchCallbackInfo = &StopwatchCallbackInfo;

    /* the first stopwatch timeout happens to be the same as the
         reset stopwatch value, 5 seconds */
    NumSeconds = StopwatchCallbackInfo.SecondsCounter;

    /* install the stopwatch callback */
    adi_rtc_InstallCallback(ADI_RTC_EVENT_STOPWATCH,
        (void*)pStopwatchCallbackInfo, NULL, RTCCallback,
          (void*)NumSeconds );

    /* wait for the stopwatch event to occur. After 5 seconds it
         will repeat three times. */
    while( StopwatchCompleteFlag == 0 );

    /* arrive here after 20 seconds */
}
```

RTC Service Application Programming Interface (API)

This section provides the details of the data structures and functions within the RTC service application program interface (API).

Notation and Naming Conventions

To safeguard against conflicts with other software libraries provided by Analog Devices (or other sources), the real-time clock service uses an unambiguous naming convention in which enumeration values and `typedef` statements use the `ADI_RTC_ prefix`. Functions and global variables use the lowercase `adi_rtc_ equivalent`.

Each function within the real-time clock service API returns an error code of the type `ADI_RTC_RESULT`. Like other system services, a return value of zero (`0=ADI_RTC_RESULT_SUCCESS`) indicates that no error occurred during the function call. Any nonzero value indicates the specific type of error that occurred. The error codes for the real-time clock service, unique from those of other system services, are defined in the `adi_rtc.h` include file, so the cause of the error can be determined from looking up the error code in that file.

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

Name – 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

RTC Service API Functions

This section describes the RTC functions that are available to the application. These functions read and write to the hardware registers so the application does not have to study the details of each register. Below is a list of the functions in the RTC API.

- "adi_rtc_Init" on page 11-18
- "adi_rtc_Terminate" on page 11-19
- "adi_rtc_SetDateTime" on page 11-20
- "adi_rtc_GetDateTime" on page 11-21
- "adi_rtc_InstallCallback" on page 11-22
- "adi_rtc_RemoveCallback" on page 11-24
- "adi_rtc_SetEpoch" on page 11-25
- "adi_rtc_GetEpoch" on page 11-26
- "adi_rtc_EnableWakeup" on page 11-27
- "adi_rtc_DisableWakeup" on page 11-28
- "adi_rtc_ResetStopwatch" on page 11-29

Each API function returns a value of type `ADI_RTC_RESULT` which indicates the success or failure of the function call. The result codes are defined in Table 11-5 on page 11-32. This table describes the structures and data types in the API.

adi_rtc_Init

Description

The `adi_rtc_Init()` function initializes the real-time clock service as described in "Initialization" on page 11-3.

Prototype

```
ADI_RTC_RESULT adi_rtc_Init(
    void *pCriticalRegionArg
);
```

Arguments

The function accepts one argument of type `void*`, which is the critical region parameter.

Return Value

`ADI_RTC_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See Table 11-5 on page 11-32 for a list of return codes.

adi_rtc_Terminate

Description

The `adi_rtc_Terminate()` function terminates the real-time clock service as described in "Termination" on page 11-4.

Prototype

```
ADI_RTC_RESULT adi_rtc_Terminate(void);
```

Arguments

The function takes no arguments.

Return Value

`ADI_RTC_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See Table 11-5 on page 11-32 for a list of return codes.

adi_rtc_SetDateTime

Description

The `adi_rtc_SetDateTime()` function is called to program the date and time into the `RTC_STAT` register. The new date and time take effect when the write completes, on the next 1 Hz tick.

Prototype

```
ADI_RTC_RESULT adi_rtc_SetDateTime(
    struct tm *pDateTime
);
```

Arguments

The function accepts one argument, a pointer to the `tm` structure that is defined in the `time.h` file.

Return Value

`ADI_RTC_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See Table 11-5 on page 11-32 for a list of return codes.

adi_rtc_GetDateTime

Description

The `adi_rtc_GetDateTime()` function is called to read the date and time from the `RTC_STAT` register.

Prototype

```
ADI_RTC_RESULT adi_rtc_GetDateTime(
    struct tm *pDateTime
);
```

Arguments

The function takes one argument, a pointer to a `tm` structure.

Return Value

`ADI_RTC_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See Table 11-5 on page 11-32 for a list of return codes.

adi_rtc_InstallCallback

Description

The `adi_rtc_InstallCallback()` function installs a callback for a real-time clock event.

Prototype

```
ADI_RTC_RESULT adi_rtc_InstallCallback(
    ADI_RTC_EVENT_ID      EventID,
    void                  *ClientHandle,
    ADI_DCB_HANDLE        DCBHandle,
    ADI_DCB_CALLBACK_FN   ClientCallback,
    void                  *Value
);
```

Arguments

The caller provides five parameters to the function, shown in Table 11-1.

Table 11-1. Parmeters for the InstallCallback Function

Data Type	Name	Description
`ADI_RTC_EVENT_ID`	`EventID`	Enumerator value that uniquely identifies the event for which the callback is being installed
`void`	`ClientHandle`	Identifier defined and supplied by the application. This value is passed to the callback function.
`ADI_DCB_HANDLE`	`DCBHandle`	The handle returned from the DCB service if callbacks are deferred. NULL when callbacks are “live”.

Table 11-1. Parmeters for the InstallCallback Function (Cont'd)

Data Type	Name	Description
`ADI_DCB_CALLBACK_FN`	`ClientCallback`	Name of client callback function
`void`	Value	RTC service uses this info to install alarm callbacks or stopwatch callbacks. It specifies number of seconds or date/time. For other callbacks, it is not used.

Return Value

`ADI_RTC_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See Table 11-5 on page 11-32 for a list of return codes.

adi_rtc_RemoveCallback

Description

The `adi_rtc_RemoveCallback()` function removes the callback functionality for the specified event. It writes to the `RTC_ICTL` register to disable the event.

Calling `adi_rtc_RemoveCallback` from within a callback routine is not supported and will result in undefined behavior.

Prototype

```
ADI_RTC_RESULT adi_rtc_RemoveCallback(
    ADI_RTC_EVENT_ID EventID
);
```

Arguments

The function takes one parameter, the `EventID`, that specifies which of the eight possible events will have its callback removed. Refer to Table 11-1 on page 11-22 which shows the possible `Event ID` and its meaning.

Return Value

`ADI_RTC_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See Table 11-5 on page 11-32 for a list of return codes.

adi_rtc_SetEpoch

Description

The `adi_rtc_SetEpoch()` function sets the epoch time.

Prototype

```
ADI_RTC_RESULT adi_rtc_SetEpoch(
    ADI_RTC_EPOCH *pEpoch
);
```

Arguments

The function takes one parameter, a pointer to an epoch time `struct` which specifies the new epoch time to use. Refer to Table 11-3 on page 11-31 which shows the fields of the epoch time structure and their meanings.

Return Value

`ADI_RTC_RESULT_SUCCESS`	No error has been encountered.

adi_rtc_GetEpoch

Description

The `adi_rtc_GetEpoch()` function returns the epoch time in the structure provided.

Prototype

```
ADI_RTC_RESULT adi_rtc_GetEpoch(
    ADI_RTC_EPOCH *pEpoch
);
```

Arguments

The function takes one parameter, a pointer to an epoch time `struct`. Refer to Table 11-3 on page 11-31 which shows the fields of the epoch time structure and their meanings.

Return Value

`ADI_RTC_RESULT_SUCCESS`	No error has been encountered.

adi_rtc_EnableWakeup

Description

The `adi_rtc_EnableWakeup()` function calls on the interrupt manager to enable the RTC bit in the system interrupt controller's wakeup register. Subsequently, all enabled RTC interrupting events generate a wake up to the processor.

Prototype

```
ADI_RTC_RESULT adi_rtc_EnableWakeup(void);
```

Arguments

The function accepts no arguments. It calls the `adi_int_SICWakeup` function in the interrupt manager service, passing a `TRUE` flag.

Return Value

`ADI_RTC_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See Table 11-5 on page 11-32 for a list of return codes.

adi_rtc_DisableWakeup

Description

The `adi_rtc_DisableWakeup()` function calls on the interrupt manager to disable the RTC bit in the system interrupt controller's wakeup register. Subsequently, none of the interrupting events wake up the processor.

Prototype

```
ADI_RTC_RESULT adi_rtc_DisableWakeup(void);
```

Arguments

The function accepts no arguments (`void`). It calls the `adi_int_SICWakeup` function in the interrupt manager service, passing a `FALSE` flag.

Return Value

`ADI_RTC_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See Table 11-5 on page 11-32 for a list of return codes.

adi_rtc_ResetStopwatch

Description

The `adi_rtc_ResetStopwatch()` function is used to re-enable a stopwatch event without reinstalling the stopwatch callback function. It can be called from inside or outside the callback, but it cannot be called unless a callback has been installed. The stopwatch event is a “one-shot” event but it may be used as a periodic event by calling this function after the stopwatch event has occurred once and the callback has executed (or is executing). This function sets the `RTC_SWCNT` register with the `NumSeconds` value and re-enables the event by writing to the `RTC_ICTL` register.

Prototype

```
ADI_RTC_RESULT adi_rtc_ResetStopwatch(
    u32
);
```

Arguments

A single value is passed to the function which indicates the number of seconds in the next duration of the stopwatch period.

Return Value

`ADI_RTC_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See Table 11-5 on page 11-32 for a list of return codes.

Real-Time Clock Service API Data Types and Enumerations

This section describes the data structures for reading and writing the date and time to the `RTC_STAT` register, which maintains the current date and time. It also describes the asynchronous event types and the result codes that can be returned from an RTC function.

tm structure

A C-standard time structure is provided in the VisualDSP++ 5.0 system include file, `time.h`. This structure contains fields for the second, minute, hour, day, month, year, plus three additional fields which are not currently used by this RTC service (Table 11-2).

Table 11-2. Fields of the tm Structure

Field Type and Name	Field Description
`int tm_sec`	Seconds after the minute
`int tm_min`	Minutes after the hour
`int tm_hour`	Hour of the day, [0,23]
`int tm_mday`	Day of the month, [1,31]
`int tm_mon`	Months since Jan
`int tm_year`	Years since 1900
`int tm_wday`	Days since Sun, [0,6]
`int tm_yday`	Days since Jan 1 [0,365]
`int tm_isdst`	Daylight savings flag

When using `adi_rtc_SetDateTime()` and `adi_rtc_GetDateTime()`, an epoch date of the first of January 1970 is used. Although this is the earliest date that can be stored using these functions, the year should still be entered using the C standard epoch date of 1900.

For example, the lowest number you can enter for this field is 70. This is because the year, month, and day fields are stored as a 15-bit day count in the `RTC_STAT` register which limits the length of time to 32,768 days (around 89 ½ years).

As with the standard `tm` structure, the `tm_isdst` field is not used. The `adi_rtc_GetDateTime()` function will always return 0 for this field.

This structure is used for compatibility with the `asctime`, `gmtime`, `mktime`, and `ctime` functions described in more detail in the C Run-Time Library reference section of the *VisualDSP++ 5.0 C/C++ Compiler and Library Manual for Blackfin Processors*.

ADI_RTC_EPOCH

This structure contains fields for the year, month, and day. It is used to optionally modify the epoch time, which defaults to January 1, 1970 (Table 11-3).

Table 11-3. ADI_RTC_EPOCH

Field Name	Field Type	Field Description
`year`	`u32`	Epoch year – [0,*]
`month`	`u32`	Epoch month of the year – [1,12]
`day`	`u32`	Epoch day of the month – [1,31]

Event IDs

There are eight possible asynchronous events that are defined by the `Event ID` (Table 11-4).

Table 11-4. Event ID and Description

Event Name	Event Description
`ADI_RTC_EVENT_SECONDS`	(0xA0001) one second periodic interrupt
`ADI_RTC_EVENT_MINUTES`	(0xA0002) one minute periodic interrupt
`ADI_RTC_EVENT_HOURS`	(0xA0003) hourly periodic interrupt
`ADI_RTC_EVENT_DAYS`	(0xA0004) daily periodic interrupt
`ADI_RTC_EVENT_STOPWATCH`	(0xA0005) stopwatch countdown event
`ADI_RTC_EVENT_EACH_DAY_ALM`	(0xA0006) time of day, daily alarm
`ADI_RTC_EVENT_ONCE_ALM`	(0xA0007) day and time once only alarm
`ADI_RTC_EVENT_WRITES_COMPLETE`	(0xA0008) pending register writes complete

Result Codes

Each of the RTCS API functions returns a result code that can be translated by looking into the RTC service API header file, `adi_rtc.h`. The result codes are summarized in Table 11-5.

Table 11-5. RTCS Return Values

Result Code Name	Result Code Description
`ADI_RTC_RESULT_SUCCESS`	Generic success = 0
`ADI_RTC_RESULT_FAILED`	Generic failure = 1
`ADI_RTC_RESULT_START`	(0xA0000) base of event code values
`ADI_RTC_ENUMERATION_START`	(0xA0000) base of event code values
`ADI_RTC_RESULT_INVALID_EVENT_ID`	(0xA0001) invalid `Event ID`

Table 11-5. RTCS Return Values (Cont'd)

Result Code Name	Result Code Description
ADI_RTC_RESULT_INTERRUPT_MANAGER_ERROR	(0xA0002) error from interrupt manager
ADI_RTC_RESULT_ERROR_REMOVING_CALLBACK	(0xA0003) no callback installed for given ID
ADI_RTC_RESULT_CALL_IGNORED	(0xA0004) function not executed
ADI_RTC_RESULT_NOT_INITIALIZED	(0xA0005) RTC service was not initialized
ADI_RTC_RESULT_CALLBACK_NOT_INSTALLED	(0xA0006) callback was never installed
ADI_RTC_RESULT_DATETIME_OUT_OF_RANGE	(0xA0007) tm struct seconds field out of range
ADI_RTC_RESULT_SERVICE_NOT_SUPPORTED	(0xA0008) no RTC service for this processor
ADI_RTC_RESULT_ALREADY_INITIALIZED	(0xA0009) service was already initialized
ADI_RTC_RESULT_CALLBACK_ALREADY_INSTALLED	(0xA000A) cannot install same callback twice
ADI_RTC_RESULT_CALLBACK_CONFLICT	(0xA000B) cannot install both alarm callbacks simultaneously

Interdependencies

This section describes interdependencies between the real-time clock service and other system services.

Interrupt Manager Service

Since the real-time clock service relies on callbacks to process the events, the application must initialize the interrupt manager service before initializing the real-time clock service. During the `adi_rtc_Init()` function, the interrupt manager is called upon to "hook" the real-time clock interrupt handler which executes the callback function. During the `adi_rtc_Terminate()` function, the interrupt manager is called upon to "unhook" the real-time clock interrupt handler.

The application may choose to have real-time clock interrupts generate a wakeup signal to the processor. The real-time clock service provides an API function called `adi_rtc_DisableWakeup()`, which calls the interrupt manager function `adi_int_SICWakeup()`, passing the peripheral ID for the real-time clock, and a `FALSE` flag. The real-time clock service provides an API function called `adi_rtc_EnableWakeup()`, which calls the interrupt manager function `adi_int_SICWakeup()`, passing the peripheral ID for the real-time clock, and a `TRUE` flag, so that all real-time clock interrupting events generate a wakeup signal to the processor.

Deferred Callback Service

Callbacks may be deferred, which means that the execution of the callback is postponed to allow a higher priority thread to execute first. In this case, the callback is not executed by the interrupt handler within the interrupt service routine. Instead, the interrupt handler calls `adi_dcb_Post()` to notify the deferred callback (DCB) manager that the event has occurred. Please refer to Chapter 5, "Deferred Callback Manager" for details on how the DCB manager processes callbacks.

12 FILE SYSTEM SERVICE

This chapter describes the file system service (FSS). The FSS provides access to mass storage media from the Blackfin processor.

This chapter contains:

Introduction

The file system service (FSS) provides a portable and extensible means of accessing embedded mass storage media from the Blackfin processor. It is designed in such a way that support for file systems such as the FAT file system (typically found on removable media such as SD cards and USB flash drive memory sticks) and physical interfaces (such as the ATA/ATAPI interface on the ADSP-BF54x family of processors) can be added to an application with minimal effort. Support for the ADSP-BF548 EZ-KIT Lite development board is provided with VisualDSP++ 5.0 for FAT file systems on the ATA and the secure digital host interfaces for access to the attached hard disk drive and the supplied SD Card and USB flash memory drive.

An application interfaces with these drivers using a framework and API known as the file system service (FSS). Once initialized, the application can make direct calls to the FSS API functions ("File System Service API Reference" on page 12-17), or if registered, using the extensible standard C I/O interface provided with the I/O library of VisualDSP++ 5.0. Once registered with the I/O library, calls to `fopen()`, `fread()`, and so on are routed to the FSS to provide seamless access to files on the mass storage media. In this way, applications using the standard C I/O interface can be readily ported to use the file system service. In addition to these functions, support is provided for certain POSIX functions such as `opendir()`, `rename()`, `remove()`, and so on. The functions available are outlined in "Additional POSIX Functions Supported by the FSS" on page 12-70.

While the FSS API provides support for extending the declaration of file and directory names to use the Unicode UTF-8 (16-bit) specification, the current implementation of the FSS caters only for ASCII file names (8-bit).

Access to each mass storage media type in an application is provided by registering a device driver, termed a physical interface driver (PID), with the FSS. Each media is formatted for a particular file system (for example, the Microsoft® FAT file system) and the interpretation of these file systems is provided by the registration of another type of device driver termed a file system driver (FSD).

Users who wish to provide additional or replacement FSDs and PIDs for either file system or physical interface support can do so simply by following a set of design rules for the appropriate class of device driver and registering that driver with the FSS upon application initialization. Please refer to "Extensibility" on page 12-82 for further details.

The file system service is designed to function in both standalone and RTOS environments. To this end, several examples are available with VisualDSP++ 5.0 to demonstrate the configuration and use of the FSS. One particular example, the shell browser example, is a multi-threaded, VDK-based application that presents a simple, command-line interface to the file system service on a terminal emulator connected to the UART interface of the ADSP-BF548 EZ-KIT Lite. Details of these examples are provided in "Examples" on page 12-82.

Getting Started

The basics for initializing and using the file system service within an application are outlined in this section. It is assumed that the system services and device manager have already been initialized. The dependencies on the system services resources are detailed in "System Service Requirements" on page 12-7.

This section describes the overall operation of the file system service. A more complete description of the application programming interface (API) can be found later in this chapter.

Initialization

The file system service is built on the system services' device driver model and as such requires the use of both the device manager and the system services. It is necessary, therefore, to configure the application for the use of the system services and device manager before initializing the file system service. The application developer should refer to the appropriate documentation for each of the required file system drivers and physical interface drivers in order to determine which resources are required.

The file system service is configured for use by registering the required file system drivers and physical interface drivers, along with the device manager and DMA manager handles, and the handle of a deferred callback queue. This is achieved by passing a command-value pair table to the `adi_fss_Init()` API function. The simplest table is:

```
ADI_FSS_CMD_VALUE_PAIR adi_fss_Config[] = {

 /* Register the ATA/ATAPI interface driver */
 { ADI_FSS_CMD_ADD_DRIVER,         (void*)&ADI_ATAPI_Def },

 /* Register the FAT File System driver */
 { ADI_FSS_CMD_ADD_DRIVER,         (void*)&ADI_FAT_Def },

 /* Assign the DMA Manager Handle */
 { ADI_FSS_CMD_SET_DMA_MGR_HANDLE, (void*)adi_dma_ManagerHandle
},

 /* Assign the Device Manager Handle */
 { ADI_FSS_CMD_SET_DEV_MGR_HANDLE, (void*)adi_dev_ManagerHandle
},

 /* Assign the DCB Queue Handle */
 { ADI_FSS_CMD_SET_DCB_MGR_HANDLE, (void*)adi_dcb_QueueHandle },
 /* Command Table Terminator */
 { ADI_FSS_CMD_END, (void*)NULL }
};
```

The address of this command-value pair table is simply passed to the `adi_fss_Init()` API function as follows:

```
Result = adi_fss_Init( adi_fss_Config );
```

The above command-value pair table configures the FSS to access a hard drive attached to the ATA/ATAPI interface of the ADSP-BF54x processor. The definition structures `ADI_ATAPI_Def` and `ADI_FAT_Def` can either be defined in the application itself, or taken from the device driver header files. To do the latter, a macro must be defined ahead of the `#include` statement. The following is the quickest way to get up and running with the above command-value pair table:

```
/* FAT12/16/32 FSD driver */
#define _ADI_FAT_DEFAULT_DEF_
#include <drivers/fsd/fat/adi_fat.h>

/* ATAPI interface */
#define _ADI_ATAPI_DEFAULT_DEF_
#include <drivers/pid/atapi/adi_atapi.h>
```

To register an additional driver with the FSS, simply add the appropriate header to the file containing the above code, choose to use the default definition or supply your own, and add the following command-value pair to the FSS configuration table:

```
      { ADI_FSS_CMD_ADD_DRIVER,(void*)&<Device-Def-Structure> },
```

Please note that if the application makes simultaneous use of two or more types of media that require the use of the same file system driver, then only one `ADI_FSS_CMD_ADD_DRIVER` entry is required in the FSS configuration table for the appropriate FSD. (For example, the ADSP-BF548 EZ-KIT Lite supports access to the hard disk, an SD Card, and a USB flash memory drive, all of which can make use of the FAT file system driver.)

The complete set of commands that can be used in the initialization of the file system service is found in the description of "adi_fss_Init" on page 12-19.

The file system service is now ready for use through its direct API. However, for greater portability of applications and libraries, the FSS is best used indirectly through the standard C I/O interface. VisualDSP++ 5.0 provides the ability to extend the file I/O support beyond the host PC access (PRIMIO) that is the default. This is achieved quite simply by including the supplied FSS header file:

```
#include <services/fss/adi_fss.h>
```

and registering the required entry point structure (of type `DevEntry`) with the I/O library:

```
add_devtab_entry( &adi_fss_entry );
```

The definition of the `DevEntry` structure can be found in Chapter 3 of the *VisualDSP++ 5.0 C/C++ Compiler and Library Manual for Blackfin Processors*. The declaration of `adi_fss_entry` can be found in the file `adi_fss_deventry.c` in the `Blackfin\lib\src\services\fss` folder of the VisualDSP++ 5.0 installation.

Termination

When the application no longer requires the file system service, a call to the termination function, `adi_fss_Terminate()` can be made. This function unmounts all mounted file systems, closes all PIDs and FSDs, and frees all dynamically-allocated memory.

System Service Requirements

It was stated in "Getting Started" on page 12-3 that the file system service is dependent on the appropriate configuration of the system services and device manager. The recommended way to get started is to make a copy of the `adi_ssl_init.c` and `adi_ssl_init.h` files found in the FSS example folders and use them as a basis for your own applications. For example, see the sample files in:

```
Blackfin\Examples\ADSP-BF548 EZ-KIT Lite\Services\File
System\HardDisk\HardDiskAccess
```

The `adi_ssl_init.c` file provides the basic sequence of initialization calls to each of the system services and the device manager, supplying each service with the memory they require. The `adi_ssl_init.h` file contains the definitions for the constants relevant to the implementation of the FSS and is the only file of the two that requires modification. In this file, the following macros are defined to reflect the requirements of the application:

```
#define ADI_SSL_INT_NUM_SECONDARY_HANDLERS  (2)
#define ADI_SSL_DCB_NUM_SERVERS             (1)
#define ADI_SSL_DMA_NUM_CHANNELS            (2)
#define ADI_SSL_FLAG_NUM_CALLBACKS          (0)
#define ADI_SSL_SEM_NUM_SEMAPHORES          (10)
#define ADI_SSL_DEV_NUM_DEVICES             (2)
```

The values to use depends not only on the FSS configuration, that is, which file system and physical interface drivers are registered, but also on which other device drivers and services are used in the application, for example the LCD display driver, an audio codec driver, and so on.

The following sections address each of these quantities in turn, but only with respect to the file system service configuration.

Interrupt Manager Service

The interrupt manager is one of the core system services' modules that underpins the implementation of the device drivers of which the physical interface drivers (PID) are no exception. Each PID has a number of peripheral interrupts associated with it. For example, the ATAPI driver has three DMA interrupts and one device interrupt. Depending on whether you choose to use the default IVG levels for the appropriate interrupts or assign custom levels determines how many secondary interrupts are required in the application. All device drivers hook interrupt handlers to the IVG level as identified in the `SIC_IARx` registers. You may find it useful to explicitly set the required priorities with a series of calls to the `adi_int_SICSetIVG()` API function for all peripherals in your application. For example, to set the ATAPI priorities you would do the following:

```
adi_int_SICSetIVG(ADI_INT_DMA10_ATAPI_RX, 10);
adi_int_SICSetIVG(ADI_INT_DMA11_ATAPI_TX, 11);
adi_int_SICSetIVG(ADI_INT_ATAPI_ERROR, 7);
adi_int_SICSetIVG(ADI_INT_DMA_ERROR, 7);
```

In this way, you are able to see at a glance which IVG levels are used by multiple peripherals. You can then determine the number of secondary handlers (`ADI_SSL_INT_NUM_SECONDARY_HANDLERS`) to use.

Please refer to the device driver documentation for each PID required by your application for details of the appropriate interrupt requirements. These documents are found under the `Blackfin\Docs\drivers\pid` folder in the VisualDSP++ 5.0 installation.

Deferred Callback Service

The file system service is best operated using deferred callbacks. For operation within VDK it is essential that the callbacks are deferred. While multiple callback servers can be used in an application, it is recommended that just one server is used and is configured to operate at IVG level 14. In fact, for use within the VDK environment, only one server is allowed and this server makes use of the *Deferred Procedure Queue* in VDK that is executed at *Kernel* level (IVG 14).

Therefore, it is recommended that you set the value of the `ADI_SSL_DCB_NUM_SERVERS` macro to 1.

In addition to the code in the `adi_ssl_init.c` file, the following is required to *open* a queue server:

```
adi_dcb_Open( 14, DCBMgrQueue1, SIZE_DCB_QUEUE, &ResponseCount,
&adi_dcb_QueueHandle );
```

where `DCBMgrQueue1` is the address of a block of memory of size at least `SIZE_DCB_QUEUE`. This code is extracted from the `InitServices.c` file in the HardDiskAccess example. The size of the queue depends on your requirements but a good starting point is to use four entries in the queue:

```
#define SIZE_DCB_QUEUE ((ADI_DCB_ENTRY_SIZE)*4)
```

DMA Service

The DMA manager is used by the file system service to transfer data between the application and the physical media, where appropriate, as determined by the nature of each PID in the application. The use of DMA and the channels affected are detailed in the appropriate PID device driver documents. The value given to the `ADI_SSL_DMA_NUM_CHANNELS` must be incremented by the number of DMA channels required by each PID.

The DMA requirements for each physical device driver are detailed in the documentation for that appropriate driver. For example, the ATAPI driver requires the DMA manager to be initialized for two DMA channels; the SD PID, one channel; and the USB host driver, no DMA channels.

Semaphore Service

The file system service requires a number of semaphores for its operation. The semaphores are used to maintain atomic access to file system and physical interface drivers and for notification of completion of data transfers. For each instance of an FSD or PID, at least two semaphores are required. (See the appropriate drive documentation for precise details.) Please note that if the application makes simultaneous use of two or more types of media that require the use of the same file system driver, then two semaphores are required for each instantiation of the appropriate FSD driver.

The number of semaphores required also varies in proportion to the number of files that are opened at one time, as the use of the file cache requires one semaphore per open file.

The value given to the `ADI_SSL_SEM_NUM_SEMAPHORES` must be incremented by the number of semaphores required for the required FSS configuration.

When the FSS is used in the VDK environment, this value is actually ignored by the semaphore service; however, you need to assign the appropriate number to the maximum active semaphores value in the kernel tab of your application project.

Real-Time Clock Service

The dependency on the real-time clock (RTC) service applies only if a file system driver, included in an application, makes use of the RTC to supply time and date details to be entered in the directory entries of new and modified files. This is the case with the FAT file system driver supplied with VisualDSP++5.0.

Initialization of the real-time clock service and how to set the current date and time are detailed in Chapter 11, “Real-Time Clock Service”.

Device Manager

Each PID and FSD registered by the application represents at least one device driver in the application; some PIDs such as the USB mass storage class driver may require two or more. Additionally, multiple instances of an FSD represent separate device drivers.

The value given to the `ADI_SSL_DEV_NUM_DEVICES` must be incremented by the number of PID and FSD instances appropriate for the required FSS configuration. For more details, see the appropriate device driver documentation in the folders, `Blackfin\Docs\drivers\fsd` and `Blackfin\Docs\drivers\pid`.

Advanced Configuration

This section describes how to go beyond the basics and tailor the initialization of the file system service to suit your requirements.

Custom Configuration of Device Drivers

In "Getting Started" on page 12-3, the FSD and PID definition structures used were those found in the appropriate header files for the drivers. However, there will be occasions when your application requires a customized definition, such as when additional commands need to be added to the driver's configuration table.

The device definition structures are of type `ADI_FSS_DEVICE_DEF` which is described in "File System Service API Data Types and Enumerations" on page 12-52. This structure is defined in the FSS header file, `<services/fss/adi_fss.h>`.

The command table is an array of command-value pairs (type `ADI_DEV_CMD_VALUE_PAIR`) that are usual for device drivers that conform to the system services' device driver model.

Valid commands for the driver configuration table are presented in the individual driver documents, found in the VisualDSP++ 5.0 installation in the appropriate sub-folder under `Blackfin\Docs\drivers\fsd` or `Blackfin\Docs\drivers\pid`. An example would be to assign a heap index for the allocation of DMA buffers (see "Dynamic Memory Usage" on page 12-13):

```
ADI_DEV_CMD_VALUE_PAIR ADI_FAT_ConfigTable [] = {
        { ADI_FSS_CMD_SET_CACHE_HEAP_ID,   (void*)2 },
        { ADI_DEV_CMD_END,                 NULL     },
};
```

In addition to being able to customize the definition of a PID, it is possible to register a PID that has already been opened and configured. This facility is primarily provided to enable the swapping of mass storage media between the FSS and other elements of an application. For example embedded NAND flash, hard disk, or SD card could be swapped between the FSS and the USB mass storage (device) class driver to enable access to the embedded mass storage from a PC or Mac.

Before the PID can be registered in this way it must be:

- **Opened:** The values in the `ADI_FSS_DEVICE_DEF` structure defined in the PID header file can be used in an explicit call to `adi_dev_Open`. For example, the members of the ATAPI device definition structure can be used as follows:

  ```
  adi_dev_Open (adi_dev_ManagerHandle,
                ADI_ATAPI_Def.pEntryPoint,
                ADI_ATAPI_Def.DeviceNumber,
                <client handle>,
                &ADI_ATAPI_Def.DeviceHandle,
                ADI_ATAPI_Def.Direction,
                adi_dma_ManagerHandle,
                adi_dcb_QueueHandle,
                <client-callback-function>);
  ```

Please note that the `<client-handle>` and `<client call-back-function>` arguments only have meaning prior to registration of the PID with the file system service; these items are reassigned within the FSS. If they have no direct relevance to the application itself, they can be assigned NULL and 1 respectively.

- **Data Flow Method Assigned:** The data flow method of the PID must be set to `ADI_DEV_MODE_CHAINED` with the following command-value pair:

  ```
  {ADI_DEV_CMD_SET_DATAFLOW_METHOD,(void *)
  ADI_DEV_MODE_CHAINED}
  ```

- **Configured:** In addition to setting the data flow method, the PID should be configured at the very least with the commands defined in the command table whose location is given by the `pConfigTable` member of the appropriate device definition structure.

Registration of the PID can either be accomplished by passing the following command-value pair to `adi_fss_Control`:

```
{ADI_FSS_CMD_REGISTER_DEVICE,(void *)&ADI_ATAPI_Def}
```

or by using the `adi_fss_RegisterDevice` function:

```
{adi_fss_RegisterDevice(&ADI_ATAPI_Def,1);
```

For further details, see the definition of this function in the section "File System Service API Reference" on page 12-17.

Dynamic Memory Usage

In a departure from other system services and device drivers, the file system service makes extensive use of dynamically-allocated memory for internal data structures and DMA buffers.

This is largely due to the somewhat arbitrary and indeterminate nature of the memory requirements associated with file systems, where buffer sizes depend on the media present and the file system used. The use of dynamic memory within all parts of the FSS framework frees the application developer from having to estimate the amount of memory required at build time.

However, in order to give the application developer some control, all dynamic memory requests throughout the FSS and its component file system and physical interface drivers are made through centralized functions, which then make calls into the C library functions or can be directed to call custom functions for either greater efficiency or monitoring purposes. There are three heap allocation functions for `malloc`, `realloc`, and `free` operations, and all three must be replaced with custom functions if any are to be replaced. To use your own functions, ensure that the functions you wish to use conform to the following prototypes for each operation.

```
malloc:
void *<custom-heap_malloc-function>( int, size_t );

realloc:
void *<custom-heap_realloc-function>( int, void *, size_t );

free:
void  <custom-heap_free-function>( int, void * );
```

These are then required to be registered with the file system service using the following command-value pairs:

```
{ ADI_FSS_CMD_SET_MALLOC_FUNC,
   (void*)<custom-heap_malloc-function>  },
{ ADI_FSS_CMD_SET_REALLOC_FUNC,
   (void*)<custom-heap_realloc-function> },
{ ADI_FSS_CMD_SET_FREE_FUNC,
   (void*)<custom-heap_free-function>    },
```

Furthermore, memory allocation is divided into two categories of memory requirement: *general* memory is defined as the memory required for data structures such as device driver instance data and file descriptors, and so on, and *cache* memory is defined as the memory required for cache-type buffers such as DMA transfer buffers. Management of these categories of memory is implemented by the use of custom (or user) heaps, available with VisualDSP++ 5.0. Each component of the FSS framework allocates general memory off the single *general* heap defined in the FSS. For cache memory requirements, each component can be assigned a separate *cache* heap to use.

Two commands are defined to assign the indexes of general and cache heaps. To assign the general heap index, use the following command-value pair in the FSS configuration table passed to the `adi_fss_Init()` function only:

```
  { ADI_FSS_CMD_SET_GENERAL_HEAP_ID, (void*)GeneralHeap },
```

The `GeneralHeap` value is the array index of the heap entry in the `heap_table`. For example, in "Shell_Browser" on page 12-85, the `GeneralHeap` value is assigned to 1 which is the index of the `FSSGeneralHeap_space` entry in the following table[1]:

```
struct heap_table_t heap_table[4] =
{
  { &ldf_heap_space, (int) &ldf_heap_length, 0 },
  { &FSSGeneralHeap_space, (int) &FSSGeneralHeap_length, 1 },
  { &FSSCacheHeap_space, (int) &FSSCacheHeap_length, 2 },
  { 0, 0, 0 }
};
```

[1] Please note that this is also the userid value in the table entry, which is a coincidental consequence of the above entries being auto-generated by the project options wizard. For user heaps defined with heap_install, this may not be the case.

This index can also be obtained from a call to `heap_lookup()` for the `userid` required:

```
GeneralHeap = heap_lookup(1);
```

To assign a cache heap use:

```
{ ADI_FSS_CMD_SET_CACHE_HEAP_ID, (void*)CacheHeap },
```

which can be added to the configuration table of each driver and the `adi_fss_Init()` function. In the latter case, it is used to assign the heap index for allocation of each file cache buffer. See "File Cache" below for further details.

The default case is for all cache heaps to default to the general heap, and the general heap to default to the system heap. This is the case for the configuration shown in "Getting Started" on page 12-3. Please note that while it is perfectly valid to use the same cache heap index for each driver (FSD or PID), the heap index is required to be registered with each device driver through its configuration table.

File Cache

The file system service is designed to use a file cache when appropriate. This cache, which is allocated from the FSS cache heap, ("Dynamic Memory Usage" on page 12-13), acts as a read-ahead, write-behind cache so as to enhance data transfer rates in either direction. To achieve higher transfer rates, cache blocks are transferred to or from the media in the background, and its use is thus optimal for physical interface drivers that use peripheral DMA. The decision to use a file cache for a file is taken by the FSS upon opening a file, when the respective file system driver and physical interface drivers are queried to determine whether a file cache can be supported. Please refer to the appropriate FSD and PID documentation for details. For example, the FAT and ATAPI drivers for access to the hard disk on the ADSP-BF548 EZ-KIT Lite support the use of a file cache.

Each file cache comprises a number of blocks, each the size of the smallest addressable unit within the respective file system. For example, in the FAT file system, the block size is that of a cluster. The cluster size depends on the size and format (for example, FAT12, FAT16 or FAT32) of the media. The FSS requests the block size from the appropriate FSD upon opening a file. The number of cache blocks to use defaults to four (4), but can be overridden upon initialization of the FSS with the inclusion of the following command-value pair in the FSS configuration table:

```
{ ADI_FSS_CMD_SET_NUMBER_CACHE_BLOCKS,
   (void*)<number-cache-blocks> },
```

When a file is opened, the file cache is dynamically allocated from the FSS cache heap; when the file is closed, the memory is freed.

Please note that for the default format of the hard disk of the ADSP-BF548 EZ-KIT Lite, a cluster—and hence each cache block—is 16 KB in size.

File System Service API Reference

This section provides the details of the data structures and functions within the FSS or file system service application program interface (API). It is usually intended that the API be used via intermediate code such as the extensible. Its use may provide marginal benefits in terms of throughput speed and code size, but this is offset by its lack of portability between development environments.

Notation and Naming Conventions

To safeguard against conflicts with other software libraries provided by Analog Devices, or other sources, the file system service uses an unambiguous naming convention in which enumeration values and `typedef` statements use the `ADI_FSS_` prefix. Functions and global variables use the lowercase `adi_fss_` equivalent.

Each function within the file system service API returns an error code of type `ADI_FSS_RESULT`. Like the other system services, a return value of zero (`ADI_FSS_RESULT_SUCCESS`) indicates that no error has occurred during the function call. Any nonzero value indicates the specific type of error that has occurred. The error codes for the file system service are unique from those of the other system services, and are defined in the `adi_fss.h` header file and in "File System Service API Data Types and Enumerations" on page 12-52. The cause of the error can be determined by looking up the error code in the file. The reference pages for the API functions use the following format:

Name – 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_fss_Init

Description

The `adi_fss_Init()` function initializes the file system service and prepares it for operation. Configuration is effected by supplying a table of command-value pairs. The only mandatory commands are:

`ADI_FSS_CMD_ADD_DRIVER`	(0xB0005) Sets the location of the `ADI_FSS_DEVICE_DEF` structure defining either a file system driver (FSD) or physical interface driver (PID). This command is mandatory only for FSDs.
`ADI_FSS_CMD_SET_DEV_MGR_HANDLE`	(0xB000E) Sets device manager handle

In addition to these commands, others may also be used as appropriate for the physical interface and file system drivers present in the overall configuration. Please refer to the relevant driver documentation for further details. The optional commands are:

`ADI_FSS_CMD_SET_DMA_MGR_HANDLE`	(0xB000D) Sets DMA manager handle. It is required only if a PID uses peripheral DMA.
`ADI_FSS_CMD_SET_DCB_MGR_HANDLE`	(0xB000F) Sets DCB queue manager handle. It is strongly recommended to use deferred callbacks in a standalone application, and compulsory in a VDK application.
`ADI_FSS_CMD_SET_CACHE_HEAP_ID`	(0xB0011) Heap ID for cache blocks
`ADI_FSS_CMD_SET_GENERAL_HEAP_ID`	(0xB0014) Heap ID for file descriptors
`ADI_FSS_CMD_SET_NUMBER_CACHE_BLOCKS`	(0xB0012) Number of cache blocks to use (min 2)
`ADI_FSS_CMD_SET_NUMBER_CACHE_SUB_BLOCKS`	(0xB0013) Number of cache sub-blocks to use (min 1). It is highly recommended not to alter this value.
`ADI_FSS_CMD_SET_MALLOC_FUNC`	(0xB0006) Sets client `malloc` function
`ADI_FSS_CMD_SET_REALLOC_FUNC`	(0xB0007) Sets client `realloc` function

`ADI_FSS_CMD_SET_FREE_FUNC`	(0xB0008) Sets client `free` function
`ADI_FSS_CMD_SET_VOLUME_SEPARATOR`	(0xB0009) Sets volume separator character
`ADI_FSS_CMD_SET_DIRECTORY_SEPARATOR`	(0xB000A) Sets directory separator character
`ADI_FSS_CMD_SET_MEDIA_CHANGE_CALLBACK`	(0xB001B) Sets the `ADI_DCB_CALLBACK_FN` callback function that is called when the media is changed; for example, when a USB flash drive or SD card is inserted or removed.
`ADI_FSS_CMD_SET_MEDIA_CHANGE_HANDLE`	(0xB001C) Sets the handle that is sent as the first argument to the callback function upon media change.
`ADI_FSS_CMD_SET_DATA_SEMAPHORE_TIMEOUT`	(0xB001D) Sets the timeout argument for all semaphore pends within the context of the FSS. This value must be the logical OR of the timeout value in ticks and the appropriate value to prevent an RTOS error condition on timeout.
`ADI_FSS_CMD_SET_TRANSFER_RETRY_COUNT`	(0xB001E) Sets the number of times a transfer is retried after a semaphore timeout. The value includes the initial attempt, so a value of one means no retries after the timeout. This is the default.
`ADI_FSS_CMD_REGISTER_DEVICE`	(0xB0019) Registers a physical interface driver by specifying the location of the appropriate `ADI_FSS_DEVICE_DEF` structure. Prior to using this command, the appropriate device driver must be opened and configured.

Prototype

```
u32 adi_fss_Init(
    ADI_FSS_CMD_VALUE_PAIR *pTable
);
```

Arguments

`pTable`	Pointer to table of command-pair values used to initialize the FSS module

Return Value

`ADI_FSS_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See Table 12-1 on page 12-55 for a list of return codes.

adi_fss_Terminate

Description

The `adi_fss_Terminate()` function terminates the file system service.

Prototype

```
u32 adi_fss_Terminate(void);
```

Arguments

The function takes no arguments.

Return Value

`ADI_FSS_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See Table 12-1 on page 12-55 for a list of return codes.

adi_fss_Control

Description

The `adi_fss_Control()` function is called to send specific commands to the FSS service module for control of operations. Valid command codes are:

Command Code Name	Command Code Description
`ADI_FSS_CMD_END`	(0xB0002) End of table of control-command pairs
`ADI_FSS_CMD_PAIR`	(0xB0003) Value is pointer to control-command pair
`ADI_FSS_CMD_TABLE`	(0xB0004) Value is pointer to table of control-command pairs
`ADI_FSS_CMD_GET_NUMBER_VOLUMES`	(0xB000B) Gets the number of available partitions
`ADI_FSS_CMD_GET_VOLUME_INFO`	(0xB000C) Gets information regarding available partitions
`ADI_FSS_CMD_FORMAT_VOLUME`	(0xB4017) Formats a volume
`ADI_FSS_CMD_REGISTER_DEVICE`	(0xB0019) Registers a physical interface driver by specifying the location of the appropriate `ADI_FSS_DEVICE_DEF` structure. Prior to using this command, the appropriate device driver must be opened and configured.
`ADI_FSS_CMD_DEREGISTER_DEVICE`	(0xB001A) Deregisters the handle of a physical interface device.
`ADI_FSS_CMD_SET_MEDIA_CHANGE_CALLBACK`	(0xB001B) Sets the `ADI_DCB_CALLBACK_FN` callback function that is called when the media is changed; for example, when a USB flash drive or SD card is inserted or removed.
`ADI_FSS_CMD_SET_MEDIA_CHANGE_HANDLE`	(0xB001C) Sets the handle that is sent as the first argument to the callback function upon media change.

Command Code Name	Command Code Description
`ADI_FSS_CMD_SET_DATA_SEMAPHORE_TIMEOUT`	(0xB001D) Sets the timeout argument for all semaphore pends within the context of the FSS. This value must be the logical OR of the timeout value in ticks and the appropriate value to prevent an RTOS error condition on timeout.
`ADI_FSS_CMD_SET_TRANSFER_RETRY_COUNT`	(0xB001E) Sets the number of times a transfer is retried after a semaphore timeout. The value includes the initial attempt, so a value of one means no retries after the timeout. This is the default. After all attempts have failed, the FSS reports the error back to the application.

Prototype

```
u32 adi_fss_Control(
                    u32     CommandID,
                    void    *Value
);
```

Arguments

`CommandID`	Command ID
`Value`	Pointer to command-specific value

Return Value

`ADI_FSS_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See Table 12-1 on page 12-55 for a list of return codes.

adi_fss_RegisterDevice

Description

The `adi_fss_RegisterDevice()` function is called to register a physical interface driver (PID) with the FSS. Upon registration, the PID is activated. Media detection can also be performed dependent on the value of the `PollForMedia` argument.

Prototype

```
u32 adi_fss_RegisterDevice(
        ADI_FSS_DEVICE_DEF     *pDeviceDef,
        u32                    PollForMedia
);
```

Arguments

`pDeviceDef`	The location of the device definition structure for the required physical interface driver
`PollForMedia`	Flag to determine whether to poll for media after activation. Set 1 to poll, 0 otherwise. If polling is not chosen, the application must explicitly poll for media

Return Value

`ADI_FSS_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See Table 12-1 on page 12-55 for a list of return codes.

adi_fss_DeRegisterDevice

Description

The `adi_fss_DeRegisterDevice()` function is called to deregister a physical interface driver (PID) with the FSS. All associated volumes will be unmounted and the device deactivated.

Prototype

```
u32 adi_fss_DeRegisterDevice(
        ADI_DEV_DEVICE_HANDLE     DeviceHandle
);
```

Arguments

`DeviceHandle`	The device handle of the physical interface driver that is to be deregistered with the FSS

Return Value

`ADI_FSS_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See Table 12-1 on page 12-55 for a list of return codes.

adi_fss_PollMediaOnDevice

Description

The `adi_fss_PollMediaOnDevice()` function is called to poll the media associated with the given PID device handle.

Prototype

```
u32 adi_fss_PollMediaOnDevice(
        ADI_DEV_DEVICE_HANDLE     DeviceHandle
);
```

Arguments

`DeviceHandle`	The device handle of the required physical interface driver

Return Value

`ADI_FSS_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See Table 12-1 on page 12-55 for a list of return codes.

adi_fss_PollMedia

Description

The `adi_fss_PollMedia()` function is called to poll the physical interface devices (PID) for changes in media.

Prototype

```
u32 adi_fss_PollMedia(void);
```

Arguments

The function takes no arguments.

Return Value

`ADI_FSS_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See Table 12-1 on page 12-55 for a list of return codes.

adi_fss_Stat

Description

The `adi_fss_Stat()` function is called to get the status of a file or directory. The name and name length of the file or directory are passed to this function, as well as a data structure in which the function stores the information about the file or directory.

Prototype

```
u32 adi_fss_Stat(
                ADI_FSS_WCHAR          *name,
                u32                    namelen,
                struct stat            *pStat
);
```

Arguments

`name`	Array to store Unicode UTF-8 name of file or directory
`namelen`	Length of name array
`pStat`	Address of structure to hold information

Return Value

`ADI_FSS_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See Table 12-1 on page 12-55 for a list of return codes.

adi_fss_UnMountDevice

Description

The `adi_fss_UnMountDevice()` function is called to unmount all volumes on the media associated with the given PID device handle. The device remains registered with the FSS.

Prototype

```
u32 adi_fss_UnMountDevice(
        ADI_DEV_DEVICE_HANDLE     DeviceHandle
);
```

Arguments

`DeviceHandle`	The device handle of the required physical interface driver

Return Value

`ADI_FSS_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See Table 12-1 on page 12-55 for a list of return codes.

adi_fss_FileOpen

Description

The `adi_fss_FileOpen()` function opens a file stream for reading or writing. The name argument can be specified as an absolute or relative path. Valid mode values are given in the following table. Combinations of mode values may be required. For example, to open a file for append would require the mode to be specified as `ADI_FSS_MODE_WRITE | ADI_FSS_MODE_APPEND`.

`ADI_FSS_MODE_READ`	Read-only access
`ADI_FSS_MODE_WRITE`	Write access only
`ADI_FSS_MODE_READ_WRITE`	Read and write access
`ADI_FSS_MODE_APPEND`	Append mode
`ADI_FSS_MODE_CREATE`	Create file if not found
`ADI_FSS_MODE_TRUNCATE`	Truncate file if existing file opened for write

Prototype

```
u32 adi_fss_FileOpen(
                    ADI_FSS_WCHAR        *name,
                    u32                  namelen,
                    u32                  mode,
                    ADI_FSS_FILE_HANDLE  *FileHandle );
```

Arguments

`name`	NULL-terminated string identifying file to open
`namelen`	Length of name string excluding NULL; for example, the value that the `strlen(name)` would return
`mode`	Mode in which file is opened
`FileHandle`	Location to store handle identifying file stream

Return Value

`ADI_FSS_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See Table 12-1 on page 12-55 for a list of return codes.

adi_fss_FileClose

Description

The `adi_fss_FileClose()` function closes the file stream identified by `FileHandle`.

Prototype

```
u32 adi_fss_FileClose(ADI_FSS_FILE_HANDLE FileHandle);
```

Arguments

`FileHandle`	Handle identifying file stream

Return Value

`ADI_FSS_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See Table 12-1 on page 12-55 for a list of return codes.

adi_fss_FileWrite

Description

The `adi_fss_FileWrite()` function writes the `size` number of bytes of data identified by `buf` to a file stream identified by `FileHandle` and returns the number of bytes written to the location pointed at by the `BytesWritten` pointer.

Prototype

```
u32 adi_fss_FileWrite(
                      ADI_FSS_FILE_HANDLE FileHandle,
                      u8          *buf,
                      u32         size,
                      u32         *BytesWritten
 );
```

Arguments

`FileHandle`	Handle identifying file stream
`buf`	Start address of buffer to write
`size`	Number of bytes to write
`BytesWritten`	Location to store actual size written

Return Value

`ADI_FSS_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See Table 12-1 on page 12-55 for a list of return codes.

adi_fss_FileRead

Description

The adi_fss_FileRead function reads data from a file stream identified by FileHandle, places it in the buffer identified by buf, and returns the number of bytes read into the location pointed to by the BytesRead pointer.

Prototype

```
u32 adi_fss_FileRead(
                    ADI_FSS_FILE_HANDLE FileHandle,
                    u8          *buf,
                    u32         size,
                    u32         *BytesRead
 );
```

Arguments

FileHandle	Handle identifying file stream
buf	Start address of buffer to fill
size	Number of bytes to read
BytesRead	Location to store actual size read

Return Value

ADI_FSS_RESULT_SUCCESS	No error has been encountered.
Any other value	Error has occurred. See Table 12-1 on page 12-55 for a list of return codes.

adi_fss_FileSeek

Description

The `adi_fss_FileSeek` function moves the current file position to the location defined by the `Offset` and `Origin` arguments.

The `Origin` value should have one of the following values (defined in `stdio.h`):

`SEEK_SET`	Seek from the start of the file
`SEEK_CUR`	Seek from the current location
`SEEK_END`	Seek from the end of the file

You can use `adi_fss_FileSeek` to move beyond the end of a file, but not before the beginning. Using `adi_fss_FileSeek` clears the EOF flag associated with that stream.

Prototype

```
u32 adi_fss_FileSeek(
                     ADI_FSS_FILE_HANDLE FileHandle,
                     s32         offset,
                     u32         whence,
                     u32         *tellpos
);
```

Arguments

`FileHandle`	Handle identifying file stream
`offset`	Offset (in bytes) from Origin location
`whence`	Location to begin seek from
`tellpos`	Location to store current position after seek

Return Value

`ADI_FSS_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See Table 12-1 on page 12-55 for a list of return codes.

adi_fss_FileTell

Description

The `adi_fss_FileTell` function reports the current position in the file.

Prototype

```
u32 adi_fss_FileTell(
                     ADI_FSS_FILE_HANDLE   FileHandle
                     u32                   *tellpos
);
```

Arguments

`FileHandle`	Handle identifying file stream
`tellpos`	Location to store current position

Return Value

`ADI_FSS_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See Table 12-1 on page 12-55 for a list of return codes.

adi_fss_IsEOF

Description

The `adi_fss_IsEOF()` function determines if the end of the file on the indicated data stream has been reached.

Prototype

```
u32 adi_fss_IsEOF(ADI_FSS_FILE_HANDLE FileHandle);
```

Arguments

`FileHandle`	Handle identifying file stream

Return Value

`ADI_FSS_RESULT_SUCCESS`	No error has been encountered.
`ADI_FSS_RESULT_EOF`	End of file has been reached.
Any other value	Error has occurred. See Table 12-1 on page 12-55 for a list of return codes.

adi_fss_FileRemove

Description

The adi_fss_FileRemove() routine removes the specified file. The name argument can be specified as an absolute or relative path.

Prototype

```
u32 adi_fss_FileRemove(
                       ADI_FSS_WCHAR   *name,
                       u32             namelen
);
```

Arguments

name	NULL-terminated string identifying file to remove
namelen	Length of name string excluding terminating NULL; for example, the value that strlen(name) would return

Return Value

ADI_FSS_RESULT_SUCCESS	No error has been encountered.
Any other value	Error has occurred. See Table 12-1 on page 12-55 for a list of return codes.

adi_fss_FileRename

Description

The `adi_fss_FileRename()` routine renames a file from `old_path` to `new_path`. Both arguments can be specified as absolute or relative paths.

Prototype

```
u32 adi_fss_FileRename(
                      ADI_FSS_WCHAR   *old_path,
                      u32             oldlen,
                      ADI_FSS_WCHAR   *new_path,
                      u32             newlen
);
```

Arguments

`old_path`	NULL-terminated string identifying file to rename
`oldlen`	Length of `old_path` string excluding terminating NULL; for example, the value that `strlen(name)` would return
`new_path`	NULL-terminated string identifying new file name
`newlen`	Length of `new_path` string excluding NULL; for example, the value that `strlen(name)` would return

Return Value

`ADI_FSS_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See Table 12-1 on page 12-55 for a list of return codes.

adi_fss_DirOpen

Description

The `adi_fss_DirOpen()` routine opens the specified directory stream. The `name` argument can be specified as an absolute or relative path.

Prototype

```
u32 adi_fss_DirOpen(
                    ADI_FSS_WCHAR      *name,
                    u32                namelen,
                    ADI_FSS_DIR_HANDLE *DirHandle
);
```

Arguments

`name`	NULL-terminated string identifying the directory to open
`namelen`	Length of name string excluding terminating NULL; for example, the value that `strlen(name)` would return
`DirHandle`	Location to store handle identifying directory stream

Return Value

`ADI_FSS_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See Table 12-1 on page 12-55 for a list of return codes.

adi_fss_DirClose

Description

The `adi_fss_DirClose()` routine closes the identified directory stream.

Prototype

```
u32 adi_fss_DirClose(ADI_FSS_DIR_HANDLE DirHandle);
```

Arguments

`DirHandle`	Handle identifying directory stream

Return Value

`ADI_FSS_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See Table 12-1 on page 12-55 for a list of return codes.

adi_fss_DirRead

Description

The `adi_fss_DirRead()` routine returns a pointer to the next directory entry from the directory stream.

Prototype

```
u32 adi_fss_DirRead(
                    ADI_FSS_DIR_HANDLE   DirHandle,
                    ADI_FSS_DIR_ENTRY    **pDirEntry
);
```

Arguments

`DirHandle`	Handle identifying directory stream
`pDirEntry`	Location to store pointer to directory entry structure

Return Value

`ADI_FSS_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See Table 12-1 on page 12-55 for a list of return codes.

adi_fss_DirSeek

Description

The `adi_fss_DirSeek()` routine seeks to the specific place in a directory stream specified by `DirHandle`. The `Seekpos` argument should be obtained from a previous call to `adi_fss_DirTell()`.

Prototype

```
u32 adi_fss_DirSeek(
                    ADI_FSS_DIR_HANDLE   DirHandle,
                    u32                  tellpos
);
```

Arguments

`DirHandle`	Handle identifying directory stream
`tellpos`	Position (in bytes) in directory stream to seek

Return Value

`ADI_FSS_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See Table 12-1 on page 12-55 for a list of return codes.

adi_fss_DirTell

Description

The `adi_fss_DirTell()` routine reports the current position (in bytes) in the specified directory stream.

Prototype

```
u32 adi_fss_DirTell(
                    ADI_FSS_DIR_HANDLE   DirHandle,
                    u32                  *tellpos
);
```

Arguments

`DirHandle`	Handle identifying directory stream
`tellpos`	Location to store current position

Return Value

`ADI_FSS_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See Table 12-1 on page 12-55 for a list of return codes.

adi_fss_DirRewind

Description

The `adi_fss_DirRewind()` routine rewinds the specified directory stream to its beginning.

Prototype

```
u32 adi_fss_DirRewind(ADI_FSS_DIR_HANDLE DirHandle);
```

Arguments

`DirHandle`	Handle identifying directory stream

Return Value

`ADI_FSS_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See Table 12-1 on page 12-55 for a list of return codes.

adi_fss_DirChange

Description

The `adi_fss_DirChange()` routine changes the current working directory, from which relative file paths are evaluated. The `name` value can be specified as either a relative or absolute path.

Prototype

```
u32 adi_fss_DirChange(
                      ADI_FSS_WCHAR  *name,
                      u32            namelen
);
```

Arguments

`name`	NULL-terminated string identifying new current working directory
`namelen`	Length of name string excluding terminating NULL; for example, the value that `strlen(name)` would return

Return Value

`ADI_FSS_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See the Table 12-1 on page 12-55 for a list of return codes.

adi_fss_GetCurrentDir

Description

The adi_fss_GetCurrentDir() routine returns the name of the current working directory.

Prototype

```
u32 adi_fss_GetCurrentDir(
                         ADI_FSS_WCHAR   *name,
                         u32             *namelen
);
```

Arguments

name	Array to store NULL-terminated string containing the current working directory
namelen	Location to store length of name array

Return Value

ADI_FSS_RESULT_SUCCESS	No error has been encountered.
Any other value	Error has occurred. See Table 12-1 on page 12-55 for a list of return codes.

adi_fss_DirCreate

Description

The `adi_fss_DirCreate()` routine creates a specified directory. The `name` argument can be specified as an absolute or relative path, provided that all intermediate directories exist.

Prototype

```
u32 adi_fss_DirCreate(
                      ADI_FSS_WCHAR   *name,
                      u32             namelen,
                      u32             mode
);
```

Arguments

`name`	NULL-terminated string identifying directory to open
`namelen`	Length of name string excluding terminating NULL; for example, the value that `strlen(name)` would return
`mode`	Mode of newly-created directory (ignored). This argument is reserved for future use.

Return Value

`ADI_FSS_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See Table 12-1 on page 12-55 for a list of return codes.

adi_fss_DirRemove

Description

The `adi_fss_DirRemove()` routine removes the specified directory. The `name` argument can be specified as an absolute or relative path.

Prototype

```
u32 adi_fss_DirRemove(
                    ADI_FSS_WCHAR   *name,
                    u32             namelen
);
```

Arguments

`name`	NULL-terminated string identifying directory to remove
`namelen`	Length of name string excluding terminating NULL; for example, the value that `strlen(name)` would return

Return Value

`ADI_FSS_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See Table 12-1 on page 12-55 for a list of return codes.

File System Service API Data Types and Enumerations

This section describes the data types used by the FSS API and the result code enumerations.

ADI_FSS_WCHAR

The `ADI_FSS_WCHAR` data type is used for all file and directory names used in the FSS API. As mentioned in the "Introduction" on page 12-2, the current implementation of the FSS caters only for ASCII file names (8-bit). This data type is defined as:

```
typedef char ADI_FSS_WCHAR;
```

ADI_FSS_VOLUME_IDENT

The `ADI_FSS_VOLUME_IDENT` data type is used for the identifiers of each mounted volume. The data type is defined as:

```
typedef char ADI_FSS_VOLUME_IDENT;
```

ADI_FSS_FILE_HANDLE

The `ADI_FSS_FILE_HANDLE` is an opaque data type used to uniquely identify the file stream to manipulate. It is defined as:

```
typedef void *ADI_FSS_FILE_HANDLE;
```

ADI_FSS_DIR_HANDLE

The ADI_FSS_DIR_HANDLE is an opaque data type used to uniquely identify the directory stream to manipulate. It is defined as:

```
typedef void *ADI_FSS_DIR_HANDLE;
```

ADI_FSS_CMD_VALUE_PAIR

The ADI_FSS_CMD_VALUE_PAIR data type is a structure containing a command code and an associated value. These command-value pairs are used in the FSS configuration table and in calls to adi_fss_Control(). The appropriate commands are detailed at "adi_fss_Init" on page 12-19 and "adi_fss_Control" on page 12-23. The structure is defined as:

```
typedef struct {
    u32     CommandID;                              */
    void    *Value;
} ADI_FSS_CMD_VALUE_PAIR;
```

where the fields are assigned as shown in the following table.

CommandID	ID of command to execute
Value	Associated value as appropriate for the command ID

ADI_FSS_DIR_ENTRY

The ADI_FSS_DIR_ENTRY data type is a structure that contains the details of a directory entry. The structure is identical to the POSIX struct dirent structure:

```
typedef struct dirent ADI_FSS_DIR_ENTRY;
```

The struct dirent structure is described in the POSIX documentation (see "Additional POSIX Functions Supported by the FSS" on page 12-70 for the location of this document) and is defined in the FSS header file, `<services/fss/adi_fss.h>`.

ADI_FSS_DEVICE_DEF

The `ADI_FSS_DEVICE_DEF` data type defines a structure to contain the information required to open and configure an FSD or PID device driver. It is defined as:

```
typedef struct {
    u32                          DeviceNumber;
    ADI_DEV_PDD_ENTRY_POINT      *pEntryPoint;
    ADI_DEV_CMD_VALUE_PAIR       *pConfigTable;
    void                         *pCriticalRegionData;
    ADI_DEV_DIRECTION            Direction;
    ADI_DEV_DEVICE_HANDLE        DeviceHandle;
    ADI_FSS_VOLUME_IDENT         DefaultMountPoint;
} ADI_FSS_DEVICE_DEF;
```

where the assigned fields are shown in the following table.

`DeviceNumber`	This defines which peripheral device to use. This is the `DeviceNumber` argument required for a call to `adi_dev_Open()`. See the appropriate driver documentation for valid values.
`pEntryPoint`	This is a pointer to the device driver entry points and is passed as the `pEntryPoint` argument required for a call to `adi_dev_Open()`. See the appropriate driver header file and documentation for the entry point declaration.
`pConfigTable`	This is a pointer to the table of command-value pairs to configure the device driver. If no commands are required, this value can be set to NULL.
`pCriticalRegionData`	This is a pointer to the argument that should be passed to the system services `adi_int_EnterCriticalRegion()` function. This is currently not used and should be set to NULL.

`Direction`	This is the `Direction` argument required for a call to `adi_dev_Open()`. See the appropriate driver documentation for valid values.
`DeviceHandle`	This is the location used internally to store the device driver handle on return from a call to `adi_dev_Open()`. It should be set to NULL prior to initialization.
`DefaultMountPoint`	This is the default drive letter used for volumes managed by the driver. This is especially useful for removable media types such as USB memory sticks, SD cards, and optical media. This can be left as NULL if no preference is desired.

Result Codes

Each of the FSS API functions returns a result code that can be interpreted by looking up the value in the FSS service API header file, `adi_fss.h`. The result codes are summarized in Table 12-1.

Table 12-1. Result Codes

Result Code Name	Result Code Description
`ADI_FSS_RESULT_SUCCESS`	Generic success = 0
`ADI_FSS_RESULT_FAILED`	Generic failure = 1
`ADI_FSS_RESULT_OPEN_FAILED`	(0XB0001) file/directory open failure (media fault)
`ADI_FSS_RESULT_NOT_FOUND`	(0XB0002) file/directory not found
`ADI_FSS_RESULT_CLOSE_FAILED`	(0XB0003) file/directory close failure (media fault)
`ADI_FSS_RESULT_NO_MEDIA`	(0xB0004) no media detected
`ADI_FSS_RESULT_MEDIA_CHANGED`	(0xB0005) media has changed since last check
`ADI_FSS_RESULT_MEDIA_FULL`	(0xB0006) no room left on media
`ADI_FSS_RESULT_NO_MEMORY`	(0xB0007) insufficient memory to perform request

Table 12-1. Result Codes (Cont'd)

Result Code Name	Result Code Description
`ADI_FSS_RESULT_INVALID_DEVICE`	(0xB0008) device driver cannot be initialized
`ADI_FSS_RESULT_BAD_FILE_HANDLE`	(0xB0009) no valid file descriptor at address supplied
`ADI_FSS_RESULT_BAD_NAME`	(0xB000A) invalid path specified for file/directory
`ADI_FSS_RESULT_EOF`	(0xB000B) end of file reached
`ADI_FSS_RESULT_EOD`	(0xB000C) end of directory reached
`ADI_FSS_RESULT_BAD_VOLUME`	(0xB000D) invalid partition specified
`ADI_FSS_RESULT_NOT_SUPPORTED`	(0xB000E) command code is not supported
`ADI_FSS_RESULT_TIMEOUT`	(0xB0010) timeout has occurred
`ADI_FSS_RESULT_BAD_CACHE_HANDLE`	(0xB0011) bad cache handle
`ADI_FSS_RESULT_CANT_CREATE_SEMAPHORE`	(0xB0012) cannot create semaphore in file cache

The Standard C I/O Interface Functions

This section briefly details the standard C I/O API. By registering the FSS with the C I/O library as detailed in Chapter 3 of the *C and C++ Compiler and Linker Manual for Blackfin Processors*, and as described in "Getting Started" on page 12-3, calls to functions in this section are routed to the FSS after processing by the I/O library. All functions follow the POSIX convention for the argument and return types. Please refer to a suitable C textbook for further details.

This list details most of the available functions, but any function defined in `<stdio.h>` for file access results in appropriate (primitive) calls to the FSS API.

fopen

Description

The `fopen()` function opens a named file in accordance to the mode flags.

Prototype

```
FILE * fopen (const char *name, const char *mode);
```

Result Codes

fclose

Description

The `fclose()` function closes a file identified by `stream`.

Prototype

```
int fclose (FILE * stream);
```

fwrite

Description

The `fwrite()` function writes `num` data elements of `size` bytes to the file identified by `stream`.

Prototype

```
int fwrite (void *buffer, size_t size, size_t num, FILE *stream);
```

Result Codes

fread

Description

The `fread()` function reads `num` data elements of `size` bytes from the file identified by `stream`.

Prototype

```
int fread (void *buffer, size_t size, size_t num, FILE *stream);
```

fprintf

Description

The `fprintf()` function writes to the file identified by `stream` as determined by the format string. The VisualDSP++ 5.0 I/O library performs all the required format processing.

Prototype

```
int fprintf (FILE *stream, const char *fmt, ...);
```

Result Codes

fscanf

Description

The `fscanf()` function reads from the file identified by `stream` as determined by the format string. The VisualDSP++ 5.0 I/O library performs all the required format processing.

Prototype

```
int fscanf (FILE *stream, const char *fmt, ...);
```

fgetc

Description

The `fgetc()` function reads the next byte from the file identified by `stream`.

Prototype

```
int fgetc (FILE *stream);
```

Result Codes

fgets

Description

The `fgets()` function reads the next `n` bytes from the file identified by `stream` into the `buf` character array.

Prototype

```
char * fgets (char *buf, int n, FILE *stream );
```

fputc

Description

The `fputc()` function writes the `ci` byte to the file identified by `stream`.

Prototype

```
int fputc (int ci, FILE *stream);
```

Result Codes

fputs

Description

The `fputs()` function writes the text string, `s`, to the file identified by `stream`.

Prototype

```
int fputs (const char *s, FILE *stream);
```

fseek

Description

The `fseek()` function moves the current file position.

Prototype

```
int fseek (FILE *stream, long offset, int origin);
```

Result Codes

ftell

Description

The `ftell()` function returns the current position in a file.

Prototype

```
long ftell (FILE *stream);
```

feof

Description

The `feof()` function returns whether the current file is at end-of-file (EOF).

Prototype

```
int feof (FILE *stream);
```

Additional POSIX Functions Supported by the FSS

This section provides a brief listing of the additional POSIX functions and structures supported by the FSS system service. A more comprehensive description of the routine can be found with an internet search on the function name or through viewing the function's reference page in *The Single UNIX® Specification, Version 2, System Interface & Headers Reference Pages*, available online at:

`http://www.opengroup.org/onlinepubs/007908775/xshix.html`.

The functions listed in this section have no implementation within the VisualDSP++ 5.0 I/O library except for `rename()` and `remove()`. These two functions also exist within the VisualDSP++ 5.0 I/O library but are limited to PRIMIO access to the file system of the host PC.

To use all functions in this section, you will need to include the FSS header file, `<services/fss/adi_fss.h>`, in your application source file. By including this header file, all references to rename and remove in the source file will resolve to the FSS variants for these functions.

opendir

Description

The `opendir()` function opens the specified directory for processing by `readdir`, `readdir_r`, `telldir`, `seekdir`, `rewinddir`, and `closedir`.

Prototype

```
DIR *opendir (const char *dirname)
```

Result Codes

closedir

Description

The `closedir()` function closes access to a directory structure pointed to by the `dirp` argument.

Prototype

```
int closedir (DIR * dirp)
```

readdir

Description

The `readdir()` function returns a pointer to a `dirent` structure representing the next directory entry in the specified directory stream. It returns NULL on reaching the end-of-file or if an error occurred. The filenames are returned in the order in which they are stored by the file system.

Prototype

```
struct dirent *readdir (DIR *dirp)
```

Result Codes

readdir_r

Description

The `readdir_r()` function behaves similarly to `readdir` but instead of returning a pointer to a `dirent` structure it populates the structure referenced by the `entry` argument with the details of the directory entry at the current position in the directory stream. A pointer to this structure is also returned in the location specified by the `result` argument. This location will contain NULL upon reaching the end of the directory stream. Like `readdir` it positions the directory stream at the next entry upon return.

Prototype

```
int readdir_r (DIR *dirp, struct dirent *entry, struct dirent
**result)
```

rewinddir

Description

The `rewinddir()` function rewinds the specified directory.

Prototype

```
void rewinddir (DIR *dirp)
```

Result Codes

seekdir

Description

The `seekdir()` function sets the position of the next `readdir()` operation on the directory stream specified by `dirp` to the position specified by `loc`.

Prototype

```
void seekdir (DIR *dirp,  long loc);
```

telldir

Description

The `telldir()` function obtains the current location associated with the specified directory stream.

Prototype

```
long telldir (DIR *dirp);
```

Result Codes

mkdir

Description

The `mkdir()` function creates the specified directory. The `mode` argument is ignored by the FSS.

Prototype

```
int mkdir (const char *path, mode_t mode)
```

rmdir

Description

The `rmdir()` utility deletes the specified directory.

Prototype

```
int rmdir (const char *path);
```

Result Codes

rename

Description

The `rename()` function renames the file from `_oldnm` to `_newnm`.

Prototype

```
int rename (const char *_oldnm, const char *_newnm);
```

remove

Description

The `remove()` function removes the specified file.

Prototype

```
int remove (const char *_filename);
```

Extensibility

The file system service supports a straightforward and simple approach to the insertion of additional or replacement drivers, to support different file systems (FSDs) to interpret the media attached to new or existing physical interfaces (PIDs).

New device drivers for insertion into the FSS must conform to either the FSD or PID design documents supplied with VisualDSP++ 5.0:

```
Blackfin\docs\drivers\fsd\Generic_FSD_Design_Document.pdf,
Blackfin\docs\drivers\pid\Generic_PID_Design_Document.pdf,
```

These documents detail how the drivers interface with the other components of the file system service framework.

Once created, the driver is directly available for incorporation within the FSS. All that is required is to declare an `ADI_FSS_DEVICE_DEF` structure (and accompanying configuration table if required) and register it with the FSS via the `ADI_FSS_CMD_ADD_DRIVER` command:

```
   { ADI_FSS_CMD_ADD_DRIVER,(void*)&<Device-Def-Structure> },
```

The details of the `ADI_FSS_DEVICE_DEF` structure are described in "File System Service API Data Types and Enumerations" on page 12-52.

Examples

Included with VisualDSP++ 5.0 are some examples to demonstrate how the FSS is configured and how it can be used. As examples, their scope of operation has been purposely limited; please employ caution when using them as a basis for your own applications. The examples described in this section are all specific to the ADSP-BF548 EZ-KIT Lite development board. Similar examples may exist for the ADSP-BF527 EZ-KIT Lite development board but will use only the USB host mass storage interface to access USB flash drives.

Three examples are included. The first (*HardDiskAccess)* shows how to get started with the file system service. The second (*HardDiskFormat*) is somewhat more complex and demonstrates how to use the constituent drivers away from the FSS in order to format the hard disk drive, especially if it has never been formatted previously. The final example (*Shell_Browser*) is a multi-threaded, VDK-based application demonstrating the use of the file system service within a multi-threaded, multi-peripheral environment.

HardDiskAccess

This simple program (the *Hello World* of the FSS) demonstrates the basic configuration of the file system service and its use.

Description

This example program performs the following operations:

1. Shows the characteristics of the FAT partition.
2. Creates a sub-directory on a hard disk.
3. Moves to that directory.
4. Creates a file in that directory.
5. Lists the contents of that directory.
6. Performs a checksum operation on the created file and displays the results.
7. Removes that file.
8. Moves back to the root directory.

9. Removes the sub-directory.
10. Exits the program.

This program assumes an attached hard drive that has been previously formatted as a single FAT partition. The file created is filled with known data and the checksum operation checks against a previously-calculated checksum value to ensure data integrity. Please note that the timestamps of the created files may appear arbitrary. This is due to the fact that the real-time clock (RTC) may have not been set yet. See the `Shell_Browser` application for details of setting the RTC.

Configuration

The `HardDiskAccess` example initializes the FSS for:

- Two cache blocks per file
- The ATA/ATAPI driver
- The FAT file system

Configuration is performed in the `InitfileSystem.c` file with a call to `adi_fss_Init()` by passing it the address of the `adi_fss_config` array. Then, the FSS system is added to the C runtime library device table. When configured, the FSS service becomes the default for file access via the standard I/O interface.

HardDiskFormat

This program is used to format the attached hard disk as a single, 32GB FAT 32 partition. This example is only relevant to the ADSP-BF548 EZ-KIT Lite development board.

Description

This example program is a simple, standalone application that initializes the SSL, loads the ATAPI PID, attaches to the ATAPI hard disk, formats the disk, and then exits.

Configuration

The application only initializes the FSS in as far as establishing the general heap index, for memory allocation within the drivers. Otherwise, both the ATAPI PID and FAT FSD drivers are configured and accessed directly to format the partition and create the master boot record (MBR).

Shell_Browser

This application provides a simple shell interface to the file system service to navigate the media available and perform certain operations such as displaying a slideshow of bitmap files to the LCD attached to the ADSP-BF548 EZ-KIT Lite.

Description

This example program uses three threads—a communication thread to interface with an I/O console connected to the UART interface, an image display thread to view images on the LCD panel, and a command thread to control the application and perform non-image related file tasks. The communication thread collects the input from the I/O console and passes it to the command thread. The image display thread takes an image file or a list of image files and displays the images to the LCD panel. The command thread interprets the input commands and performs the required actions. A full list of the commands supported is available by typing "help" or "?". Likewise, the syntax of any individual command can be obtained by typing the command followed by `-h`. For example, `rmdir -h` lists the syntax and purpose of the command.

Since the example consists of three separate threads, the order of initialization and interoperability is important. The command thread performs the initialization of the system services prior to the other threads being executed. The image display thread and the console thread are reliant on the fact that all required system services have been initialized already.

Configuration

The `Shell_Browser` sets up the FSS for:

- Two cache blocks per file
- The ATA/ATAPI driver
- The SD card driver
- The USB host mass storage driver
- The FAT file system
- The UART device driver for data transmission

When the example is running, it:

- Registers the UART console driver to be used for `stdin` and `stdout`
- Initializes the image viewer LCD driver
- Starts the image viewer thread

Once all three threads are running, the program is ready to accept keyboard input and perform the input operations. Note that the `Shell_Browser` application makes extensive use of most of the different system services to perform its operations. Its utility is not limited to simply showing the operation of the file system service, but extends to giving an example of the use of system services and device drivers, and the FSS in particular, in a threaded environment.

Please note that the timestamps of any created files may appear arbitrary. This is due to the fact that the real-time clock (RTC) may have not been set yet. This can be achieved using the date command. For example, to set the date to 2:30 P.M. on Monday, September 17, 2007, enter the following at the prompt: `date mon 091714302007`. Enter `'date -h'` for full usage details.

13 PULSE-WIDTH MODULATION

This chapter describes the pulse-width modulation (PWM) service, which provides the application with an easy-to-use interface to the PWM controller, featured on some Blackfin processors.

This chapter contains:

Introduction

The system services pulse-width modulation (PWM) service facilitates control over the programmable, pulse-width modulation unit, which generates waveforms to drive a three-phase voltage source inverter, for use in motor control applications.

A single PWM is featured on the ADSP-BF512/514/516/518 family of Blackfin processors. Two identical PWM modules are featured on the ADSP-BF50/504F/506F Blackfin family.

This chapter is devoted entirely to the use of the system services PWM service in software applications. Further details of the PWM module hardware are available in the *ADSP-BF50x* and *ADSP-BF51x Blackfin Processor Hardware Reference* manuals.

The debug version of the system services library provides parameter checking for a more complete test of the API function parameters and other error conditions. Analog Devices strongly recommends that development work be done using the debug versions of the system service library, and that final test and deployment be done with the release version of the library.

Operation

This section describes the basic features and overall operation of the PWM service. Details on the application programming interface (API) can be found later in this chapter.

Initialization

Prior to using the PWM service, the application must initialize the service by calling the PWM initialization function, adi_pwm_Init.

In the presence of two PWMs, adi_pwm_Init is called once, with separate command pairs for each PWM. No command pairs should be passed for an unused PWM. The commands and their associated values are selected from the predefined enumerations and data structures described within this chapter, all of which begin with the `ADI_PWM_` prefix. When it is necessary to identify which PWM a command or value is intended for, an optional 0 or 1 is appended after the prefix, documented as `ADI_PWM(x)_` , where the “`x`” represents the optional PWM number.

For each enabled PWM, certain parameters must be passed to the initialization function, which the PWM service uses to set the associated values in the PWM memory-mapped registers (MMRs). The required parameters are:

- Port mux mapping for each signal (primary or secondary)
- Period of synchronization pulse
- Width of synchronization pulse (must be > 2 system clock periods)
- Dead time inserted after the “ideal” output pair
- Duty cycle for each channel pair
- Enable status for each individual channel or for all channels
- Polarity of all output signals
- Operating mode of the PWM (single update or double update)

There are additional optional configuration parameters which may be passed to either the initialization function, or the PWM control function, adi_pwm_Control. These optional parameters are:

- **Sync pulse on output pin**. The synchronization pulse signal may be generated on an output pin. This behavior is disabled by default, and may be enabled or disabled by passing a command with an argument which specifies either enable or disable.
- **Internal or external sync pulse.** The synchronization pulse is generated internally, by default, as a function of the PWM synchronization pulse width and frequency. The PWM may be programmed to expect the synchronization pulse from an external source, appearing on an input pin. This allows multiple PWMs to share a synchronization pulse from the same external source.
- **Synchronous or asynchronous external sync pulse.** By default, an externally supplied synchronization pulse is synchronous to the internal system clock, but a command may be passed to change the external pulse to asynchronous.
- **IVG levels.** IVG10 is the default assignment of the two PWM interrupting signals, trip and sync. A command may be passed to change the IVG of either interrupt signal, to avoid any possible IVG sharing conflicts within the application.
- **Switch reluctance.** This mode is disabled by default but may be enabled or disabled by passing a command, with an argument which specifies enable or disable.
- **Channel crossover mode.** This mode, disabled by default, can be enabled or disabled by passing a command with an argument specifying the channel pair (A, B, or C) and the desired state (enable or disable). When crossover is enabled, the two signals of the specified PWM channel pair are swapped, so that the PWM signal destined for the high-side switch is diverted to the complementary low-side output, and vice versa.

- **Gate chopping mode.** This mode, disabled by default, may be enabled or disabled by passing a command with an argument specifying the desired state (enable or disable). With gate chopping enabled, the output signals may be mixed with a high-frequency chopping signal. Chopping may be independently enabled for the high-side and low-side outputs, using two separate commands.
- **Trip input signal.** The trip input signal may be permanently disabled in the hardware by a pull-up resistor. Generally, the trip input signal is enabled by default, and may be enabled or disabled by passing a command with an argument specifying either "enable" or "disable".

Commands to enable or disable the trip and sync interrupt signals should not be passed to the adi_pwm_Init function. These interrupts are automatically disabled during the initialization phase, and any pending trip and sync interrupt signals are cleared.

The initialization function saves the trip and sync interrupt vector group (IVG) assignments for future reference. It calls the interrupt manager service to hook the PWM trip and PWM sync interrupt handlers, which service the trip and sync interrupts.

When built for debug mode, the initialization function verifies that all required initialization parameters have been passed in and processed, for each enabled PWM, before it sets a PWM initialization flag, which then tells the other PWM service API functions that PWM initialization has been successfully completed. The initialization function returns an error result code to the caller, in debug mode, if not all of the required parameters were passed in. If a PWM API function is called while the PWM initialization flag is not set, an error result code will be returned to the caller.

Once initialization has been successfully completed, the application may install callbacks (see Callbacks) for the trip and sync events on each enabled PWM (see PWM Events) by calling adi_pwm_InstallCallback.

Beginning with VisualDSP++ 5.0 Update 8, the adi_pwm_Init function no longer enables the PWM unit. The PWM hardware should be enabled after callbacks have been installed, by passing the `ADI_PWM_CMD_SET_PWM_ENABLE` command to the adi_pwm_Control function.

In the presence of multiple PWMs, there is one trip event and one sync event for each PWM.

The interrupt for the trip event is not enabled until either a callback is installed for the trip event, or the `ADI_PWM_CMD_SET_TRIP_INT_ENABLE` command is passed to the adi_pwm_Control function, with the value `ADI_PWM(x)_ENABLE` argument.

The interrupt for the sync event is not enabled until either a callback is installed for the sync event or the `ADI_PWM_CMD_SET_SYNC_INT_ENABLE` command is passed to the adi_pwm_Control function, with the value `ADI_PWM(x)_ENABLE` argument.

Before initializing the PWM service, the application should initialize the interrupt manager by calling adi_pwm_Init. Because PWM signals are multiplexed, the port control manager service must be initialized by calling `adi_ports_Init()`. If callbacks are to be deferred, rather than “live”, then the deferred callback (DCB) manager should also be initialized by calling `adi_dcb_Init()`.

Termination

When the application no longer requires the use of PWM service, it calls the termination function, adi_pwm_Terminate. The termination function disables the PWM interrupts, unhooks the PWM interrupt handler, uninstalls any callbacks (see Callbacks) which had been installed by the application, cleans up any statically-defined data structures used by the PWM service, and clears the PWM service initialization flag, so that PWM API functions may no longer be called.

PWM Events

The PWM service provides a mechanism for allowing certain events to be serviced at the interrupt level during regular program execution. Each event is identified by a unique `Event ID` that is defined in the include file `adi_pwm.h`. The application may provide a separate callback function to handle each event individually, or it may provide a single callback to handle all the events. The PWM service provides the mechanism for installing and removing callbacks (see Callbacks), and for invoking a callback when the appropriate conditions are met. The PWM service supports two external asynchronous events for each PWM: trip and synchronization. Each of these events can be configured to generate an interrupt to the core. The application enables an event by calling the adi_pwm_InstallCallback function, passing the appropriate `Event ID`.

Trip Signal Event

The application may enable the trip event to generate an interrupt by calling the adi_pwm_InstallCallback function, passing the argument `ADI_PWM(x)_EVENT_TRIP`. The trip input is an emergency fault condition, which automatically shuts down the PWM, placing all six PWM channels in the OFF state. Passing the `ADI_PWM_CMD_SET_TRIP_INT_ENABLE` command to the adi_pwm_Control function, with the command argument `ADI_PWM(x)_ENABLE`, causes the trip event to generate an interrupt with no callback.

Synchronization Pulse Event

The application may enable a periodic synchronization (or sync) interrupt to occur on the rising edge of the synchronization pulse for each PWM. This interrupting event is enabled by passing the argument `ADI_PWM(x)_EVENT_SYNC` to the adi_pwm_InstallCallback function. Passing the `ADI_PWM_CMD_SET_SYNC_INT_ENABLE` command to the adi_pwm_Control function, with the command argument `ADI_PWM(x)_ENABLE`, causes the sync event to generate an interrupt with no callback.

The period of the sync pulse is set by passing the ADI_PWM_CMD_SET_PERIOD command to the adi_pwm_Init function. The sync pulse is available for external use by passing the ADI_PWM_CMD_SET_SYNC_OUT_ENABLE command. The width of the pulse is programmed by passing the command ADI_PWM_CMD_SET_SYNC_PULSE_WIDTH.

By default, the sync pulse is internally generated. The command ADI_PWM_CMD_SET_SYNC_SOURCE may be passed to tell the PWM service that the pulse is generated externally. The command ADI_PWM_CMD_SET_SYNC_SELECT should also be passed to specify whether the externally-generated sync pulse is synchronous or asynchronous to the internal clock.

In single update mode, a synchronization event occurs at the beginning of each period. In double update mode, the event occurs at both the beginning and at the mid-point of each period, so two sync events happen in each period. The command, ADI_PWM_CMD_GET_PHASE, can be passed to the adi_pwm_Control function to determine whether the sync event signifies the first half or the second half of the PWM cycle.

Typically, the application installs a callback for the synchronization event, for the purpose of triggering an ADC to sample data, and possibly for updating the three PWM channel duties, according to the control algorithm performed on the sampled data.

This event stays enabled and periodic interrupts continue to occur, until the application disables the event by passing the argument ADI_PWM(x)_EVENT_SYNC to the adi_pwm_RemoveCallback function, or by passing the command ADI_PWM_CMD_SET_TRIP_INT_ENABLE, with the command argument ADI_PWM(x)_DISABLE, to the adi_pwm_Control function.

Callbacks

The application may enable and process any of the PWM events, by installing a callback to handle the event. The callback is executed whenever the associated event occurs. A separate callback function may handle each event, or a single callback function may handle all of the PWM events. Unless a callback is deferred, it is executed from within the interrupt service routine, by the PWM trip interrupt handler, or PWM sync interrupt handler, which are internal to the PWM service.

The PWM service provides the functions to install the callbacks, the functions to remove the callbacks, and the interrupt handlers which invoke the callbacks. The application provides the callback functions, which control how each event is processed. The application may also provide an optional data structure, comprised of event-specific information, that is passed to the callback function from the PWM service trip and sync interrupt handlers.

Installing a Callback

To install callback functionality for one of the PWM events, the application simply calls the API function adi_pwm_InstallCallback passing the `Event ID` for the event to be processed by the callback, and the name of the callback function. The PWM service maintains a callback table with an entry for each event. When a callback is installed, the entry for the event is updated to point to the callback function. The PWM is configured so that the event triggers an interrupt, and the interrupt manager service is configured to unmask the interrupt in the core event controller (CEC) register, if it had not already been enabled.

An optional data structure called `ClientHandle` may be passed, which contains information to be stored in the callback table and passed to the callback, when it occurs.

Removing a Callback

The application may disable the processing of an event by passing the `EventID` as a parameter to the adi_pwm_RemoveCallback function. This function removes the entry for that event from the callback table and calls upon the interrupt manager service to disable the interrupt for the event. The event may be enabled to interrupt, without executing a callback, by passing the appropriate command `ADI_PWM_CMD_SET_TRIP_INT_ENABLE` or

ADI_PWM_CMD_SET_SYNC_INT_ENABLE, with the argument ADI_PWM(x)_ENABLE.

The PWM Service Interrupt Handlers

The PWM service has an interrupt handler for each of the PWM events: trip (for each PWM) and sync (for each PWM). The handlers are hooked into the interrupt manager's chain for the associated interrupt vector group (IVG). Note the difference between the following three types of functions.

1. **Interrupt handler.** This is defined inside the PWM service to process PWM events.
2. **Interrupt service routine (ISR).** This is defined inside of the interrupt manager, to service an interrupting event. The interrupt manager's ISR executes the PWM interrupt handlers as well as any other handlers that are hooked into the chain for the same IVG.
3. **Callback function.** When the ISR calls a PWM trip or sync handler function, the handler executes any live callbacks it finds in the callback table, passing the `ClientHandle` parameter which is also stored in the callback table entry. If callbacks are deferred, the handler posts the callbacks to the deferred callback service.

The PWM sync interrupt handler checks to see if the sync interrupt signal is active. In the presence of multiple PWMs, there is a sync handler for each PWM, which also verifies that the sync interrupt is active for the correct PWM. It then checks the callback table entry for that event, to find the address of the callback, plus other event-specific information. The handler executes the callback, passing the other information that it finds in the callback list entry for the event. When the callback is complete, it returns to the PWM sync interrupt handler, which returns control to the interrupt manager.

The PWM trip interrupt handler checks to see if the trip interrupt signal is active. In the presence of multiple PWMs, there is a trip handler for each PWM, which also verifies that the trip interrupt is active for the correct PWM. It then checks the callback table entry for that event, to find the address of the callback, plus other event-specific information. The handler executes the callback, passing the other information that it finds in the callback list entry for the event. When the callback is complete, it returns to the PWM trip interrupt handler, which returns control to the interrupt manager.

Using the ClientHandle Parameter in a Callback

An optional data structure called `ClientHandle` is passed as an argument to the adi_pwm_InstallCallback function. This user-defined structure contains event-specific information to be stored in the callback table entry. When the event occurs, the interrupt manager executes the PWM service interrupt handler. The interrupt handler uses the callback table to find the address of the callback and the `ClientHandle` information that it must pass to the callback.

Programming Examples

A complete PWM service coding example is included in the VisualDSP++ installation. The example, which generates a 50 Hz, 3-phase sine wave, demonstrates the proper initialization of the PWM service, the calculation of dead time, sync pulse period, and sync pulse width, based on a desired fundamental frequency and the system clock (SCLK). It demonstrates the installation of a sync pulse callback, which occurs periodically to modify the duties for each channel.

The following sections contain code samples which demonstrate the correct usage of the PWM commands in an application.

In the ADI_PWM(x)_ prefix, the "x" in parentheses indicates an optional PWM number. It should be replaced by a 0 or 1 for the ADSP-BF50x family (ADI_PWM0_ or ADI_PWM1_). For the ADSP-BF51x family it should be removed, as there is only one PWM, (ADI_PWM_).

Initialization – Command-Pair Table

Listing 13-1 and Listing 13-2 show command-pair tables of required commands, which must be passed to the PWM initialization function for proper PWM functionality. Listing 13-1 is for the ADSP-BF51x Blackfin family, which has one PWM. Listing 13-2 is for the ADSP-BF50x Blackfin family, which has two PWMs. Optional commands, which may be passed here or in later calls to the adi_pwm_Control function, are described in the sub-sections which follow.

For more information on the command-pair structure data type, see "ADI_PWM_NUMBER" on page 13-46.

Please refer to the sine wave example in the VisualDSP++ installation for information on calculating the values for PWM_SyncPeriod, PWM_SyncWidth, and PWM_DeadTime.

Listing 13-1. Command-Pair Table Initialization for the ADSP-BF51x

```
/* Populate an ADI_PWM_CHANNEL_STATUS structure to enable all
three channels at once */
ADI_PWM_CHANNEL_STATUS pwm_EnableALL_struct  =
   {ADI_PWM_CHANNEL_ALL, ADI_PWM_ENABLE};

/* Populate three ADI_PWM_CHANNEL_DUTY_CYCLE structures to set
the duty cycles for the three individual channel pairs (0 = 50%
duty cycle) */
 ADI_PWM_CHANNEL_DUTY_CYCLE  pwm_dutyA_struct  =
   {ADI_PWM_CHANNEL_A, 0};
 ADI_PWM_CHANNEL_DUTY_CYCLE  pwm_dutyB_struct  =
   {ADI_PWM_CHANNEL_B, 0};
 ADI_PWM_CHANNEL_DUTY_CYCLE  pwm_dutyC_struct  =
   {ADI_PWM_CHANNEL_C, 0};

/* Define a port mux mapping table using all primary
mux signals */
 ADI_PWM_PORT_MAP pwm_PortMuxMap  =
 {
    ADI_PWM_MUX_PRI,
    ADI_PWM_MUX_PRI,
    ADI_PWM_MUX_PRI,
    ADI_PWM_MUX_PRI,
    ADI_PWM_MUX_PRI,
    ADI_PWM_MUX_PRI,
    ADI_PWM_MUX_PRI,
    ADI_PWM_MUX_PRI,
 };

    ADI_PWM_COMMAND_PAIR PwmInitTable[] =
 {
 /* required initialization commands */
{ADI_PWM_CMD_SET_PORT_MUX, (void*) &pwm_PortMuxMap},
{ADI_PWM_CMD_SET_PERIOD, (void*) PWM_SyncPeriod},
{ADI_PWM_CMD_SET_SYNC_PULSE_WIDTH, (void*) PWM_SyncWidth},
{ADI_PWM_CMD_SET_DEAD_TIME, (void*) PWM_DeadTime},
{ADI_PWM_CMD_SET_DUTY_CYCLE, (void*) &pwm_dutyA_struct},
{ADI_PWM_CMD_SET_DUTY_CYCLE, (void*) &pwm_dutyB_struct},
{ADI_PWM_CMD_SET_DUTY_CYCLE, (void*) &pwm_dutyC_struct},
{ADI_PWM_CMD_SET_CHANNEL_ENABLE, (void*) &pwm_EnableALL_struct},
{ADI_PWM_CMD_SET_POLARITY, (void*) ADI_PWM_POLARITY_HIGH},
{ADI_PWM_CMD_SET_UPDATE_MODE, (void*)ADI_PWM_DOUBLE_UPDATE },

/* indicate the last command of the table */
{ADI_PWM_CMD_END, (void*)0      }
};

/* Initialize PWM  service */
Result = adi_pwm_Init(PwmInitTable, (void*)NULL);
```

Listing 13-2. Command-Pair Table Initialization for the ADSP-BF50x

```
/* This structure enables all channels for PWM0 */
ADI_PWM_CHANNEL_STATUS pwm_EnablePWM0_struct  =
   {ADI_PWM_CHANNEL_ALL, ADI_PWM0_ENABLE};

/* This structure enables all channels for PWM1 */
ADI_PWM_CHANNEL_STATUS pwm_EnablePWM1_struct  =
   {ADI_PWM_CHANNEL_ALL, ADI_PWM1_ENABLE};

/* Populate the three structures to set the duty cycles for the
three channels on PWM0 (0 = 50% duty cycle) */
 ADI_PWM_CHANNEL_DUTY_CYCLE  pwm0_dutyA_struct  =
   {ADI_PWM_0, ADI_PWM_CHANNEL_A, 0};
 ADI_PWM_CHANNEL_DUTY_CYCLE  pwm0_dutyB_struct  =
   {ADI_PWM_0, ADI_PWM_CHANNEL_B, 0};
 ADI_PWM_CHANNEL_DUTY_CYCLE  pwm0_dutyC_struct  =
   {ADI_PWM_0, ADI_PWM_CHANNEL_C, 0};
```

```
/* Populate the three structures to set the duty cycles for the
three channels on PWM1 (0 = 50% duty cycle) */
 ADI_PWM_CHANNEL_DUTY_CYCLE  pwm1_dutyA_struct  =
   {ADI_PWM_1, ADI_PWM_CHANNEL_A, 0};
 ADI_PWM_CHANNEL_DUTY_CYCLE  pwm1_dutyB_struct  =
   {ADI_PWM_1, ADI_PWM_CHANNEL_B, 0};
 ADI_PWM_CHANNEL_DUTY_CYCLE  pwm1_dutyC_struct  =
   {ADI_PWM_1, ADI_PWM_CHANNEL_C, 0};

/* Define a port mux mapping table using all primary
mux signals */
 ADI_PWM_PORT_MAP pwm_PortMuxMap  =
 {
    ADI_PWM_MUX_PRI, //AH_0_MUX:1;
    ADI_PWM_MUX_PRI, //AL_0_MUX:1;
    ADI_PWM_MUX_PRI, //BH_0_MUX:1;
    ADI_PWM_MUX_PRI, //BL_0_MUX:1;
    ADI_PWM_MUX_PRI, //CH_0_MUX:1;
    ADI_PWM_MUX_PRI, //CL_0_MUX:1;
    ADI_PWM_MUX_PRI, //SYNC_0_MUX:1;
    ADI_PWM_MUX_PRI, //TRIP_0_MUX:1;
    ADI_PWM_MUX_PRI, //AH_1_MUX:1;
    ADI_PWM_MUX_PRI, //AL_1_MUX:1;
    ADI_PWM_MUX_PRI, //BH_1_MUX:1;
    ADI_PWM_MUX_PRI, //BL_1_MUX:1;
    ADI_PWM_MUX_PRI, //CH_1_MUX:1;
    ADI_PWM_MUX_PRI, //CL_1_MUX:1;
    ADI_PWM_MUX_PRI, //SYNC_1_MUX:1;
    ADI_PWM_MUX_PRI, //TRIP_1_MUX:1;
 };
```

```
      ADI_PWM_COMMAND_PAIR PWMInitTable[] =
   {
   /* required initialization commands */
  {ADI_PWM_CMD_SET_PORT_MUX, (void*) &pwm_PortMuxMap},
  {ADI_PWM_CMD_SET_PERIOD, (void*) &pwm0_SyncPeriod},
  {ADI_PWM_CMD_SET_PERIOD, (void*) &pwm1_SyncPeriod},

  {ADI_PWM_CMD_SET_SYNC_PULSE_WIDTH, (void*) &pwm0_SyncWidth},
  {ADI_PWM_CMD_SET_SYNC_PULSE_WIDTH, (void*) &pwm1_SyncWidth},

  {ADI_PWM_CMD_SET_DEAD_TIME, (void*) &pwm0_DeadTime},
  {ADI_PWM_CMD_SET_DEAD_TIME, (void*) &pwm1_DeadTime},

  {ADI_PWM_CMD_SET_DUTY_CYCLE, (void*) &pwm0_dutyA_struct},
  {ADI_PWM_CMD_SET_DUTY_CYCLE, (void*) &pwm0_dutyB_struct},
  {ADI_PWM_CMD_SET_DUTY_CYCLE, (void*) &pwm0_dutyC_struct},

  {ADI_PWM_CMD_SET_DUTY_CYCLE, (void*) &pwm1_dutyA_struct},
  {ADI_PWM_CMD_SET_DUTY_CYCLE, (void*) &pwm1_dutyB_struct},
  {ADI_PWM_CMD_SET_DUTY_CYCLE, (void*) &pwm1_dutyC_struct},

  {ADI_PWM_CMD_SET_CHANNEL_ENABLE, (void*)&pwm_EnablePWM0_struct},
  {ADI_PWM_CMD_SET_CHANNEL_ENABLE, (void*)&pwm_EnablePWM1_struct},

  {ADI_PWM_CMD_SET_POLARITY, (void*) ADI_PWM0_POLARITY_HIGH},
  {ADI_PWM_CMD_SET_POLARITY, (void*) ADI_PWM1_POLARITY_HIGH},

  {ADI_PWM_CMD_SET_UPDATE_MODE, (void*) ADI_PWM0_DOUBLE_UPDATE },
  {ADI_PWM_CMD_SET_UPDATE_MODE, (void*) ADI_PWM1_DOUBLE_UPDATE },
  /* OPTIONAL - Change the IVG num of PWM0 and PWM1 */
  {ADI_PWM_CMD_SET_SYNC_IVG, (void*) &pwm0_SYNCIVG_struct },
  {ADI_PWM_CMD_SET_TRIP_IVG, (void*) &pwm0_TRIPIVG_struct },
  {ADI_PWM_CMD_SET_SYNC_IVG, (void*) &pwm1_SYNCIVG_struct },
  {ADI_PWM_CMD_SET_TRIP_IVG, (void*) &pwm1_TRIPIVG_struct },

  /* indicate the last command of the table */
  {ADI_PWM_CMD_END, (void*)0      }
  };
```

Set Switch Reluctance

Listing 13-3 enables or disables the switch reluctance mode. This command may be passed to adi_pwm_Control or it may appear in a command-pair table to be passed to adi_pwm_Init.

Listing 13-3. Switch Reluctance Mode – Enable/Disable

```
/* To enable switch reluctance mode */
adi_pwm_Control(ADI_PWM_CMD_SET_SWITCH_RELUCTANCE,
                                   ADI_PWM(x)_ENABLE);

/* To disable switch reluctance mode */
adi_pwm_Control(ADI_PWM_CMD_SET_SWITCH_RELUCTANCE,
                                  ADI_PWM(x)_DISABLE);
```

Crossover

Listing 13-4 enables or disables crossover mode on each channel pair.

Listing 13-4. Crossover Mode – Enable/Disable

```
/* structures to disable or enable crossover for each channel
pair */
ADI_PWM_CHANNEL_STATUS pwm_CrossA_struct =
               {ADI_PWM_CHANNEL_A, ADI_PWM(x)_DISABLE};
ADI_PWM_CHANNEL_STATUS pwm_CrossB_struct =
               {ADI_PWM_CHANNEL_B, ADI_PWM(x)_ENABLE};
ADI_PWM_CHANNEL_STATUS pwm_CrossC_struct =
               {ADI_PWM_CHANNEL_C, ADI_PWM(x)_ENABLE};

ADI_PWM_COMMAND_PAIR PwmCrossoverTable[] =
{
    {ADI_PWM_CMD_SET_CROSSOVER, (void*) &pwm_CrossA_struct},
    {ADI_PWM_CMD_SET_CROSSOVER, (void*) &pwm_CrossB_struct},
    {ADI_PWM_CMD_SET_CROSSOVER, (void*) &pwm_CrossC_struct}
};
adi_pwm_Control(PwmCrossoverTable);
```

Gate Chopping

Listing 13-5 sets gate chopping frequency, and enables or disables gate chopping on the high or low sides.

Listing 13-5. Gate Chopping Commands

```
/* Command to Set Gate Chopping Frequency for BF51x Family */
adi_pwm_Control(ADI_PWM_CMD_SET_GATE_CHOPPING_FREQ,
     (void*) 0x30);
```

```
/* Command to Set Gate Chopping Frequency for BF50x Family */
ADI_PWM_NUMBER_AND_VALUE pwm0ChopFreq = { ADI_PWM_0, 0x30 };
adi_pwm_Control(ADI_PWM_CMD_SET_GATE_CHOPPING_FREQ,
     (void*) &pwm0ChopFreq);

/* command to enable gate chopping on low side */
adi_pwm_Control(ADI_PWM_CMD_SET_GATE_ENABLE_LOW,
     ADI_PWM(x)_ENABLE);

/* command to enable gate chopping on high side */
adi_pwm_Control(ADI_PWM_CMD_SET_GATE_ENABLE_HIGH,
     ADI_PWM(x)_ENABLE);
```

Channel Enable/Disable (Individual)

Listing 13-6 enables or disables individual channels.

Listing 13-6. Individual Channel – Enable/Disable

```
/* ADI_PWM_CHANNEL_STATUS structures used to enable the channels
individually */
ADI_PWM_CHANNEL_STATUS pwm_EnableAH_struct =
     {ADI_PWM_CHANNEL_AH, ADI_PWM(x)_ENABLE};
ADI_PWM_CHANNEL_STATUS pwm_EnableBH_struct =
     {ADI_PWM_CHANNEL_BH, ADI_PWM(x)_ENABLE};
ADI_PWM_CHANNEL_STATUS pwm_EnableCH_struct =
     {ADI_PWM_CHANNEL_CH, ADI_PWM(x)_ENABLE};
ADI_PWM_CHANNEL_STATUS pwm_EnableAL_struct =
     {ADI_PWM_CHANNEL_AL, ADI_PWM(x)_ENABLE};
ADI_PWM_CHANNEL_STATUS pwm_EnableBL_struct =
     {ADI_PWM_CHANNEL_BL, ADI_PWM(x)_ENABLE};
ADI_PWM_CHANNEL_STATUS pwm_EnableCL_struct =
     {ADI_PWM_CHANNEL_CL, ADI_PWM(x)_ENABLE};
```

```
/* ADI_PWM_CHANNEL_STATUS structures used to disable the channels
individually */
ADI_PWM_CHANNEL_STATUS pwm_DisableAH_struct =
     {ADI_PWM_CHANNEL_AH, ADI_PWM(x)_DISABLE};
ADI_PWM_CHANNEL_STATUS pwm_DisableBH_struct =
     {ADI_PWM_CHANNEL_BH, ADI_PWM(x)_DISABLE};
ADI_PWM_CHANNEL_STATUS pwm_DisableCH_struct =
     {ADI_PWM_CHANNEL_CH, ADI_PWM(x)_DISABLE};
ADI_PWM_CHANNEL_STATUS pwm_DisableAL_struct =
     {ADI_PWM_CHANNEL_AL, ADI_PWM(x)_DISABLE};
ADI_PWM_CHANNEL_STATUS pwm_DisableBL_struct =
     {ADI_PWM_CHANNEL_BL, ADI_PWM(x)_DISABLE};
ADI_PWM_CHANNEL_STATUS pwm_DisableCL_struct =
     {ADI_PWM_CHANNEL_CL, ADI_PWM(x)_DISABLE};
/* Command-Pair Table to enable channel pairs A and B,
     disable C  */
ADI_PWM_COMMAND_PAIR ChannelEnableTable[] =
{
    {ADI_PWM_CMD_SET_CHANNEL_ENABLE,
                     (void*) &pwm_EnableAH_struct},
    {ADI_PWM_CMD_SET_CHANNEL_ENABLE,
                     (void*) &pwm_EnableBH_struct},
    {ADI_PWM_CMD_SET_CHANNEL_ENABLE,
                     (void*) &pwm_DisableCH_struct},
    {ADI_PWM_CMD_SET_CHANNEL_ENABLE,
                     (void*) &pwm_EnableAL_struct},
    {ADI_PWM_CMD_SET_CHANNEL_ENABLE,
                     (void*) &pwm_EnableBL_struct},
    {ADI_PWM_CMD_SET_CHANNEL_ENABLE,
                     (void*) &pwm_DisableCL_struct},
};
adi_pwm_Control(ChannelDisableTable);
```

Low Side Invert

Listing 13-7 enables or disables low side invert.

Listing 13-7. Low Side Invert – Enable/Disable

```
/* ADI_PWM_CHANNEL_STATUS structures used to enable low side
invert for individual channel pairs */
ADI_PWM_CHANNEL_STATUS pwm_lsi_A_Enable =
     {ADI_PWM_CHANNEL_A, ADI_PWM(x)_ENABLE};
ADI_PWM_CHANNEL_STATUS pwm_lsi_B_Enable =
     {ADI_PWM_CHANNEL_B, ADI_PWM(x)_ENABLE};
ADI_PWM_CHANNEL_STATUS pwm_lsi_C_Enable =
     {ADI_PWM_CHANNEL_C, ADI_PWM(x)_ENABLE};

/* ADI_PWM_CHANNEL_STATUS structures used to disable low side
invert for individual channel pairs */
ADI_PWM_CHANNEL_STATUS pwm_lsi_A_Disable =
     {ADI_PWM_CHANNEL_A, ADI_PWM(x)_DISABLE};
ADI_PWM_CHANNEL_STATUS pwm_lsi_B_Disable =
     {ADI_PWM_CHANNEL_B, ADI_PWM(x)_DISABLE};
ADI_PWM_CHANNEL_STATUS pwm_lsi_C_Disable =
     {ADI_PWM_CHANNEL_C, ADI_PWM(x)_DISABLE};

/* Command-Pair Table to enable low side invert  for channels A
and B, but not C* /
ADI_PWM_COMMAND_PAIR LowSideInvertTable[] =
{
    {ADI_PWM_CMD_SET_LOW_SIDE_INVERT,
           (void*) &pwm_lsi_A_Enable},
    {ADI_PWM_CMD_SET_LOW_SIDE_INVERT,
           (void*) &pwm_lsi_B_Enable},
    {ADI_PWM_CMD_SET_LOW_SIDE_INVERT,
           (void*) &pwm_lsi_C_Disable}
}
adi_pwm_Control(LowSideInvertTable);
```

External Sync Pulse

Listing 13-8 enables an external sync pulse and configures it as synchronous or asynchronous to the internal system clock. The listing also enables the sync pulse to be output on a pin.

Listing 13-8. External Sync Pulse Commands

```
/* Command to enable external sync pulse */
adi_pwm_Control(ADI_PWM_CMD_SET_SYNC_SOURCE,
     (void*)ADI_PWM(x)_SYNC_SOURCE_EXTERNAL);

/* Command to enable the external sync pulse to be asynchronous
to the internal clock */
adi_pwm_Control(ADI_PWM_CMD_SET_SYNC_SEL,
     (void*)ADI_PWM(x)_SYNC_ASYNCH);

/* Command to enable the external sync pulse to be synchronous to
the internal clock */
adi_pwm_Control( ADI_PWM_CMD_SET_SYNC_SEL,
     (void*)ADI_PWM(x)_SYNC_SYNCH);

/* Command to enable the sync pulse to be output on a pin */
adi_pwm_Control(ADI_PWM_CMD_SET_SYNC_OUT_ENABLE,
     (void*)ADI_PWM(x)_ENABLE);
```

Trip and Sync Interrupts

Listing 13-9 is mainly for informational purposes, as these commands are not usually required. An interrupt is automatically enabled for a trip or sync event when a callback is installed for that event. The interrupt is disabled when the callback is removed for the event. The interrupt condition is cleared by the interrupt handler, when the trip or sync interrupt occurs.

Listing 13-9. Trip and Sync Interrupt Commands

```
/* Command to enable Trip Interrupt - Happens when callback is
installed */
adi_pwm_Control(ADI_PWM_CMD_TRIP_INT_ENABLE,
     (void*)ADI_PWM(x)_ENABLE);

/* Command to disable Trip Interrupt - Happens when callback is
removed. */
adi_pwm_Control(ADI_PWM_CMD_TRIP_INT_ENABLE,
     (void*)ADI_PWM(x)_DISABLE);

/* Command to enable Sync Interrupt - Happens when callback is
installed. */
adi_pwm_Control(ADI_PWM_CMD_SYNC_INT_ENABLE,
     (void*)ADI_PWM(x)_ENABLE);

/* Command to disable Sync Interrupt - Happens when callback is
removed. */
adi_pwm_Control(ADI_PWM_CMD_SYNC_INT_ENABLE,
     (void*)ADI_PWM(x)_DISABLE);

/* ADSP-BF51x Command to Clear Sync Interrupt - PWM Sync Handler
does this automatically */
adi_pwm_Control(ADI_PWM_CMD_CLEAR_SYNC_INT, (void*)1);
```

```
/* ADSP-BF50x Command to Clear PWM 0 Sync Interrupt - PWM Sync
Handler does this automatically */
adi_pwm_Control(ADI_PWM_CMD_CLEAR_SYNC_INT, ADI_PWM_0);

/* ADSP-BF51x Command to Clear Trip Interrupt - PWM Trip Handler
does this automatically */
adi_pwm_Control(ADI_PWM_CMD_CLEAR_SYNC_INT, (void*)1);

/* ADSP-BF50x Command to Clear PWM 1 Trip Interrupt - PWM Sync
Handler does this automatically */
adi_pwm_Control(ADI_PWM_CMD_CLEAR_SYNC_INT, ADI_PWM_1);
```

Change the IVG Level of the Trip or Sync Interrupt

Listing 13-10 demonstrates the command pairs that must be included in a command-pair table that is passed to the initialization function, in order to change the IVG level of the trip and sync interrupts. In this example, the IVG is set to 9. The IVG level should not be changed after initialization.

Listing 13-10. Command-Pair Table Commands for IVG Level Change

```
/* ADSP-BF51x Family - Set Trip=IVG9; Sync=IVG9 */
/* Pass this in command-pair table to adi_pwm_Init, to change the
IVG level of the Trip interrupt */
{ADI_PWM_CMD_SET_TRIP_IVG, (void*)9}

/* Pass this in command-pair table to adi_pwm_Init, to change the
IVG level of the Sync interrupt */
 {ADI_PWM_CMD_SET_SYNC_IVG, (void*)9}

/* ADSP-BF50x Family - Set PWM0 Trip=IVG7; Sync=IVG11 */
 ADI_PWM_NUMBER_AND_VALUE   pwm0_TRIPIVG_struct={ADI_PWM_0,7};
 ADI_PWM_NUMBER_AND_VALUE   pwm0_SYNCIVG_struct={ADI_PWM_0,11};
 {ADI_PWM_CMD_SET_SYNC_IVG, (void*) &pwm0_SYNCIVG_struct};
 {ADI_PWM_CMD_SET_TRIP_IVG, (void*) &pwm0_TRIPIVG_struct};

/* ADSP-BF50x Family - Set PWM0 Trip=IVG7; Sync=IVG11 */
 ADI_PWM_NUMBER_AND_VALUE   pwm1_TRIPIVG_struct={ADI_PWM_1,7};
 ADI_PWM_NUMBER_AND_VALUE   pwm1_SYNCIVG_struct={ADI_PWM_1,11};
 {ADI_PWM_CMD_SET_SYNC_IVG, (void*) &pwm1_SYNCIVG_struct};
 {ADI_PWM_CMD_SET_TRIP_IVG, (void*) &pwm1_TRIPIVG_struct};
```

Trip Input Signal

Listing 13-11 enables or disables the trip input signal.

Caution! Use this command with care. The trip input signal is used to detect a hardware failure and to turn off all channel signals.

Listing 13-11. Trip Input Signal – Enable/Disable

```
/* Command to enable Trip Input */
adi_pwm_Control(ADI_PWM_CMD_SET_TRIP_INPUT_ENABLE,
     (void*) ADI_PWM(x)_ENABLE);

/* Command to disable Trip Input */
adi_pwm_Control(ADI_PWM_CMD_SET_TRIP_INPUT_ENABLE,
     (void*) ADI_PWM(x)_DISABLE);
```

PWM Enable/Disable

Listing 13-12 shows the command to enable or disable the PWM. Enabling the PWM begins motor control operation using the parameters that have been specified by adi_pwm_Init. This command should be passed after the PWM is configured, and callbacks have been installed.

Listing 13-12. PWM Enable/Disable

```
/* Command to enable the PWM */
adi_pwm_Control(ADI_PWM_CMD_SET_PWM_ENABLE,
     (void*) ADI_PWM(x)_ENABLE);

/* Command to disable the PWM */
adi_pwm_Control(ADI_PWM_CMD_SET_PWM_ENABLE,
     (void*) ADI_PWM(x)_DISABLE);
```

PWM Service Application Programming Interface (API)

This section provides the details of the data structures and functions within the PWM service application program interface (API).

Notation and Naming Conventions

To safeguard against conflicts with other software libraries provided by Analog Devices, or other sources, the PWM service uses an unambiguous naming convention in which enumeration values and `typedef` statements use the `ADI_PWM_` prefix. Functions and global variables use the lowercase `adi_pwm_` equivalent.

Each function within the PWM service API returns an error code of the type `ADI_PWM_RESULT`. Like the other system services, a return value of zero (`0=ADI_PWM_RESULT_SUCCESS`) indicates that no error has occurred during the function call. Any nonzero value indicates the specific type of error that has occurred. The error codes for the PWM service, unique from those of the other system services, are defined in the `adi_pwm.h` include file, so the cause of the error can be determined from looking up the error code in that file.

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

Name – 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

PWM Service API Functions

This section describes the PWM functions that are available to the application. These functions read and write to the hardware registers so the application developer does not have to study the details of each register. Below is a list of the functions in the PWM API.

- adi_pwm_Init
- adi_pwm_Terminate
- adi_pwm_Control
- adi_pwm_InstallCallback
- adi_pwm_RemoveCallback

Each API function returns a value of type `ADI_PWM_RESULT` which indicates the success or failure of the function call. The result codes are defined in a table in the subsection that describes the structures and data types in the API.

adi_pwm_Init

Description

The `adi_pwm_Init()` function initializes the pulse-width modulation service as described in the initialization section.

Prototype

```
ADI_PWM_RESULT adi_pwm_Init(
  const ADI_PWM_COMMAND_PAIR *table,
  void *pCriticalRegionArg
);
```

Arguments

The function accepts two arguments: a `ADI_PWM_COMMAND_PAIR` which specifies the commands for the function to process, and a `void*`, which is the critical region parameter.

Return Value

`ADI_PWM_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See "ADI_PWM_ RESULT" on page 13-48.

adi_pwm_Terminate

Description

The `adi_pwm_Terminate()` function terminates the pulse-width modulation service as described in the termination section.

Prototype

```
ADI_PWM_RESULT adi_pwm_Terminate(void);
```

Arguments

The function accepts no arguments.

Return Value

`ADI_PWM_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See "ADI_PWM_RESULT" on page 13-48.

adi_pwm_Control

Description

The `adi_pwm_Control()` function enables the pulse-width modulation control/status registers to be configured or queried according to specified command-value pairs, in one of three ways:

1. A single command-value pair is passed.

```
adi_pwm_Control(ADI_PWM(x)_CMD_SET_POLARITY,
 (void*)ADI_PWM(x)_POLARITY_HIGH,);
```

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

```
ADI_PWM_COMMAND_PAIR cmd =
 {ADI_PWM_CMD_SET_POLARITY(void*) ADI_PWM(x)_POLARITY_
HIGH};
adi_pwm_Control(ADI_PWM_CMD_PAIR,(void*)&cmd);
```

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

```
ADI_PWM_COMMAND_PAIR table[] = {
  {ADI_PWM_CMD_SET_POLARITY,(void*) ADI_PWM(x)_POLARITY_
HIGH},
  {ADI_PWM_CMD_SET_SYNC_OUT_ENABLE,(void*) ADI_PWM(x)_
ENABLE},
  {ADI_PWM_CMD_END,0}
};
```

Prototype

```
ADI_PWM_RESULT adi_pwm_Control(
  ADI_PWM_COMMAND Command,
  void *Value
);
```

Arguments

Refer to the `ADI_PWM_COMMAND` for the complete list of commands and associated values.

`Command`	`ADI_PWM_COMMAND` enumeration value specifies the meaning of the associated value argument.
`Value`	The required value. See "ADI_PWM_ RESULT" on page 13-48.

Return Value

`ADI_PWM_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See "ADI_PWM_ RESULT" on page 13-48.

adi_pwm_InstallCallback

Description

The `adi_pwm_InstallCallback()` function installs a callback for a pulse-width modulation event.

Prototype

```
ADI_PWM_RESULT adi_pwm_InstallCallback(
  ADI_PWM_EVENT_ID       EventID,
  void                   *ClientHandle,
  ADI_DCB_HANDLE         DCBHandle,
  ADI_DCB_CALLBACK_FN    ClientCallback
);
```

Arguments

The caller provides four parameters to the function.

Data Type	Name	Description
`ADI_PWM_EVENT_ID`	`EventID`	ID of the interrupting event (See "ADI_PWM_EVENT_ID" on page 13-46.)
`Void`	`ClientHandle`	Application data passed to call-back
`ADI_DCB_HANDLE`	`DCBHandle`	Deferred callback service handle
`ADI_DCB_CALLBACK_FN`	`ClientCallback`	Name of client callback function

Return Value

`ADI_PWM_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See "ADI_PWM_RESULT" on page 13-48.

adi_pwm_RemoveCallback

Description

The `adi_pwm_RemoveCallback()` function removes a callback for a pulse-width modulation event.

Calling `adi_pwm_RemoveCallback` from within a callback routine is not supported and will result in undefined behavior.

Prototype

```
ADI_PWM_RESULT adi_pwm_RemoveCallback(
            ADI_PWM_EVENT_ID EventID
);
```

Arguments

`EventID`	`ADI_PWM_EVENT` enumeration value specifies the meaning of the associated value argument.

Return Value

`ADI_PWM_RESULT_SUCCESS`	No error has been encountered.
Any other value	Error has occurred. See "ADI_PWM_RESULT" on page 13-48.

PWM Service API Data Types and Enumerations

This section defines the data types and enumerations that are exposed to the application by the application programming interface (API), for selecting command-pair values. The constant value ADI_PWM_ENUMERATION_START is defined in the system services include file, services.h. It defines the base of a unique bank of enumeration values for the PWM service to use. Some of the following enumerations begin at 0. The rest are based on the value ADI_PWM_ENUMERATION_START, to identify them as PWM enumerations.

The PWM number must be specified for the ADSP-BF50x Blackfin family, which has two identical PWMs. Some of the data structures and enumeration names vary between the ADSP-BF51x and ADSP-BF50x Blackfin processor families.

ADI_PWM_CHANNEL_STATUS

The channel and value structure is used as a command argument to be passed along with a command to the adi_pwm_Init or adi_pwm_Control functions. A channel is combined with a channel-specific value, for use with the commands ADI_PWM_CMD_SET_CHANNEL_ENABLE, ADI_PWM_CMD_SET_CROSSOVER, and ADI_PWM_CMD_SET_LOW_SIDE_INVERT. This structure indicates the channel ID and the enable status of the specified signal for that channel. In the presence of multiple PWMs, the PWM number is embedded in the enable status structure.

Table 13-1. ADI_PWM_CHANNEL_STATUS

Type	Name	Description
`ADI_PWM_CHANNEL`	`Channel`	Indicates the channel ID
`ADI_PWM_ENABLE_STATUS`	`Status`	Indicates the channel specific status, and the PWM number, in the presence of multiple PWMs

ADI_PWM_CHANNEL_DUTY_CYCLE

The channel and duty cycle structure is used as a command argument to be passed along with the `ADI_PWM_SET_DUTY_CYCLE` command to the adi_pwm_Init or adi_pwm_Control functions. In this structure, the channel ID is combined with the channel specific, unsigned integer duty cycle value for that channel.

Table 13-2. ADI_PWM_CHANNEL_DUTY_CYCLE

Data Type	Name	Description
`ADI_PWM_NUMBER`	`PwmNumber`	Indicates which PWM (ADSP-BF50x only)
`ADI_PWM_CHANNEL`	`Channel`	Indicates the channel ID
`u32`	`Value`	Indicates the channel specific value

ADI_PWM_COMMAND_PAIR

A command-pair structure combines a command with a value. The value can be a data structure which combines multiple values, such as channel ID and Boolean for enabling or disabling a feature of that channel.

Table 13-3. ADI_PWM_COMMAND_PAIR

Data Type	Name	Description
`ADI_PWM_COMMAND`	`Kind`	Indicates the command to execute
`Void*`	`Value`	The command argument, which can be cast to any other data type

ADI_PWM_NUMBER_AND_CHANNEL_STATUS

When requesting the enable status, crossover status, or low side invert status of a channel, it is necessary to specify the channel and which PWM the requested status is for, if there are multiple PWMs. This structure allows the PWM number to be specified in the `PwmNumber` field, and an `ADI_PWM_CHANNEL_STATUS` structure to be passed in the `ChannelStatus` field, so that the proper information can be returned in the `Status` field which is inside the `ADI_PWM_CHANNEL_STATUS` structure named `ChannelStatus`.

Table 13-4. ADI_PWM_NUMBER_AND_CHANNEL_STATUS

Data Type	Name	Description
`ADI_PWM_NUMBER`	`PwmNumber`	Specifies PWM0 or PWM1
`ADI_PWM_CHANNEL_STATUS`	`ChannelStatus`	Enable channel status

ADI_PWM_NUMBER_AND_ENABLE_STATUS

When requesting the status of switch reluctance or gate enable modes, if there are multiple PWMs, it is necessary to specify which PWM the requested status is for. This structure allows the PWM number to be specified in the `PwmNumber` field, so that the proper enable status can be returned in the `Status` field.

Table 13-5. ADI_PWM_NUMBER_AND_ENABLE_STATUS

Data Type	Name	Description
`ADI_PWM_NUMBER`	`PwmNumber`	Specifies PWM0 or PWM1
`ADI_PWM_ENABLE_STATUS`	`Status`	Enable status

ADI_PWM_NUMBER_AND_VALUE

A simple `u32` type value is passed to the adi_pwm_Init or adi_pwm_Control functions to set the sync period, pulse width, or dead time of a PWM. In the presence of multiple PWMs, the PWM number must also be specified. This data structure combines an `ADI_PWM_NUMBER` enumeration with a `u32` value, such as the dead time, sync pulse width, or the sync period.

Table 13-6. ADI_PWM_NUMBER_AND_VALUE

Data Type	Name	Description
`ADI_PWM_NUMBER`	`PwmNumber`	Specifies PWM0 or PWM1
`void*`	`Value`	Value to set

ADI_PWM_PORT_MAP

The ADI_PWM_PORT_MAP structure has eight bit fields for the ADSP-BF51x, or sixteen bit fields for the ADSP-BF50x, each named after one of the eight multiplexed signals on each PWM. Each field must be set to one of the two ADI_PWM_PORT_MUX enumerations, either ADI_PWM_PORT_MUX_PRI or ADI_PWM_PORT_MUX_SEC, to select the primary or secondary port mapping. This structure must be passed to the adi_pwm_Init function, following the ADI_PWM_CMD_SET_PORT_MUX command.

Table 13-7. ADI_PWM_PORT_MAP

Data Type	Name	Description
Bit Field	AH_MUX	Port mapping for PWM 0 AH signal
Bit Field	AL_MUX	Port mapping for PWM 0 AL signal
Bit Field	BH_MUX	Port mapping for PWM 0 BH signal
Bit Field	BL_MUX	Port mapping for PWM 0 BL signal
Bit Field	CH_MUX	Port mapping for PWM 0 CH signal
Bit Field	CL_MUX	Port mapping for PWM 0 CL signal
Bit Field	SYNC_MUX	Port mapping for PWM 0 SYNC signal
Bit Field	TRIP_MUX	Port mapping for PWM 0 TRIP signal
Bit Field	AH_MUX	Port mapping for PWM 1 AH signal (ADSP-BF50x only)
Bit Field	AL_MUX	Port mapping for PWM 1 AL signal (ADSP-BF50x only)
Bit Field	BH_MUX	Port mapping for PWM 1 BH signal (ADSP-BF50x only)
Bit Field	BL_MUX	Port mapping for PWM 1 BL signal (ADSP-BF50x only)
Bit Field	CH_MUX	Port mapping for PWM 1 CH signal (ADSP-BF50x only)
Bit Field	CL_MUX	Port mapping for PWM 1 CL signal (ADSP-BF50x only)
Bit Field	SYNC_MUX	Port mapping for PWM 1 SYNC signal (ADSP-BF50x only)
Bit Field	TRIP_MUX	Port mapping for PWM 1 TRIP signal (ADSP-BF50x only)

ADI_PWM_CHANNEL

Channel enumerations are passed as part of a command argument to specify which channel the command is intended for.

Table 13-8. ADI_PWM_CHANNEL

Name	Description
`ADI_PWM_CHANNEL_A`	Channel pair AH, AL
`ADI_PWM_CHANNEL_B`	Channel pair BH, BL
`ADI_PWM_CHANNEL_C`	Channel pair CH, CL
`ADI_PWM_CHANNEL_AH`	High side output channel A
`ADI_PWM_CHANNEL_AL`	Low side output channel A
`ADI_PWM_CHANNEL_BH`	High side output channel B
`ADI_PWM_CHANNEL_BL`	Low side output channel B
`ADI_PWM_CHANNEL_CH`	High side output channel C
`ADI_PWM_CHANNEL_CL`	Low side output channel C

ADI_PWM_COMMAND

The `ADI_PWM_COMMAND` enumeration type describes the command type in an `ADI_PWM_COMMAND_PAIR` structure. Each of the PWM service commands can be found in the PWM service API header file, `adi_pwm.h`.

Command enumerations for the PWM service begin with the value `ADI_PWM_ENUMERATION_START` (defined in `include/services/services.h`) for easy identification.

Table 13-9 lists the available commands and the context for their use.

Table 13-9. ADI_PWM_COMMAND

Name	Description	Argument
ADI_PWM_CMD_END	End of a command-pair table	
ADI_PWM_CMD_PAIR	Passing a command pair	
ADI_PWM_CMD_TABLE	Passing a command-pair table. See "Programming Examples" on page 13-11.	
ADI_PWM_CMD_SET_CHANNEL_ENABLE	Enable/disable a PWM signal channel in PWMSEG. See "Programming Examples" on page 13-11.	pADI_PWM_CHANNEL_STATUS (specifies PWM number if applicable)
ADI_PWM_CMD_SET_DUTY_CYCLE	Set the duty cycle in PWMCHA, PWMCHB, PWMCHC registers or in PWMCHAL, PWMCHBL, PWMCHCL registers in SR mode. See "Programming Examples" on page 13-11.	pADI_PWM_CHANNEL_DUTY_CYCLE (specifies PWM number if applicable)
ADI_PWM_CMD_SET_DEAD_TIME	Set PWM switching dead time value in PWMDT register. See "Programming Examples" on page 13-11.	u32 (ADSP-BF51x) ADI_PWM_NUMBER_AND_VALUE (ADSP-BF50x)
ADI_PWM_CMD_SET_PERIOD	Set the period for sync pulse in the PWMTM register. See "Programming Examples" on page 13-11.	u32 (ADSP-BF51x) ADI_PWM_NUMBER_AND_VALUE (ADSP-BF50x)
ADI_PWM_CMD_SET_SYNC_PULSE_WIDTH	Set sync pulse width in PWMSYNCWT. See "Programming Examples" on page 13-11.	u32 (ADSP-BF51x) ADI_PWM_NUMBER_AND_VALUE (ADSP-BF50x) (Sync pulse must be greater than two system clock periods.)
ADI_PWM_CMD_SET_CROSSOVER	Enable/disable crossover mode for specified channel in PWMSEG register. See "Programming Examples" on page 13-11.	pADI_PWM_CHANNEL_STATUS

Table 13-9. ADI_PWM_COMMAND (Cont'd)

Name	Description	Argument
ADI_PWM_CMD_SET_POLARITY	Set the polarity for a signal in PWMCTRL. See "Programming Examples" on page 13-11.	ADI_PWM_POLARITY
ADI_PWM_CMD_SET_GATE_ CHOPPING_FREQ	Set gate drive chopping frequency in PWMGATE. See "Programming Examples" on page 13-11.	u32 (ADSP-BF51x) ADI_PWM_NUMBER_AND_VALUE (ADSP-BF50x)
ADI_PWM_CMD_SET_GATE_ ENABLE _LOW	Enable/disable gate chopping for low side in PWMGATE. See "Programming Examples" on page 13-11.	ADI_PWM_ENABLE_STATUS
ADI_PWM_CMD_SET_GATE_ ENABLE _HIGH	Enable/disable gate chopping for high side in PWMGATE. See "Programming Examples" on page 13-11.	ADI_PWM_ENABLE_STATUS
ADI_PWM_CMD_SET_LOW_ SIDE_INVERT	Set low side invert in PWMLSI register for a given channel. See "Programming Examples" on page 13-11.	ADI_PWM_CHANNEL_STATUS
ADI_PWM_CMD_SET_SWITCH_ RELUCTANCE	Enable/disable switch reluctance mode in PWMCTRL. See "Programming Examples" on page 13-11.	ADI_PWM_ENABLE_STATUS
ADI_PWM_CMD_SET_SYNC_ INT_ENABLE	Enables/disables sync pulse interrupt input in PWMCTRL. See "Programming Examples" on page 13-11.	ADI_PWM_ENABLE_STATUS
ADI_PWM_CMD_SET_SYNC_ SOURCE	Set sync pulse source to internal/external in PWMCTRL. See "Programming Examples" on page 13-11.	ADI_PWM_SYNC_SOURCE
ADI_PWM_CMD_SET_SYNC_ SELECT	Set external sync select in PWMCTRL 0=asynch, 1=sync def=1. See "Programming Examples" on page 13-11.	ADI_PWM_SYNC_SELECT

Table 13-9. ADI_PWM_COMMAND (Cont'd)

Name	Description	Argument
ADI_PWM_CMD_SET_PORT_MUX	Selects either primary or alternate port mux mapping for each signal	ADI_PWM_PORT_MUX
ADI_PWM_CMD_SET_SYNC_OUT_ENABLE	Enable/disable SyncEnable in PWMCTRL for sync pulse to appear on output pin. See "Programming Examples" on page 13-11.	ADI_PWM_ENABLE_STATUS
ADI_PWM_CMD_SET_TRIP_INT_ENABLE	Enables/disables trip pulse interrupt input in PWMCTRL. See "Programming Examples" on page 13-11.	ADI_PWM_ENABLE_STATUS
ADI_PWM_CMD_SET_TRIP_INPUT_ENABLE	Enables/disable trip input. See "Programming Examples" on page 13-11.	ADI_PWM_ENABLE_STATUS
ADI_PWM_CMD_SET_UPDATE_MODE	Set PWM operating mode in PWMCTRL - single/double update. See "Programming Examples" on page 13-11.	ADI_PWM_UPDATE_MODE
ADI_PWM_CMD_SET_PWM_ENABLE	Enables a software shutdown of all 6 channels	ADI_PWM_ENABLE_STATUS
ADI_PWM_CMD_SET_TRIP_IVG	Change the IVG level of the trip interrupt. See "Programming Examples" on page 13-11.	&u32 (ADSP-BF51x) pADI_PWM_NUMBER_AND_VALUE (ADSP-BF50x)
ADI_PWM_CMD_SET_SYNC_IVG	Change the IVG level of the sync interrupt. See "Programming Examples" on page 13-11.	&u32 (ADSP-BF51x) pADI_PWM_NUMBER_AND_VALUE (ADSP-BF50x)
ADI_PWM_CMD_CLEAR_SYNC_INT	Clear sync interrupt in PWMSTAT. See "Programming Examples" on page 13-11.	N/A (ADSP-BF51x) PWM_NUMBER (ADSP-BF50x)

Table 13-9. ADI_PWM_COMMAND (Cont'd)

Name	Description	Argument
ADI_PWM_CMD_CLEAR_TRIP_INT	Clear trip interrupt in PWMSTAT. See "Programming Examples" on page 13-11.	N/A (ADSP-BF51x) PWM_NUMBER (ADSP-BF50x)
ADI_PWM_CMD_GET_CHANNEL_ENABLE	Get enable status of a channel from PWMSEG	pADI_PWM_CHANNEL_STATUS (ADSP-BF51x) pADI_PWM_NUMBER_AND_CHANNEL_STATUS (ADSP-BF50x)
ADI_PWM_CMD_GET_DUTY_CYCLE	Get duty cycle from PWMCHA, PWMCHB, PWMCHC, PWMCHAL, PWMCHBL, PWMCHCL registers, in SR mode only	pADI_PWM_CHANNEL_DUTY_CYCLE
ADI_PWM_CMD_GET_DEAD_TIME	Get switching dead time from PWMDT register	&u32 (ADSP-BF51x) pADI_PWM_NUMBER_AND_VALUE (ADSP-BF50x)
ADI_PWM_CMD_GET_PERIOD	Get the period for the sync pulse from PWMTM register	&u32 (ADSP-BF51x) pADI_PWM_NUMBER_AND_VALUE (ADSP-BF50x)
ADI_PWM_CMD_GET_SYNC_PULSE_WIDTH	Get sync pulse width in PWMSYNCWT register	&u32 (ADSP-BF51x) pADI_PWM_NUMBER_AND_VALUE (ADSP-BF50x)
ADI_PWM_CMD_GET_CROSSOVER	Get crossover mode for specified channel PWMSEG register	pADI_PWM_CHANNEL_STATUS (ADSP-BF51x) pADI_PWM_NUMBER_AND_CHANNEL_STATUS (ADSP-BF50x)
ADI_PWM_CMD_GET_POLARITY	Get PWM polarity from PWMSTAT	&ADI_PWM_POLARITY
ADI_PWM_CMD_GET_GATE_CHOPPING_FREQ	Get gate drive chopping frequency from PWMGATE	u32
ADI_PWM_CMD_GET_GATE_ENABLE_HIGH	Get gate chopping enable status for high side from PWMGATE	&ADI_PWM_ENABLE_STATUS

Table 13-9. ADI_PWM_COMMAND (Cont'd)

Name	Description	Argument
ADI_PWM_CMD_GET_GATE_ENABLE_LOW	Get gate chopping enable status for low side from PWMGATE	&ADI_PWM_ENABLE_STATUS
ADI_PWM_CMD_GET_LOW_SIDE_INVERT	Get low side invert for a channel in PWMLSI register	pADI_PWM_CHANNEL_STATUS
ADI_PWM_CMD_GET_SWITCH_RELUCTANCE	Get SR mode PWMSTAT	&ADI_PWM_ENABLE_STATUS
ADI_PWM_CMD_GET_SYNC_INT	Get sync interrupt from PWMSTAT	&ADI_PWM_ENABLE_STATUS (ADSP-BF51x) pADI_PWM_NUMBER_AND_VALUE (ADSP-BF50x)
ADI_PWM_CMD_GET_TRIP_INT	Get trip interrupt in PWMSTAT	&ADI_PWM_ENABLE_STATUS (ADSP-BF51x) pADI_PWM_NUMBER_AND_VALUE (ADSP-BF50x)
ADI_PWM_CMD_GET_SYNC_SOURCE	Get sync pulse source (internal or external)	&ADI_PWM_SYNC_SOURCE (ADSP-BF51x) pADI_PWM_NUMBER_AND_VALUE (ADSP-BF50x)
ADI_PWM_CMD_GET_SYNC_SEL	Get extern sync select 0=asynch, 1=sync def=1	&ADI_PWM_SYNC_SELECT (ADSP-BF51x) pADI_PWM_NUMBER_AND_VALUE (ADSP-BF50x)
ADI_PWM_CMD_GET_UPDATE_MODE	Get PWM operating mode from PWMSTAT- single/double update	&ADI_PWM_UPDATE_MODE (ADSP-BF51x) pADI_PWM_NUMBER_AND_VALUE (ADSP-BF50x)
ADI_PWM_CMD_GET_PWM_ENABLE	Get software shutdown status (all 6 channel disable)	&ADI_PWM_ENABLE_STATUS (ADSP-BF51x) pADI_PWM_NUMBER_AND_VALUE (ADSP-BF50x)
ADI_PWM_CMD_GET_PHASE	Get PWM phase from PWMSTAT 0=1st half 1=2nd half def=0	&ADI_PWM_ENABLE_STATUS (ADSP-BF51x) pADI_PWM_NUMBER_AND_VALUE (ADSP-BF50x)

Table 13-9. ADI_PWM_COMMAND (Cont'd)

Name	Description	Argument
`ADI_PWM_CMD_GET_TRIP_PIN`	Get trip pin value from `PWMSTAT`	`&ADI_PWM_ENABLE_STATUS` (ADSP-BF51x) `pADI_PWM_NUMBER_AND_VALUE` (ADSP-BF50x)
`ADI_PWM_CMD_GET_OUTPUT`	Get output signal from `PWMSTAT2`	`&u32` (ADSP-BF51x) `pADI_PWM_NUMBER_AND_VALUE` (ADSP-BF50x)

ADI_PWM_ENABLE_STATUS

This enumeration is used in conjunction with a command and sometimes also with a channel ID to indicate whether a particular feature is enabled or disabled. It is used to enable or disable channels, gate chopping, switch reluctance, sync, and trip interrupts. In the presence of multiple PWMs, this enumeration also specifies which PWM a command is intended for.

Table 13-10. ADI_PWM_ENABLE_STATUS

Name	Numeric Value	Description

`ADI_PWM_DISABLE,`
`ADI_PWM0_DISABLE`
	`0`	Disable status for PWM0

`ADI_PWM_ENABLE,`
`ADI_PWM0_ENABLE`
	`1`	Enable status for PWM0

`ADI_PWM1_DISABLE`	`2`	Disable status for PWM1
`ADI_PWM1_ENABLE`	`3`	Enable status for PWM1

ADI_PWM_EVENT_ID

On each PWM module, there are two events which are defined by unique event IDs. In the presence of multiple PWMs, the enumeration name contains a digit to specify which PWM the enumeration applies to.

Table 13-11. ADI_PWM_EVENT_ID

Name	Numeric Offset Value (from ADI_PWM_ ENUMERATION_START)	Description
`ADI_PWM_EVENT_START`, `ADI_PWM_ENUMERATION_START`	`0x00`	Base value for PWM. Defined in `services.h` service enumerations.
`ADI_PWM_EVENT_TRIP`, `ADI_PWM0_EVENT_TRIP`	`0x01`	Trip interrupt event for PWM0
`ADI_PWM_EVENT_SYNC`, `ADI_PWM0_EVENT_SYNC`	`0x02`	Sync interrupt event for PWM0
`ADI_PWM1_EVENT_TRIP`	`0x03`	Trip interrupt event for PWM1
`ADI_PWM1_EVENT_SYNC`	`0x04`	Sync interrupt event for PWM1

ADI_PWM_NUMBER

In the presence of multiple PWMs, an enumeration called `ADI_PWM_NUMBER` is used, sometimes within other data structures, to specify which PWM an action is intended for. The value of `ADI_NUM_PWM` specifies how many PWMs there are. Currently the maximum is two.

Table 13-12. ADI_PWM_NUMBER

Name	Numeric Value	Description
`ADI_PWM_0`	`0`	Specifies PWM0
`ADI_PWM_1`	`1`	Specifies PWM1
`ADI_NUM_PWM`	`2`	Total number of PWMs

ADI_PWM_POLARITY

This enumeration is used in conjunction with the command `ADI_PWM_CMD_SET_POLARITY` to set the polarity for all output signals to active high or active low. In the presence of multiple PWMs, this enumeration also specifies which PWM a command is intended for.

Table 13-13. ADI_PWM_POLARITY

Name	Description
`ADI_PWM_SOURCE_LOW`, `ADI_PWM0_SOURCE_LOW`	Low polarity PWM0
`ADI_PWM_SOURCE_HIGH`, `ADI_PWM0_SOURCE_HIGH`	High polarity PWM0
`ADI_PWM1_SOURCE_LOW`	Low polarity PWM1
`ADI_PWM1_SOURCE_HIGH`	High polarity PWM1

ADI_PWM_PORT_MUX

This enumeration is used to specify which port mux mapping to use, primary or secondary, for each individual signal.

Table 13-14. ADI_PWM_PORT_MUX

Name	Description
`ADI_PWM_PORT_MUX_PRI`	Primary port mux mapping
`ADI_PWM_PORT_MUX_SEC`	Secondary port mux mapping

ADI_PWM_RESULT

Each of the PWM service API functions returns a result code that can be translated by looking into the PWM service API header file, `adi_pwm.h`.

As with all system services, generic success is defined as 0 and generic failure is defined as 1. This allows the calling function to quickly evaluate the return code for a zero or nonzero value. All detailed result codes for the PWM service begin with the value `ADI_PWM_ENUMERATION_START` for easy identification.

The result codes are summarized in Table 13-15.

Table 13-15. ADI_PWM_RESULT

Name	Description
`ADI_PWM_RESULT_INVALID_EVENT_ID`	Specified `EventID` does not exist
`ADI_PWM_RESULT_INTERRUPT_MANAGER_ERROR`	Error returned from interrupt manager
`ADI_PWM_RESULT_ERROR_REMOVING_CALLBACK`	Cannot find callback for given ID
`ADI_PWM_RESULT_NOT_INITIALIZED`	PWM service has not been initialized.
`ADI_PWM_RESULT_CALLBACK_NOT_INSTALLED`	Specified callback was never installed.
`ADI_PWM_RESULT_SERVICE_NOT_SUPPORTED`	PWM service is not supported for this processor
`ADI_PWM_RESULT_ALREADY_INITIALIZED`	PWM service has already been initialized
`ADI_PWM_RESULT_CALLBACK_ALREADY_INSTALLED`	Cannot install the same callback twice
`ADI_PWM_RESULT_INVALID_CHANNEL`	Specified channel does not exist
`ADI_PWM_RESULT_INVALID_COMMAND`	Specified command does not exist
`ADI_PWM_RESULT_INVALID_ENABLE_STATUS`	Invalid parameter passed with enable command
`ADI_PWM_RESULT_INVALID_DUTY_CYCLE`	Invalid duty cycle
`ADI_PWM_RESULT_INVALID_DEAD_TIME`	Invalid switching dead time
`ADI_PWM_RESULT_INVALID_PERIOD`	Invalid period for sync pulse

Table 13-15. ADI_PWM_RESULT (Cont'd)

Name	Description
ADI_PWM_RESULT_INVALID_PULSE_WIDTH	Invalid pulse width
ADI_PWM_RESULT_INVALID_CROSSOVER	Invalid crossover mode
ADI_PWM_RESULT_INVALID_POLARITY	Invalid polarity for output signals
ADI_PWM_RESULT_INVALID_GATE_CHOPPING_FREQ	Invalid gate drive chopping frequency
ADI_PWM_RESULT_INVALID_GATE_ENABLE	Invalid argument passed with gate chopping enable command
ADI_PWM_RESULT_INVALID_LOW_SIDE_INVERT	Invalid parameter passed with low side invert command
ADI_PWM_RESULT_INVALID_SWITCH_RELUCTANCE	Invalid switch reluctance mode
ADI_PWM_RESULT_INVALID_SYNC_INT_ENABLE	Invalid parameter passed with sync pulse interrupt command
ADI_PWM_RESULT_INVALID_SYNC_SOURCE	Invalid parameter passed with sync pulse source command
ADI_PWM_RESULT_INVALID_SYNC_SEL	Invalid external sync select
ADI_PWM_RESULT_INVALID_TRIP_INT_ENABLE	Invalid parameter passed with trip pulse interrupt command
ADI_PWM_RESULT_INVALID_SYNC_OUT_ENABLE	Invalid parameter passed with sync pulse output enable command
ADI_PWM_RESULT_INVALID_UPDATE_MODE	Invalid PWM operating mode
ADI_PWM_RESULT_INVALID_PWM_ENABLE	Invalid PWM enable status
ADI_PWM_RESULT_PORT_MUX_MAPPING_NOT_SET	Port mux mapping not passed to `adi_pwm_Init`
ADI_PWM_RESULT_PERIOD_NOT_SET	Period not passed to `adi_pwm_Init`
ADI_PWM_RESULT_DEAD_TIME_NOT_SET	Dead time not passed to `adi_pwm_Init`
ADI_PWM_RESULT_DUTY_CYCLE_A_NOT_SET	Duty cycle not passed to `adi_pwm_Init` for channel A.
ADI_PWM_RESULT_DUTY_CYCLE_B_NOT_SET	Duty cycle not passed to `adi_pwm_Init` for channel B.

Table 13-15. ADI_PWM_RESULT (Cont'd)

Name	Description
`ADI_PWM_RESULT_DUTY_CYCLE_C_NOT_SET`	Duty cycle not passed to `adi_pwm_Init` for channel C.
`ADI_PWM_RESULT_SYNC_PULSE_WIDTH_NOT_SET`	Sync pulse width not passed to `adi_pwm_Init`.
`ADI_PWM_RESULT_OPERATING_MODE_NOT_SET`	Operating mode not passed to `adi_pwm_Init`.
`ADI_PWM_RESULT_CHANNEL_ENABLE_NOT_SET`	Channel enable command not passed to `adi_pwm_Init`.
`ADI_PWM_RESULT_POLARITY_NOT_SET`	Polarity not passed to `adi_pwm_Init`.
`ADI_PWM_RESULT_SWITCH_RELUCTANCE_IS_ACTIVE`	Command meaningless in switch reluctance mode
`ADI_PWM_RESULT_SWITCH_RELUCTANCE_NOT_ACTIVE`	Command meaningless when not in switch reluctance mode

ADI_PWM_SYNC_SEL

When the synchronization pulse source is external, this enumeration is used in conjunction with the command `ADI_PWM_CMD_SET_SYNC_SELECT` to set the external synchronization pulse to synchronous or asynchronous. In the presence of multiple PWMs, this enumeration also specifies which PWM a command is intended for.

Table 13-16. ADI_PWM_SYNC_SEL

Name	Description
`ADI_PWM_SYNC_ASYNC`, `ADI_PWM0_SYNC_ASYNC`	External sync source for PWM0 is asynchronous.
`ADI_PWM_SYNC_SYNC`, `ADI_PWM0_SYNC_SYNC`	External sync source for PWM0 is synchronous.
`ADI_PWM1_SYNC_ASYNC`	External sync source for PWM1 is asynchronous.
`ADI_PWM1_SYNC_SYNC`	External sync source for PWM1 is synchronous.

ADI_PWM_SYNC_SOURCE

This enumeration is used in conjunction with the command `ADI_PWM_CMD_SET_SYNC_SOURCE` to set the synchronization pulse source to either internal or external. In the presence of multiple PWMs, this enumeration also specifies which PWM a command is intended for.

Table 13-17. ADI_PWM_SYNC_SOURCE

Name	Description
`ADI_PWM_SOURCE_INTERNAL`, `ADI_PWM0_SOURCE_INTERNAL`	Internal sync source PWM0
`ADI_PWM_SOURCE_EXTERNAL`, `ADI_PWM0_SOURCE_EXTERNAL`	External sync source PWM0
`ADI_PWM1_SOURCE_INTERNAL`	Internal sync source PWM1
`ADI_PWM1_SOURCE_EXTERNAL`	External sync source PWM1

ADI_PWM_UPDATE_MODE

This enumeration is used to specify the operating mode of the PWM: single update or double update. In the presence of multiple PWMs, this enumeration also specifies which PWM a command is intended for.

Table 13-18. ADI_PWM_UPDATE_MODE

Name	Numeric Value	Description
`ADI_PWM_SINGLE_UPDATE`, `ADI_PWM0_SINGLE_UPDATE`	0	Single update operating mode PWM0
`ADI_PWM_DOUBLE_UPDATE`, `ADI_PWM0_DOUBLE_UPDATE`	1	Double update operating mode PWM0
`ADI_PWM1_SINGLE_UPDATE`	2	Single update operating mode PWM1
`ADI_PWM1_DOUBLE_UPDATE`	3	Double update operating mode PWM1

Interdependencies

This section describes interdependencies between the pulse-width modulation service and other system services.

Interrupt Manager Service

Since the PWM service relies on callbacks to process the events, the application must initialize the interrupt manager service before initializing the PWM service. During the adi_pwm_Init function, the interrupt manager is called upon to clear any pending interrupt signals from the PWM unit, and may also be called upon to *hook* the PWM trip and/or sync interrupt handlers, which execute the callback functions. During the adi_pwm_Terminate function, the interrupt manager is called upon to *unhook* the PWM interrupt handlers.

Deferred Callback Service

Callbacks may be deferred, which means they are executed after a higher priority thread has finished, rather than inside the interrupt that services the event. If callbacks are to be deferred, the deferred callback manager should be initialized after the interrupt manager is initialized, by calling `adi_dcb_Init()`. The handle returned from the deferred callback service is used later when installing and processing callbacks. Please refer to Chapter 5, "Deferred Callback Manager" for details.

Port Control Manager Service

The PWM signals are multiplexed. The application must call the `adi_ports_Init()` function to initialize the port control manager, prior to initializing the PWM service. Please refer to Chapter 9, "Port Control Service" for details. During the `adi_pwm_Init()` function, the port control manager is called upon to configure the port registers to select the PWM signal functionality. The application must pass a command along with a port mapping structure, which selects either the primary or secondary port mapping for each signal.

14 STDIO SERVICE

This chapter describes how to use the STDIO service found within the system services library. This service follows similar application programming interfaces (APIs) found in other system services. The chapter includes:

Introduction

The STDIO service helps to redirect STDIO streams (STDIN, STDOUT and STDERR) to different output peripherals. Each STDIO stream can be redirected to different output devices. For example, STDIN can be done through default device (JTAG) and STDOUT can be redirected to UART device. The APIs are designed so more devices can be supported in the future.

The STDIO service is designed to work in a multithreaded environment. It uses the underlying RTOS services for protecting the critical regions.

Getting Started

This section describes the overall operation of the STDIO service. Details on the API can be found in "STDIO Service API Reference" on page 14-8. All the APIs are declared in the header file adi_stdio.h.

Initialization

Before using the STDIO service, an application needs to initialize the service by calling the function adi_stdio_Init.

The function adi_stdio_Init initializes all the internal data structures of the STDIO service. To protect the data structures from being reinitialized when they are being used, this function can be called only once. Note that if the application calls adi_stdio_Terminate, then adi_stdio_Init would have to be called again before the STDIO service can be used.

Each of the STDIO devices needs memory to maintain its internal data structures. The required memory is allocated dynamically by the STDIO service from the system heap using heap allocation functions. The application should also pass the handles to other required services (device manager, DMA manager and DCB manager) through the initialization function. The initialization function returns the handle to the default STDIO device (JTAG). The default STDIO handle can be used to redirect STDIO streams back to the default device. The following code snippet shows a typical initialization procedure.

```
<services/stdio/adi_stdio.h>  /* STDIO service include */

uint32_t main(void)
{
   /*
   ** Variable to hold return code
   */
    ADI_STDIO_RESULT eResult;

   // Initialize other required service here to obtain
   // adi_dev_ManagerHandle, adi_dma_ManagerHandle

   /*
   ** Initialize STDIO Service
   */
   eResult = adi_stdio_Init(
                        /* Device Manager Handle */
                         adi_dev_ManagerHandle,
                        /* DMA Manager Handle */
                         adi_dma_ManagerHandle,
                        /* DCB Manager Handle */
                         adi_dcb_ManagerHandle,
                        /* Pointer to JTAG STDIO Device handle */
                         &hSTDIOJTAG
                            );
   /* Check if STDIO Service is initialized successfully */
   if(eResult != ADI_STDIO_RESULT_SUCCESS)
   {
    // Failed to initialize STDIO Service take appropriate
       action here
   }
   ...
   ...
}
```

Register the Required STDIO Device Types

After the STDIO service is initialized, the application should determine the STDIO device types that it is going to use and register them with the service. The following code snippet shows the registration procedure for a UART device.

```
/*
** Register the UART Device Type
*/
adi_stdio_RegisterUART();
```

Open the Required STDIO Device(s)

Once the required device type is registered with the STDIO service, it can be opened. The STDIO device must be open before redirecting any STDIO streams to it. The STDIO service takes care of configuring the STDIO device with default values (see "Commands (ADI_STDIO_COMMAND)" on page 14-23 for default values for each of the configuration parameters) to make it operational. The application uses the control interface (adi_stdio_ControlDevice API function) to change the default configuration values. The following code snippet shows the procedure to open an UART device.

```
/*
** Open the UART Device Type
*/
eResult = adi_stdio_OpenDevice(
            ADI_STDIO_DEVICE_TYPE_UART, /* UART Device Type */
            0,                          /* Physical Device number */
            &hSTDIOUART                 /* Pointer to the handle */
          );

/* Check if STDIO Device opened successfully */
if(eResult != ADI_STDIO_RESULT_SUCCESS)
{
   // Failed to open STDIO device type, take appropriate action
here
}
```

Configure STDIO Device

After the device is opened, it can be configured to change the default configuration values (see "Commands (ADI_STDIO_COMMAND)" on page 14-23 for default values for each of the configuration parameters). This step can be skipped if the default configuration values are suitable for the application's needs. The following code snippet shows an example of how to disable character echo on `STDOUT` device.

```
/*
** Disable character echo on the UART Device
*/
eResult = adi_stdio_ControlDevice(
        hSTDIOUART,                        /* UART Device Type */
        ADI_STDIO_COMMAND_ENABLE_CHAR_ECHO, /* Command ID */
        (void *) false                     /* false to disable */
       );
```

```
/* Check if command is successfully executed */
if(eResult != ADI_STDIO_RESULT_SUCCESS)
{
   // Failed to execute STDIO control command, take appropriate
action here
}
```

Redirect STDIO Stream

After the device is opened, any of the STDIO streams (`STDIN`, `STDOUT`, `STDERR`) can be redirected to it. The STDIO streams are enumerated as `ADI_STDIO_STREAM_STDIN`, `ADI_STDIO_STREAM_STDOUT`, `ADI_STDIO_STREAM_STDERR` for `STDIN`, `STDOUT`, `STDERR` respectively and `ADI_STDIO_STREAM_ALL_CONSOLE_IO` represents all three STDIO streams. See "Stream Types (ADI_STDIO_STREAM_TYPE)" on page 14-22 for more explanation on steam types. By default, all the STDIO streams are directed to default STDIO device (JTAG) by the run-time library. The application can choose a particular STDIO stream and redirect that stream to the STDIO device that it opened and the remaining streams are still directed to the default STDIO device. The following code snippet shows a redirection of all streams to the UART device that is already opened.

```
/* Redirect all STDIO streams to UART */
eResult = adi_stdio_RedirectStream (
        hSTDIOUART,                /* UART Device Handle */
        ADI_STDIO_STREAM_ALL_CONSOLE_IO /* Stream Type       */
           );

/* Check if streams are successfully redirected */
if(eResult != ADI_STDIO_RESULT_SUCCESS)
{
   // Failed to redirect STDIO streams, take appropriate action
here
}
```

If required, the streams can be redirected back to the default device. The handle to the default device is provided to the application when the STDIO service is initialized.

Disable STDIO Stream

If required, an application can disable a particular stream. When a particular stream is disabled, it is redirected to a `NULL` device. This is done internally by the STDIO service. This feature is useful when an application needs to disable all the debug print messages that are printed on the console window. In this case, the application can disable the `STDOUT` (`ADI_STDIO_STREAM_STDOUT`) stream and still continue to get error messages by keeping the `STDERR` stream enabled. The following code snippet shows disabling `STDOUT` stream.

```
/*
** Disable STDOUT stream alone
*/
eResult = adi_stdio_DisableStream (
            ADI_STDIO_STREAM_STDOUT /* Stream type to be disabled */
             );

/* Check if required stream disabled successfully */
if(eResult != ADI_STDIO_RESULT_SUCCESS)
{
    // Failed to disable STDOUT stream, take appropriate action
here
}
```

Termination

The STDIO service can be terminated using the function adi_stdio_Terminate. After the STDIO service is terminated, none of the APIs work. To use the APIs, the STDIO must be initialized again.

```
/*
** Terminate STDIO Service
*/
eResult = adi_stdio_Terminate();

/* Check if STDIO service is terminated successfully */
If(eResult != ADI_STDIO_RESULT_SUCCESS)
{
    // Failed to terminate STDIO service take appropriate action
here
}
```

STDIO Service API Reference

This section documents the STDIO service application programming interface (API).

Notation and Naming Conventions

To safeguard against conflicts with other software libraries provided by Analog Devices, or other sources, the STDIO service uses an unambiguous naming convention in which enumeration values and `typedef` statements use the `ADI_STDIO_` prefix. Functions and global variables use the lowercase `adi_stdio_` equivalent.

Each function within the STDIO service API returns an error code of the type `ADI_STDIO_RESULT`. Like the other system services, a return value of zero (`0=ADI_STDIO_RESULT_SUCCESS`) indicates that no error has occurred during the function call. Any nonzero value indicates the specific type of error that has occurred. The error codes for the STDIO service are unique from those of the other system services and they are defined in the `adi_stdio.h` include file. This allows the user to determine the cause of the error by looking up the error code in that file.

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

Name – Name and purpose of the function

Description – Function specification

Conditions – Special conditions for the function

Prototype – Required header file and functional prototype

Arguments – Description of function arguments

Return Value – Description of function return values

adi_stdio_Init

Description

This function is used to initialize the STDIO service. The STDIO service should be initialized before calling any of the other APIs.

Each STDIO device needs memory to store its internal data structures. The required memory is allocated dynamically by the STDIO service. The application should pass the handle to the memory pool which is used for dynamic allocations. If `NULL` is passed as the handle to the memory pool, then memory is allocated from the system heap using heap allocation functions.

Conditions

The STDIO service can be reinitialized only after it is terminated. The function adi_stdio_Terminate should be called to terminate the STDIO service.

None of the other STDIO APIs work until the service is initialized.

Prototype

```
ADI_STDIO_RESULT  adi_stdio_Init (
     ADI_DEV_MANAGER_HANDLE        const hDeviceMgr,
     ADI_DMA_MANAGER_HANDLE        const hDMAMgr,
     ADI_DCB_HANDLE                const hDcbMgr,
     ADI_STDIO_DEVICE_HANDLE            *phDefaultDevice
);
```

Arguments

`hDeviceMgr`	Handle to the device manager
`hDMAMgr`	Handle to the DMA manager
`hDcbMgr`	Handle to deferred callback manager
`phDefaultDevice`	Pointer to a location where the handle to the default STDIO device is written. It is not required to open the default STDIO device—it is opened by the run-time library. Upon successful initialization, the STDIO service writes the handle to the default device into the given pointer to the default device handle.

Return Value

`ADI_STDIO_RESULT_SUCCESS`	Successfully initialized the STDIO service
`ADI_STDIO_RESULT_INVALID_POINTER`	The given pointer to instance memory or the pointer to the default STDIO device handle is invalid
`ADI_STDIO_RESULT_INVALID_HANDLE`	One of the given handles is invalid
`ADI_STDIO_RESULT_SERVICE_ALREADY_INITIALIZED`	The STDIO service is already initialized. It can only be initialized after terminating the service using the `adi_stdio_Terminate` API.
`ADI_STDIO_RESULT_MUTEX_FAILURE`	Failed to create Mutex which is required to protect the internal data structures

adi_stdio_RegisterUART

Description

This function is used to register the UART device with the STDIO service.

Conditions

This function can be called any number of times, but should be called at least once before opening the UART device type.

Once the device is registered with the STDIO service, it is not required to register again, even if the STDIO service is terminated and reinitialized again.

Prototype

```
void adi_stdio_RegisterUART (
    void
);
```

Arguments

None

Return Value

None

adi_stdio_OpenDevice

Description

This function is used to open the STDIO device which can be later used to redirect one or more STDIO streams.

Conditions

Before opening the device type, register it with the STDIO service. See "Device Type (ADI_STDIO_DEVICE_TYPE)" on page 14-23 to find registration functions that correspond to each device type. It is not required to register the device again even after the STDIO service is terminated and reinitialized.

The device type should be one of the device types listed in the enumeration Device Type (ADI_STDIO_DEVICE_TYPE) and should not be the reserved device types `ADI_STDIO_DEVICE_TYPE_RESERVED1` or `ADI_STDIO_DEVICE_TYPE_RESERVED2`. These reserved device types are used internally by the STDIO service.

Prototype

```
ADI_STDIO_RESULT  adi_stdio_OpenDevice (
     ADI_STDIO_DEVICE_TYPE        eDeviceType,
     uint32_t                     nPhysicalDevNum,
     ADI_STDIO_DEVICE_HANDLE     *phStdioDevice
);
```

Arguments

`eDeviceType`	The type of the device to be opened. Should be one of the values listed in the enumeration Device Type (ADI_STDIO_DEVICE_TYPE).
`nPhysicalDevNum`	Physical device instance number to be opened. There can be more than one device of the same device type on a given board. This parameter is used to identify the device to be opened among several instances of the same device. As an example, there could be more than one UART on a given board. This parameter identifies the UART number to be opened for the STDIO operations. Users should refer to the hardware manual of their board to determine the device that is used for the console I/O.
`phStdioDevice`	Pointer to the handle of the device to be opened. Upon success, the STDIO service writes the handle to the opened STDIO device. Note that it is not the physical device driver handle but it is the STDIO device handle. The actual physical device handle is stored within the STDIO service. If required, the physical device handle can be obtained by the application using the control command ADI_STDIO_COMMAND_GET_DEVICE_HANDLE (0x120002).

Return Value

`ADI_STDIO_RESULT_SUCCESS`	Successfully opened given STDIO device
`ADI_STDIO_RESULT_SERVICE_NOT_INITIALIZED`	STDIO service is not initialized. STDIO service should be initialized before using this API.
`ADI_STDIO_RESULT_INVALID_POINTER`	Given pointer to the STDIO device handle is invalid
`ADI_STDIO_RESULT_NO_STDIO_DEVICES`	Reached the limit of maximum number of STDIO devices that can be opened
`ADI_STDIO_RESULT_DEVICE_FAILED`	Failed to open the physical device
`ADI_STDIO_RESULT_INVALID_DEVICE_TYPE`	Given device type is not valid
`ADI_STDIO_RESULT_DEVICE_NOT_REGISTERED`	Given device type is not registered. Each device type should be registered with the STDIO service once before opening the device of the given device type.

adi_stdio_Redirect

Description

This function is used to redirect one or more STDIO streams to the given STDIO device. To redirect all the streams at once, the application should pass the stream type as `ADI_STDIO_STREAM_ALL_CONSOLE_IO`.

Conditions

The device to which the stream is redirected should be opened before calling this function.

Prototype

```
ADI_STDIO_RESULT adi_stdio_RedirectStream (
    ADI_STDIO_DEVICE_HANDLE   hStdioDevice,
    ADI_STDIO_STREAM_TYPE     eStreamType
);
```

Arguments

`hStdioDevice`	Handle to the STDIO device to which the given stream is redirected to
`eStreamType`	Stream type to be redirected. Stream type should be one of the values listed by the enumeration Stream Types (ADI_STDIO_STREAM_TYPE).

Return Value

`ADI_STDIO_RESULT_SUCCESS`	Successfully redirected the given stream type to the given device
`ADI_STDIO_RESULT_SERVICE_NOT_INITIALIZED`	STDIO service is not initialized
`ADI_STDIO_RESULT_INVALID_HANDLE`	Given handle to the STDIO device is invalid
`ADI_STDIO_RESULT_STREAM_NOT_SUPPORTED`	Given stream type is not supported by the given device
`ADI_STDIO_RESULT_REDIRECT_FAILED`	Failed to redirect the given stream
`ADI_STDIO_RESULT_DEVICE_FAILED`	STDIO physical device initialization failed
`ADI_STDIO_RESULT_SEMAPHORE_FAILURE`	Failed to create semaphore that is required internally for the STDIO service
`ADI_STDIO_RESULT_DEVTAB_REGISTER_FAILED`	Failed to register the given device into the device table of the run-time library

adi_stdio_DisableStream

Description

This function is used to disable a given STDIO stream. When a stream is disabled, all the activity of that stream is redirected to a `NULL` device and will not be seen on the console.

Conditions

When a particular stream is disabled, other streams are still active and not affected.

Prototype

```
ADI_STDIO_RESULT adi_stdio_DisableStream (
  ADI_STDIO_STREAM_TYPE    eStreamType
);
```

Arguments

`eStreamType`	Stream type to be disabled

Return Value

`ADI_STDIO_RESULT_SUCCESS`	Successfully disabled the given stream
`ADI_STDIO_RESULT_SERVICE_NOT_INITIALIZED`	STDIO service is not initialized
`ADI_STDIO_RESULT_SEMAPHORE_FAILURE`	Failed to create semaphore that is required internally for the STDIO service
`ADI_STDIO_RESULT_DEVTAB_REGISTER_FAILED`	Failed to register the given device into the device table of the run-time library

adi_stdio_ControlDevice

Description

This function is primarily used to change configuration parameters of the given device. The function is also used to get some parameters from the STDIO service, for example, to get the handle of the physical device.

Conditions

None

Prototype

```
ADI_STDIO_RESULT adi_stdio_ControlDevice (
    ADI_STDIO_DEVICE_HANDLE            hStdioDevice,
    uint32_t                           nCommandID,
    void                        *const pValue
);
```

Arguments

hStdioDevice	Handle to the STDIO device
nCommandID	Command ID to be executed. See Commands (ADI_STDIO_COMMAND) for a list of available commands.
pValue	Argument required for executing the command. Depending upon the command, different types of arguments are required. See Commands (ADI_STDIO_COMMAND) to learn about the command-specific arguments.

Return Value

`ADI_STDIO_RESULT_SUCCESS`	Successfully executed the given command
`ADI_STDIO_RESULT_INVALID_HANDLE`	Given handle to the STDIO device is invalid
`ADI_STDIO_RESULT_DEVICE_FAILED`	Failure detected from physical device
`ADI_STDIO_RESULT_COMMAND_NOT_SUPPORTED`	Given command is not supported by the given device

adi_stdio_CloseDevice

Description

This function is used to close the STDIO device that was opened by the application.

Conditions

When a device is closed, all the streams that were directed to the given device are redirected to the default STDIO device.

Prototype

```
ADI_STDIO_RESULT adi_stdio_CloseDevice (
     ADI_STDIO_DEVICE_HANDLE   const hStdioDevice
);
```

Arguments

`hStdioDevice`	Handle to the STDIO device that needs to be closed

Return Value

`ADI_STDIO_RESULT_SUCCESS`	Successfully closed the given STDIO device
`ADI_STDIO_RESULT_INVALID_HANDLE`	The STDIO service is not initialized. The STDIO service should be initialized before using this API.
`ADI_STDIO_RESULT_DEVICE_FAILED`	Failed to close the physical device corresponding to the given STDIO device. The physical device is opened by the STDIO service when the application calls the `adi_stdio_OpenDevice` function.

adi_stdio_Terminate

Description

This function is used to terminate the STDIO service. The application can use this function to terminate the STDIO service when it does not need the STDIO service any longer.

Conditions

This function does not close any of the STDIO devices that are opened by the application. If the intent is to reinitialize the STDIO service after terminating, it is recommended to close all the devices before terminating the STDIO service.

Before terminating, the STDIO service redirects all the STDIO streams back to the default device.

None of the STDIO APIs work once the STDIO service is terminated. It needs to be reinitialized by calling the API function `adi_stdio_Init`.

Prototype

```
ADI_STDIO_RESULT adi_stdio_Terminate(void)
```

Arguments

None

Return Value

`ADI_STDIO_RESULT_SUCCESS`	Successfully terminated the STDIO service
`ADI_STDIO_RESULT_SERVICE_NOT_INITIALIZED`	Trying to terminate the STDIO service when it is not initialized

STDIO Service API Structures, Definitions, and Enumerations

This section lists all the structure definitions, constant definitions, and enumerations that are used in the APIs. All of these can be found in the file `adi_stdio.h`.

Stream Types (ADI_STDIO_STREAM_TYPE)

Table 14-1 is an enumerated list of supported STDIO stream types. These values are used to specify the stream that needs to be redirected to a particular device or to disable a stream.

Table 14-1. Stream Types and Description

Stream Type Name	Value	Description
`ADI_STDIO_STREAM_STDIN`	0	Console input stream (STDIN)
`ADI_STDIO_STREAM_STDOUT`	1	Console output stream (STDOUT)
`ADI_STDIO_STREAM_STDERR`	2	Console error stream (STDERR)
`ADI_STDIO_STREAM_ALL_CONSOLE_IO`	3	All the above console I/O streams (STDIN, STDOUT, STDERR)

Device Type (ADI_STDIO_DEVICE_TYPE)

Table 14-2 is an enumerated list of supported STDIO device types. The device type is used to open the device for STDIO.

Table 14-2. Device Types and Description

Device Type	Value	Description
ADI_STDIO_DEVICE_TYPE_RESERVED1	0	Device type is reserved and used internally by the STDIO service
ADI_STDIO_DEVICE_TYPE_RESERVED2	1	Device type is reserved and used internally by the STDIO service
ADI_STDIO_DEVICE_TYPE_UART	2	UART device. Before opening this device type, it needs to be registered by calling the function adi_stdio_RegisterUART.
ADI_STDIO_DEVICE_TYPE_MAX	3	Maximum number of supported device types

Commands (ADI_STDIO_COMMAND)

There are several commands available to set or get a configuration parameter.

ADI_STDIO_COMMAND_ENABLE_UNIX_MODE (0x120000)

This command is used to enable or disable the Unix mode end-of-line (EOL) character. In Unix mode, LF (line feed) is used as EOL and in DOS mode, CR + LF (carriage return and line feed) is used as EOL. By default, Unix mode is set.

Command Specific Value

'True' to enable Unix mode and 'false' to disable it (set to DOS mode).

ADI_STDIO_COMMAND_ENABLE_CHAR_ECHO (0x120001)

This command is used to allow the service to print the characters that are typed on the `stdin` stream onto the `stdout` device. By default, character echo is enabled.

Command Specific Value

'True' to enable character echo and 'false' to disable it.

ADI_STDIO_COMMAND_GET_DEVICE_HANDLE (0x120002)

This command is used to get the physical device handle. When the STDIO device is opened using the API `adi_stdio_OpenDevice`, the corresponding physical device is opened by the STDIO service and its handle is stored internally. Applications are generally not required to get the handle to the physical device in order to use the service. This command can be used in unusual cases where there is a need to set some physical device specific configurations which are not accessible through the STDIO service API. It is not recommended to set any physical device configuration values directly from the application. This may lead to unintended behavior.

Command Specific Value

Pointer to `ADI_DEV_DEVICE_HANDLE` (defined in `adi_dev.h` file) where the physical device handle is written by the STDIO service

ADI_STDIO_COMMAND_SET_UART_PARITY_TYPE (0x120004)

This command is used to disable parity check or set odd or even parity check for UART communication. By default, parity check is disabled by the STDIO service.

Command Specific Value

One of the values from the enumerated list Parity Types (ADI_STDIO_PARITY_TYPE). When `ADI_STDIO_PARITY_TYPE_ODD` or `ADI_STDIO_PARITY_TYPE_EVEN` is chosen, then the STDIO service automatically enables parity check.

ADI_STDIO_COMMAND_SET_UART_WORD_LENGTH (0x120005)

This command is used to set the UART word length. By default, the word length is set to 8 bits.

Command Specific Value

```
uint8_t  nWordLength;
```

`nWordLength` can be set to 5, 6, 7 or 8 bits.

ADI_STDIO_COMMAND_SET_UART_NUM_STOP_BITS (0x120006)

This command is used to set the number of UART stop bits. By default, the number of UART stop bits is set to 1.

Command Specific Value

```
uint8_t  nStopBits;
```

`nStopBits` can be set to 1 or 2. A value of 1sets one stop bit, a value of 2 sets two stop bits when the UART word length is set to a non-5-bit value and 1½ stop bits for a 5-bit word length.

ADI_STDIO_COMMAND_SET_UART_AUTO_BAUD_CHAR (0x120007)

This command is used to set the auto baud character. The auto baud character is used when auto baud detection is enabled. This command specifies the character the driver should expect for autobaud detection.

Command Specific Value

```
uint8_t  cAutoBaud;
```

`cAutoBaud` can be any ASCII character.

ADI_STDIO_COMMAND_ENABLE_AUTO_BAUD_CHAR (0x120008)

This command is used to enable the auto baud detection. The command enables the driver to automatically sense the baud rate of the serial line and configure the UART accordingly. This command should be used in conjunction with the `ADI_STDIO_COMMAND_SET_UART_AUTO_BAUD_CHAR`. The application should first set the auto baud character before enabling auto baud detection.

Command Specific Value

None. There are no command specific arguments required for this command.

ADI_STDIO_COMMAND_SET_UART_BAUD_RATE (0x120009)

This command is used to configure the UART for a given baud rate. By default, the baud rate is set to 57600 Hz.

Command Specific Value

```
uint16_t nBaudRate;
```

`nBaudRate` is the baud rate in Hz.

Parity Types (ADI_STDIO_PARITY_TYPE)

Table 14-3 is an enumerated list of possible parity types

Table 14-3. Parity Types and Description

Parity Type	Value	Description
ADI_STDIO_PARITY_TYPE_NONE	0x0	No parity check
ADI_STDIO_PARITY_TYPE_ODD	0x1	Odd parity
ADI_STDIO_PARITY_TYPE_EVEN	0x2	Even parity

Result Codes (ADI_STDIO_RESULT)

Table 14-4 is an enumerated list of possible result codes from the STDIO service.

Table 14-4. Result Codes and Description

Result Code	Value	Description
ADI_STDIO_RESULT_SUCCESS	0x000000	Generic success
ADI_STDIO_RESULT_FAILED	0x000001	Generic failure

Table 14-4. Result Codes and Description (Cont'd)

Result Code	Value	Description
`ADI_STDIO_RESULT_SERVICE_NOT_INITIALIZED`	0x120000	STDIO service is not initialized
`ADI_STDIO_RESULT_SERVICE_ALREADY_INITIALIZED`	0x120001	STDIO service already initialized
`ADI_STDIO_RESULT_INSUFFICIENT_MEMORY`	0x120002	Insufficient memory passed to support requested number of instances
`ADI_STDIO_RESULT_INVALID_HANDLE`	0x120003	Given handle is invalid
`ADI_STDIO_RESULT_INVALID_DEVICE_TYPE`	0x120004	Given device type is invalid
`ADI_STDIO_RESULT_DEVICE_NOT_REGISTERED`	0x120005	Given device type is not registered
`ADI_STDIO_RESULT_REDIRECT_FAILED`	0x120006	Failed to redirect the given stream to the given device
`ADI_STDIO_RESULT_SEMAPHORE_FAILURE`	0x120007	Failed to create the semaphore
`ADI_STDIO_RESULT_MUTEX_FAILURE`	0x120008	Failed to create the Mutex
`ADI_STDIO_RESULT_DEVTAB_REGISTER_FAILED`	0x120009	Failed to register the STDIO device with LIBIO
`ADI_STDIO_RESULT_COMMAND_NOT_SUPPORTED`	0x12000A	Given command is not supported by the given device
`ADI_STDIO_RESULT_DEVICE_FAILED`	0x12000B	Physical device driver failed
`ADI_STDIO_RESULT_INVALID_POINTER`	0x12000C	Given pointer is invalid or pointing to `NULL`

Table 14-4. Result Codes and Description (Cont'd)

Result Code	Value	Description
`ADI_STDIO_RESULT_INVALID_FUNCTION_POINTER`	`0x12000D`	Given function pointer is pointing to `NULL`
`ADI_STDIO_RESULT_NO_STDIO_DEVICES`	`0x12000E`	Reached the limit of maximum number of STDIO devices that can be opened in the system
`ADI_STDIO_RESULT_STREAM_NOT_SUPPORTED`	`0x12000F`	Given stream is not supported by the given device

I INDEX