TMS320C6000 Chip Support Library API Reference Guide

Literature Number SPRU401
March 2000

Printed on Recycled Paper

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgement, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

CERTAIN APPLICATIONS USING SEMICONDUCTOR PRODUCTS MAY INVOLVE POTENTIAL RISKS OF DEATH, PERSONAL INJURY, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE ("CRITICAL APPLICATIONS"). TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF TI PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE FULLY AT THE CUSTOMER'S RISK.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.

Preface

Read This First

About This Manual

Welcome to the TMS320C6000 Chip Support Library, or CSL for short. The CSL is a set of application programming interfaces (APIs) used to configure and control all on-chip peripherals. It is intended to make it easier for developers by eliminating much of the tedious grunt-work usually needed to get algorithms up and running in a real system. Some of the advantages offered by the CSL include: peripheral ease of use, a level of compatibility between devices, shortened development time, portability, standardization, and hardware abstraction. A version of the CSL is available for all TMS320C6000 devices.

This document contains a reference for the CSL APIs and is organized as follows:

- ❏ Overview – a high level overview of the CSL
- ❏ CSL API Module Descriptions – a description of the individual CSL API modules
- ❏ CSL API Functions – a brief description of all CSL API functions in table format
- ❏ CSL API Reference – an alphabetical listing of all CSL API identifiers
- ❏ HAL Reference – a low-level reference of the hardware abstraction layer listing all macros and constants for manipulating the peripheral registers

How to Use This Manual

The information in this document describes the contents of the TMS320C6000 chip support library in several different ways.

- ❏ Chapter 1 provides an overview of the CSL and its 2-layer architecture consisting of the service layer and the hardware abstraction layer (HAL).
- ❏ Chapter 2 provides an introduction to the service-layer API modules and gives a description of each in alphabetical order together with tables showing the various functions, macros, constants, etc., and a section and page

reference for more detailed information about each. This chapter is intended for those who want to understand the internal workings of the service layer APIs.

- Chapter 3 provides a quick overview of all CSL API functions in table format for easy reference. The information shown for each function includes the syntax, a brief description, and a page reference for obtaining more detailed information.
- Chapter 4 provides an alphabetical listing of the CSL service-layer API functions, enumerations, type definitions, macros, structures, constants, and global variables. This chapter uses examples to show how these elements are used.
- Chapter 5 contains an alphabetical reference of the chip support library hardware abstraction layer (CSL HAL).

Notational Conventions

This document uses the following conventions:

- Program listings, program examples, and interactive displays are shown in a `special typeface`.
- In syntax descriptions, the function or macro appears in a **bold typeface** and the parameters appear in plainface within parentheses. Portions of a syntax that are in **bold** should be entered as shown; portions of a syntax that are within parentheses describe the type of information that should be entered.
- Macro names are written in uppercase text; function names are written in lowercase.
- TMS320C6000 devices are referred to throughout this reference guide as ’C6201, ’C6202, etc.

Related Documentation From Texas Instruments

The following books describe the TMS320C6x devices and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477–8924. When ordering, please identify the book by its title and literature number. Many of these documents can be found on the Internet at http://www.ti.com.

TMS320C62x/C67x Technical Brief (literature number SPRU197) gives an introduction to the ’C62x/C67x digital signal processors, development tools, and third-party support.

TMS320C6000 CPU and Instruction Set Reference Guide (literature number SPRU189) describes the 'C6000 CPU architecture, instruction set, pipeline, and interrupts for these digital signal processors.

TMS320C6201/C6701 Peripherals Reference Guide (literature number SPRU190) describes common peripherals available on the TMS320C6201/6701 digital signal processors. This book includes information on the internal data and program memories, the external memory interface (EMIF), the host port interface (HPI), multichannel buffered serial ports (McBSPs), direct memory access (DMA), enhanced DMA (EDMA), expansion bus, clocking and phase-locked loop (PLL), and the power-down modes.

TMS320C6000 Programmer's Guide (literature number SPRU198) describes ways to optimize C and assembly code for the TMS320C6000 DSPs and includes application program examples.

TMS320C6000 Assembly Language Tools User's Guide (literature number SPRU186) describes the assembly language tools (assembler, linker, and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the 'C6000 generation of devices.

TMS320C6000 Optimizing C Compiler User's Guide (literature number SPRU187) describes the 'C6000 C compiler and the assembly optimizer. This C compiler accepts ANSI standard C source code and produces assembly language source code for the 'C6000 generation of devices. The assembly optimizer helps you optimize your assembly code.

TMS320C62x DSP Library (literature number SPRU402) describes the 32 high-level, C-callable, optimized DSP functions for general signal processing, math, and vector operations.

TMS320C62x Image/Video Processing Library (literature number SPRU400) describes the optimized image/video processing functions including many C-callable, assembly-optimized, general-purpose image/video processing routines.

Contents

Figures

Tables

Chapter 1

Introduction

This chapter provides an overview of the chip support library (CSL) and its two-layer architecture.

Topic **Page**

1.1 CSL Overview

The CSL is written primarily in C with some assembly language where needed. The library is made up of discrete modules that are built and archived into a library file. Each module represents an individual API and is referred to simply as an API module. The module granularity is architected such that each peripheral is covered by a single API module. Hence, there is a DMA API module for the DMA peripheral, a MCBSP API module for the McBSP peripheral, and so on.

Current List of CSL API Modules:

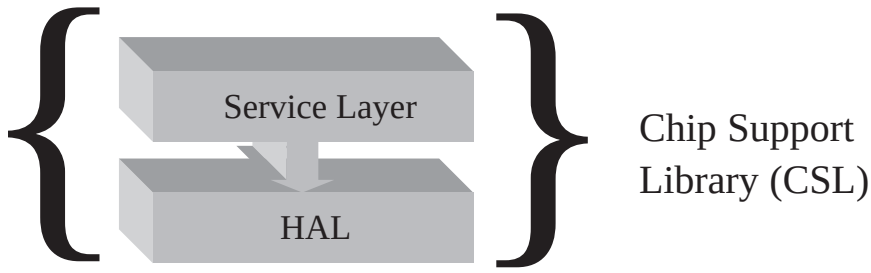
- ❑ CACHE cache module
- ❑ CSL top-level module
- ❑ DAT device independent data copy/fill module
- ❑ CHIP chip specific module
- ❑ DMA direct memory access module
- ❑ EDMA enhanced direct memory access module
- ❑ EMIF external memory interface module
- ❑ HPI host port interface module
- ❑ IRQ interrupt controller module
- ❑ MCBSP multichannel buffered serial port module
- ❑ PWR power down module
- ❑ STDINC standard include module
- ❑ TIMER timer module

Although each API module is unique, there exists some interdependency between the modules. For example, the DMA module depends on the IRQ module because of DMA interrupts. This comes into play when linking code because if you use the DMA module, the IRQ module automatically gets linked also.

The CSL is architected using a two-layer approach: the top layer is the *service layer* and the bottom layer is the *hardware abstraction layer* or *HAL*. This is illustrated in Figure 1–1, *CSL Layers*.

Note: The CSL depends on DSP/BIOS for its hardware interrupt dispatcher. Hence, you must create a DSP/BIOS application with your Code Composer Studio project to use the CSL.

Figure 1–1. CSL Layers



1.2 HAL Overview

The hardware abstraction layer, or HAL, is a set of constants and macros that fully describes the peripheral registers by way of symbols. It is capable of hiding subtle differences between devices such as bit-fields changing size or position within a register. In addition, the HAL can substitute a NULL register in the case where a register is supported on one device but not another. The whole purpose of the HAL is to provide the service layer a symbolic interface into the hardware. It is not intended as a user interface or API. This is explained further in section 1.3 , *Service Layer Overview*.

What the CSL HAL Offers:

- ❏ Symbol definitions for every peripheral register
  - ■ HPER_REG_ADDR
  - ■ HPER_REG
- ❏ Symbol definitions for every bit-field of every peripheral register
  - ■ HPER_REG_FIELD_MASK
  - ■ HPER_REG_FIELD_SHIFT
- ❏ Macro definitions to get and set any field of any peripheral register
  - ■ HPER_REG_FIELD_GET()
  - ■ HPER_REG_FIELD_SET()
- ❏ Macro definitions to get and set any peripheral register
  - ■ HPER_REG_GET()
  - ■ HPER_REG_SET()
- ❏ Macro definitions to configure any peripheral register based on field values
  - ■ HPER_REG_CFG()

HPER – peripheral module name, ex. DMA

REG – peripheral register name, ex. PRICTL

FIELD – peripheral register field name, ex. ESIZE

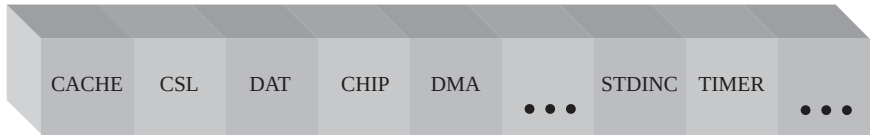
For a complete reference of the HAL, see Chapter 5, *HAL Reference.*

1.3 Service Layer Overview

The service layer is where the actual APIs are defined and is the layer the user interfaces to. It is possible for the user to interface directly into the HAL but this is not advisable because it could have undesired side effects on the operation of the service layer APIs. For example, you would not want to use the HAL to directly write to a DMA register while that register is in use by the DMA service layer API. If however, the user decides not to use the service layer at all, the HAL is available and can still save the user time and effort.

Figure 1–2 illustrates the individual API modules within the service layer. This architecture allows for future expansion of the CSL because new API modules can be added to the service layer as new peripheral devices emerge.

Figure 1–2. API Modules within Service Layer



It is important to note that not all devices support all API modules. This depends on if the device actually has the peripheral that an API covers. For example, the EDMA API module is not supported on a 'C6201 because this device does not have an EDMA peripheral. Other modules, however, are supported on all devices such as the IRQ module. As will be shown later in the *API Reference* section, each module has a compile time constant symbol defined that denotes if the module is supported or not for a given device. For example, the symbol `EDMA_SUPPORT` has a value of 1 if the current device supports it, and a value of 0 otherwise. You can use these support symbols in their application code to make decisions if so desired.

Note:

When using the CSL, it is up to the user to define a project-wide symbol from a predetermined set to identify which device is being used. This device identification symbol is then used in the CSL header files to conditionally define the support symbols. See the CHIP API reference (section 4.5, CHIP, on page 4-19) for more information.

Chapter 2

CSL API Module Descriptions

This chapter describes the purpose of the individual CSL API modules within the chip support library (CSL) and explains how they work. This information is intended for those who want to understand the internal workings of the service layer APIs.

Topic **Page**

2.1 CSL API Module Introduction

There are certain methods used across different modules that are worth mentioning at the global level.

OPEN and CLOSE Functions

Peripherals that have multiple channels, ports, etc. must be managed as resources in a shared environment. The CSL APIs allow for this by way of *Open* and *Close* API functions. For example, a piece of application code can open a DMA channel for exclusive use. This precludes any other part of the program from opening the same DMA channel. If the resource is no longer needed, it can be freed up by closing it. The methodology used is the *handle* concept. You obtain a peripheral handle by calling the *Open* function for that peripheral. This handle is then used in all subsequent API calls for that peripheral. This of course only applies to peripheral devices that have multiple channels or ports such as the DMA, McBSP, and timer. Other peripherals such as the EMIF need no such resource management and do not have handle based API calls.

MK Macros

Another method that many of the API modules share is the idea of the *MK* macro that stands for *make*. Inevitably when configuring peripherals, you will have to set some registers to some values. It can be painstaking work to calculate bit-field values and then shift-merge them all together to form a register value. To make this easier, the API modules define *MK* macros. These macros take individual right-justified field values as arguments and form the merged value. In addition, symbolic constants are provided that may be used for the field values. To illustrate, consider a hypothetical register named REG that is part of a peripheral named PER. This register has 5 fields, F0, F1, F2, F3, and F4. The *MK* macro will look like this:

```
PER_MK_REG(f0,f1,f2,f3,f4)
```

Additionally, there will be field value constants similar to the following:

```
PER_REG_F0_VAL0
PER_REG_F0_VAL1
PER_REG_F1_VAL0
...
PER_REG_F4_VAL3
```

This macro may be used several ways. You can use it without the symbolic constants like this:

```
val = PER_MK_REG(0,4,1,3,8);
```

or, to make your code more readable and self-documenting, you can use the symbolic constants for the field values like this:

```
val = PER_MK_REG(
  PER_REG_F0_VAL0,
  PER_REG_F1_VAL1,
  PER_REG_F2_VAL1,
  PER_REG_F3_VAL0,
  PER_REG_F4_VAL3
);
```

You could just as well use variables for the field values. If you used all constants for the field values, the whole macro resolves down to a single constant number at compile time.

A couple rules apply to these macros:

- ❏ Only writeable register fields are arguments, no read-only fields
- ❏ The register field arguments are specified least-significant first
- ❏ If a field is not implemented for a particular device, use the PER_REG_FIELD_NA value

Initializing the Registers of a Peripheral

Related to the *MK* macros is a pair of functions and a structure definition that many of the API modules implement. The two functions are *PER_ConfigA()* and *PER_ConfigB()* and the structure is *PER_CONFIG* where *PER* is the peripheral module name such as DMA. These functions along with the structure give you two ways of initializing the registers of a peripheral. Using method A, you initialize a configuration structure with the appropriate register values. Then you pass the address of this structure to the *ConfigA* function that in turn initializes the peripheral registers with the values in the structure. The other method (method B) does not use the configuration structure; instead, you pass the register values as individual arguments to the *ConfigB* function. The two methods may be used interchangeably. Using either of the two methods, you still have to come up with the register values. This is where the *MK* (make) macros help: you can use the macros inside of the structure initializer or you can use them in place of the arguments when calling the *ConfigB* function.

Consider this hypothetical example:

```
PER_CONFIG MyConfig = {
  reg0 val,
  reg1 val,
  ...
};
...
PER_ConfigA(&MyConfig);
```

Or for method B:

```
PER_ConfigB(reg0 val, reg1 val, ...);
```

Function Inlining

Another topic that is global to the API modules is function inlining. It turns out that a high percentage of the API functions are very short, setting a single register bit for example. For these small functions, it does not make sense to always incur the overhead of a C function call. So, the API declares them as *static inline* when the user enables inlining. You can actually reduce code size by

enabling inlining when the functions are very small. This is because the size of the function is smaller than the state saving code required to call the function if it were not inlined. The API reference does not state which functions are inlined and which are not: this allows for future changes. You should see an increase in CSL code performance when you enable function inlining.

2.2 CACHE

The CACHE module offers a small set of API functions for managing data and program cache. The CACHE module includes:

- Constant
  - CACHE_SUPPORT
- Functions
  - CACHE_Clean
  - CACHE_EnableCaching
  - CACHE_Flush
  - CACHE_GetL2SramSize
  - CACHE_Invalidate
  - CACHE_Reset
  - CACHE_SetL2Mode
  - CACHE_SetPccMode

CACHE Architectures

Currently, there are two cache architectures used on ’C6x devices. The first type, which is present on the ’6201device, provides program cache by disabling on-chip program RAM and turning it into cache. The second type, which is present on the ’6211 device, is the newer two-level (L2) cache architecture.

The CACHE module has APIs that are specific for the L2 cache and specific for the older program cache architecture. However, the API functions are callable on both types of platforms to make application code portable. On devices without L2, the L2-specific cache API calls do nothing but return immediately.

For additional information about the CACHE module, refer to Table 3–1 on page 3-2 and Section 4.2 on page 4-3.

2.3 CSL

The CSL module is the top-level API module whose primary purpose is to initialize the library. Only one function is exported:

- `CSL_Init()`

The *CSL_Init()* function must be called once at the beginning of your program before calling any other CSL API functions.

For additional information about the CSL module, refer to Table 3–2 on page 3-2 and Section 4.3 on page 4-10.

2.4 DAT

The DAT module, which stands for data, is used to move data around by means of DMA/EDMA hardware. This module serves as a level of abstraction such that it works the same for devices that have the DMA peripheral and devices that have the EDMA peripheral. So if you write application code that uses the DAT module, it is compatible across all current devices without worrying about what type of DMA controller it has. The DAT module includes:

- Constants
  - `DAT_SUPPORT`
- Functions
  - `DAT_Close`
  - `DAT_Copy`
  - `DAT_Fill`
  - `DAT_Open`
  - `DAT_Wait`
  - `DAT_Copy2D`

DAT Routines

The DAT module is intentionally kept simple. There are routines to copy data from one location to another and to also fill a region of memory. These operations occur in the background on dedicated DMA hardware independent of the CPU. Because of this asynchronous nature, there is API support that enables waiting until a given copy/fill operation completes. It works like this: call one of the copy/fill functions and get an ID number as a return value. Then use this ID number later on to wait for the operation to complete. This allows the operation to be submitted and performed in the background while the CPU performs other tasks in the foreground. Then as needed, the CPU may block on completion of the operation before moving on.

DMA/EDMA Management

Since the DAT module uses the DMA/EDMA peripheral, it must do so in a managed way. In other words, it must not use a DMA channel that is already allocated by the application. To ensure this doesn't happen, the DAT module must be opened before use, this is accomplished using the `DAT_Open()` API function. Opening the DAT module allocates a DMA channel for exclusive use. If the module is no longer needed, the DMA resource may be freed up by closing the DAT module, i.e. `DAT_Close()`.

Note:

For devices that have EDMA, the DAT module uses the quick DMA feature. This means that the module doesn't have to internally allocate a DMA channel. However, you are still required to open the DAT module before use.

Devices With DMA

On devices such as the '6201 that have the DMA peripheral, only one request may be active at once since only one DMA channel is used. If you submit two requests back to back, the first one will be programmed into the DMA hardware immediately but the second one will have to wait until the first completes. The APIs will block (spin) if called while a request is still busy by polling the transfer complete interrupt flag. The completion interrupt is not actually enabled to eliminate the overhead of taking an interrupt but the interrupt flag is still active.

Devices With EDMA

On devices with EDMA, it is possible to have multiple requests pending because of hardware request queues. Each call into the `DAT_Copy()` or `DAT_Fill()` functions return a unique transfer ID number. This ID number is then used by the user to wait for transfer completion. The ID number allows the library to distinguish between multiple pending transfers. As with the DMA, transfer completion is determined by monitoring EDMA transfer complete codes (interrupt flags).

For additional information about the DAT module, refer to Table 3–3 on page 3-3 and Section 4.4 on page 4-11.

2.5 CHIP

The CHIP module is where chip-specific and chip-related code resides. This module has the potential to grow in the future as more devices are placed on the market. Currently, CHIP has some API functions for obtaining device endianess, memory map mode if applicable, and CPU and REV IDs. The CHIP module includes:

- Constants
  - `CHIP_6XXX`
  - `CHIP_SUPPORT`
- Functions
  - `CHIP_GetCpuId`
  - `CHIP_GetEndian`
  - `CHIP_GetMapMode`
  - `CHIP_GetRevId`

For additional information about the CHIP module, refer to Table 3–4 on page 3-3 and Section 4.5 on page 4-19.

2.6 DMA

Currently, there are two DMA architectures used on ’C6x devices: DMA and EDMA (enhanced DMA). Devices such as the ’6201 have the DMA peripheral, whereas the ’6211 has the EDMA peripheral. The two architectures are different enough to warrant a separate API module for each. The DMA module includes:

- Constants
  - DMA_CHA_CNT
  - DMA_SUPPORT
- Functions
  - DMA_AllocGlobalReg
  - DMA_AutoStart
  - DMA_Close
  - DMA_ConfigA
  - DMA_ConfigB
  - DMA_FreeGlobalReg
  - DMA_GetEventId
  - DMA_GetGlobalReg
  - DMA_GetStatus
  - DMA_Open
  - DMA_Pause
  - DMA_Reset
  - DMA_SetAuxCtl
  - DMA_SetGlobalReg
  - DMA_Start
  - DMA_Stop
  - DMA_Wait
- Macros
  - DMA_CLEAR_CONDITION
  - DMA_GET_CONDITION
- Macros (cont.)

- DMA_MK_AUXCTL
- DMA_MK_DST
- DMA_MK_GBLADDR
- DMA_MK_GBLCNT
- DMA_MK_GBLIDX
- DMA_MK_PRICTL
- DMA_MK_SECCTL
- DMA_MK_SRC
- DMA_MK_XFRCNT

❑ Structure

- DMA_CONFIG

Using a DMA Channel

To use a DMA channel, you must first open it and obtain a device handle using DMA_Open(). Once opened, you use the device handle to call the other API functions. The channel may be configured by passing a DMA_CONFIG structure to DMA_ConfigA() or by passing register values to the DMA_ConfigB() function. To assist in creating register values, there are *DMA_MK* (make) macros that construct register values based on field values. Additionally, there are symbol constants that may be used for the field values.

There are functions for managing shared global DMA registers, DMA_AllocGlobalReg(), DMA_FreeGlobalReg(), DMA_SetGlobalReg(), and DMA_GetGlobalReg().

For additional information about the DMA module, refer to Table 3–5 on page 3-4 and Section 4.6 on page 4-22.

2.7 EDMA

Currently, there are two DMA architectures used on ’C6x devices, DMA and EDMA (enhanced DMA). Devices such as the ’6201 have the DMA peripheral whereas the ’6211 has the EDMA peripheral. The two architectures are different enough to warrant a separate API module for each. The EDMA module includes:

- Constants
  - EDMA_CHA_CNT
  - EDMA_SUPPORT
  - EDMA_TABLE_CNT
- Functions
  - EDMA_AllocTable
  - EDMA_ClearChannel
  - EDMA_Close
  - EDMA_ConfigA
  - EDMA_ConfigB
  - EDMA_DisableChannel
  - EDMA_EnableChannel
  - EDMA_FreeTable
  - EDMA_GetChannel
  - EDMA_GetPriQStatus
  - EDMA_GetScratchAddr
  - EDMA_GetScratchSize
  - EDMA_GetTableAddress
  - EDMA_Open
  - EDMA_Reset
  - EDMA_SetChannel
- Macros
  - EDMA_MK_CNT
  - EDMA_MK_DST
- Macros (cont.)

- EDMA_MK_IDX
- EDMA_MK_OPT
- EDMA_MK_RLD
- EDMA_MK_SRC

❑ Structure

- EDMA_CONFIG

Using an EDMA Channel

To use an EDMA channel, you must first open it and obtain a device handle using EDMA_Open(). Once opened, use the device handle to call the other API functions. The channel may be configured by passing an EDMA_CONFIG structure to EDMA_ConfigA() or by passing register values to the EDMA_ConfigB() function. To assist in creating register values, there are *EDMA_MK* (make) macros that construct register values based on field values. Additionally, there are symbol constants that may be used for the field values.

There are functions for managing parameter RAM (PRAM) tables, EDMA_AllocTable() and EDMA_FreeTable().

For additional information about the EDMA module, refer to Table 3–6 on page 3-6 and Section 4.7 on page 4-51.

2.8 EMIF

The EMIF module has a simple API for configuring the EMIF registers.

The EMIF may be configured by passing an `EMIF_CONFIG` structure to `EMIF_ConfigA()` or by passing register values to the `EMIF_ConfigB()` function. To assist in creating register values, there are *EMIF_MK* (make) macros that construct register values based on field values. In addition, there are symbol constants that may be used for the field values. The EMIF module includes:

- Constant
  - `EMIF_SUPPORT`
- Functions
  - `EMIF_ConfigA`
  - `EMIF_ConfigB`
- Macros
  - `EMIF_MK_CECTL`
  - `EMIF_MK_GBLCTL`
  - `EMIF_MK_SDCTL`
  - `EMIF_MK_SDEXT`
  - `EMIF_MK_SDTIM`
- Typedef
  - `EMIF_CONFIG`

For additional information about the EMIF module, refer to Table 3–7 on page 3-7 and Section 4.8 on page 4-69.

2.9 HPI

The HPI module has a simple API for configuring the HPI registers. Functions are provided for reading HPI status bits and setting interrupt events. The HPI module includes:

- Constant
  - `HPI_SUPPORT`
- Functions
  - `HPI_GetDspint`
  - `HPI_GetEventId`
  - `HPI_GetFetch`
  - `HPI_GetHint`
  - `HPI_GetHrdy`
  - `HPI_GetHwob`
  - `HPI_SetDspint`
  - `HPI_SetHint`

For additional information about the HPI module, refer to Table 3–8 on page 3-8 and Section 4.9 on page 4-81.

2.10 IRQ

The IRQ module manages CPU interrupts. The IRQ module includes:

- Constants
  - IRQ_EVT_NNNN
  - IRQ_SUPPORT
- Functions
  - IRQ_Clear
  - IRQ_Disable
  - IRQ_Enable
  - IRQ_Map
  - IRQ_Set
  - IRQ_Test

2.11 MCBSP

The MCBSP module contains a set of API functions for configuring the McBSP registers. The MCBSP module includes:

- Constants
  - MCBSP_PORT_CNT
  - MCBSP_SUPPORT
- Functions
  - MCBSP_Close
  - MCBSP_ConfigA
  - MCBSP_ConfigB
  - MCBSP_EnableFsync
  - MCBSP_EnableRcv
  - MCBSP_EnableSrgr
  - MCBSP_EnableXmt
  - MCBSP_GetPins
  - MCBSP_GetRcvAddr
  - MCBSP_GetRcvEventId
  - MCBSP_GetXmtAddr
  - MCBSP_GetXmtEventId
  - MCBSP_Open
  - MCBSP_Read
  - MCBSP_Reset
  - MCBSP_Rfull
  - MCBSP_Rrdy
  - MCBSP_RsyncErr
  - MCBSP_SetPins
  - MCBSP_Write
  - MCBSP_Xempty
- Functions (cont.)
  - MCBSP_Xrdy

- MCBSP_XsyncErr

- Macros
  - MCBSP_MK_MCR
  - MCBSP_MK_PCR
  - MCBSP_MK_RCER
  - MCBSP_MK_RCR
  - MCBSP_MK_SPCR
  - MCBSP_MK_SRGR
  - MCBSP_MK_XCER
  - MCBSP_MK_XCR

- Structure
  - MCBSP_CONFIG

Using a MCBSP Port

To use a MCBSP port, you must first open it and obtain a device handle using MCBSP_Open(). Once opened, use the device handle to call the other API functions. The port may be configured by passing a MCBSP_CONFIG structure to MCBSP_ConfigA() or by passing register values to the MCBSP_ConfigB() function. To assist in creating register values, there are *MCBSP_MK* (make) macros that construct register values based on field values. In addition, there are symbol constants that may be used for the field values.

There are functions for directly reading and writing to the data registers DRR and DXR, MCBSP_Read() and MCBSP_Write(). The addresses of the DXR and DRR registers are also obtainable for use with DMA configuration, MCBSP_GetRcvAddr() and MCBSP_GetXmtAddr().

MCBSP status bits are easily read using efficient inline functions.

For additional information about the MCBSP module, refer to Table 3–10 on page 3-9 and Section 4.11 on page 4-87.

2.12 PWR

The PWR module is used to configure the power-down control registers, if applicable, and to invoke various power-down modes. The PWR module includes:

- Constant
  - PWR_SUPPORT
- Functions
  - PWR_ConfigB
  - PWR_PowerDown
- Macro
  - PWR_MK_PDCTL

For additional information about the PWR module, refer to Table 3–11 on page 3-10 and Section 4.12 on page 4-114.

2.13 STDINC

The STDINC module defines some identifiers that are globally useful to everyone and are used throughout the CSL source code. The application is free to use any identifiers defined here. The main set of identifiers are type definitions for integer data types. The names themselves denote whether the data is signed or unsigned and the number of bits, i.e. 8, 16, 32, or 40. The STDINC module includes:

- Constants
  - FALSE
  - INV
  - NO
  - TRUE
  - YES
- Typedef's
  - BOOL
  - INT16
  - INT32
  - INT40
  - INT8
  - UINT16
  - UINT32
  - UINT40
  - UINT8

2.14 TIMER

The TIMER module has a simple API for configuring the timer registers. The TIMER module includes:

- Constants
  - TIMER_DEVICE_CNT
  - TIMER_SUPPORT
- Functions
  - TIMER_Close
  - TIMER_ConfigA
  - TIMER_ConfigB
  - TIMER_GetCount
  - TIMER_GetDatin
  - TIMER_GetEventId
  - TIMER_GetPeriod
  - TIMER_GetTstat
  - TIMER_Open
  - TIMER_Pause
  - TIMER_Reset
  - TIMER_Resume
  - TIMER_SetCount
  - TIMER_SetDatout
  - TIMER_SetPeriod
  - TIMER_Start
- Macro
  - TIMER_MK_CTL
- Structure
  - TIMER_CONFIG

Using a TIMER Device

To use a TIMER device, you must first open it and obtain a device handle using TIMER_Open(). Once opened, use the device handle to call the other API

functions. The timer device may be configured by passing a TIMER_CONFIG structure to TIMER_ConfigA() or by passing register values to the TIMER_ConfigB() function. To assist in creating register values, there are *TIMER_MK* (make) macros that construct register values based on field values. In addition, there are symbol constants that may be used for the field values.

There is a delay function to do precise delays at the millisecond resolution: TIMER_Delay().

For additional information about the TIMER module, refer to Table 3–12 on page 3-11 and Section 4.14 on page 4-119.

Chapter 3

CSL API Function Tables

This chapter provides tables containing all CSL API functions, a brief description of each, and a page reference for more detailed information.

Topic **Page**

3.1 CSL Function Tables

Each of the tables in this section contains information about a specific CSL module and its APIs. The syntax is shown next to a column indicating the type of API: Function, Constant, Macro, or Structure. A brief description of each is provided along with a page reference for more detailed information.

Table 3–1. CACHE

Syntax	Type	Description	Page
CACHE_Clean	F	Cleans a specific cache region	4-3
CACHE_EnableCaching	F	Enables caching for a specified block of address space	4-4
CACHE_Flush	F	Flushes a region of cache	4-5
CACHE_GetL2SramSize	F	Returns current L2 size configured as SRAM	4-5
CACHE_Invalidate	F	Invalidates a region of cache	4-6
CACHE_Reset	F	Resets cache to power-on default	4-7
CACHE_SetL2Mode	F	Sets L2 cache mode	4-7
CACHE_SetPccMode	F	Sets program cache mode	4-9
CACHE_SUPPORT	C	A compile time constant whose value is 1 if the device supports the CACHE module	4-9

Note: F = Function; C = Constant; M = Macro; S = Structure; T = Typedef

Table 3–2. CSL

Syntax	Type	Description	Page
CSL_Init	F	Initializes the CSL library	4-10

Note: F = Function; C = Constant; M = Macro; S = Structure; T = Typedef

Table 3–3. DAT

Syntax	Type	Description	Page
DAT_Close	F	Closes the DAT module	4-11
DAT_Copy	F	Copies a linear block of data from Src to Dst using DMA or EDMA hardware	4-11
DAT_Copy2D	F	Performs a 2-dimensional data copy using DMA or EDMA hardware.	4-13
DAT_Fill	F	Fills a linear block of memory with the specified fill value using DMA or EDMA hardware	4-14
DAT_Open	F	Opens the DAT module	4-16
DAT_SUPPORT	C	A compile time constant whose value is 1 if the device supports the DAT module	4-18
DAT_Wait	F	Waits for a previous transfer to complete	4-18

Note: F = Function; C = Constant; M = Macro; S = Structure; T = Typedef

Table 3–4. CHIP

Syntax	Type	Description	Page
CHIP_6XXX	C	Current device identification symbols	4-19
CHIP_GetCpuId	F	Returns the CPU ID field of the CSR register	4-20
CHIP_GetEndian	F	Returns the current endian mode of the device	4-20
CHIP_GetMapMode	F	Returns the current map mode of the device	4-21
CHIP_GetRevId	F	Returns the CPU revision ID	4-21
CHIP_SUPPORT	C	A compile time constant whose value is 1 if the device supports the CHIP module	4-21

† This speed is only a software variable and in no way affects the actual chip operating frequency

Note: F = Function; C = Constant; M = Macro; S = Structure; T = Typedef

Table 3–5. DMA

Syntax	Type	Description	Page
DMA_AutoStart	F	Starts a DMA channel with autoinitialization	4-23
DMA_AllocGlobalReg	F	Provides resource management for the DMA global registers	4-22
DMA_CHA_CNT	C	Number of DMA channels for the current device	4-23
DMA_CLEAR_CONDITION	M	Clears one of the condition flags in the DMA secondary control register	4-24
DMA_Close	F	Closes a DMA channel opened via DMA_Open()	4-24
DMA_CONFIG	S	The DMA configuration structure used to set up a DMA channel	4-25
DMA_ConfigA	F	Sets up the DMA channel using the configuration structure	4-25
DMA_ConfigB	F	Sets up the DMA channel using the register values passed in	4-26
DMA_FreeGlobalReg	F	Frees a global DMA register previously allocated by calling DMA_AllocGlobalReg()	4-27
DMA_GET_CONDITION	M	Gets one of the condition flags in the DMA secondary control register	4-28
DMA_GetEventId	F	Returns the IRQ event ID for the DMA completion interrupt	4-28
DMA_GetGlobalReg	F	Reads a global DMA register that was previously allocated by calling DMA_AllocGlobalReg()	4-29
DMA_GetStatus	F	Reads the status bits of the DMA channel	4-30
DMA_MK_AUXCTL	M	For making a value suitable for the auxiliary control register	4-30
DMA_MK_DST	M	For making a value suitable for the destination address register	4-32
DMA_MK_GBLADDR	M	For making a value suitable for a global address register	4-33
DMA_MK_GBLCNT	M	For making a value suitable for a global count reload register	4-34
DMA_MK_GBLIDX	M	For making a value suitable for a global index register	4-35
DMA_MK_PRICTL	M	For making a value suitable for a primary control register	4-36
DMA_MK_SECCTL	M	For making a value suitable for a secondary control register	4-41
DMA_MK_SRC	M	For making a value suitable for the source address register	4-44
DMA_MK_XFRCNT	M	For making a value suitable for a transfer count register	4-45

Note: F = Function; C = Constant; M = Macro; S = Structure; T = Typedef

Table 3–5. DMA (Continued)

Syntax	Type	Description	Page
DMA_Open	F	Opens a DMA channel for use	4-46
DMA_Pause	F	Pauses the DMA channel by setting the START bits in the primary control register appropriately	4-47
DMA_Reset	F	Resets the DMA channel by setting its registers to power-on defaults	4-47
DMA_SetAuxCtl	F	Sets the DMA AUXCTL register	4-48
DMA_SetGlobalReg	F	Sets value of a global DMA register previously allocated by calling `DMA_AllocGlobalReg()`	4-48
DMA_Start	F	Starts a DMA channel running without autoinitialization	4-49
DMA_Stop	F	Stops a DMA channel by setting the START bits in the primary control register appropriately	4-49
DMA_SUPPORT	C	A compile time constant whose value is 1 if the device supports the DMA module	4-49
DMA_Wait	F	Enters a spin loop that polls the DMA status bits until the DMA completes	4-50

Note: F = Function; C = Constant; M = Macro; S = Structure; T = Typedef

Table 3–6. EDMA

Syntax	Type	Description	Page
EDMA_AllocTable	F	Allocates a parameter RAM table from PRAM	4-51
EDMA_CHA_CNT	C	Number of EDMA channels	4-51
EDMA_ClearChannel	F	Clears the EDMA event flag in the EDMA channel event register	4-52
EDMA_Close	F	Closes a previously opened EDMA channel	4-52
EDMA_CONFIG	S	The EDMA configuration structure used to set up an EDMA channel	4-53
EDMA_ConfigA	F	Sets up the EDMA channel using the configuration structure	4-54
EDMA_ConfigB	F	Sets up the EDMA channel using the EDMA parameter arguments	4-55
EDMA_DisableChannel	F	Disables an EDMA channel	4-56
EDMA_EnableChannel	F	Enables an EDMA channel	4-56
EDMA_FreeTable	F	Frees up a PRAM table previously allocated	4-57
EDMA_GetChannel	F	Returns the current state of the channel event	4-57
EDMA_GetPriQStatus	F	Returns the value of the priority queue status register (PQSR)	4-58
EDMA_GetScratchAddr	F	Returns the starting address of the EDMA PRAM used as non-cacheable on-chip SRAM (scratch area)	4-58
EDMA_GetScratchSize	F	Returns the size (in bytes) of the EDMA PRAM used as non-cacheable on-chip SRAM (scratch area)	4-58
EDMA_GetTableAddress	F	Returns the 32-bit absolute address of the table	4-59
EDMA_MK_CNT	M	For making a value suitable for the EDMA CNT parameter	4-59
EDMA_MK_DST	M	For making a value suitable for the EDMA DST parameter	4-60
EDMA_MK_IDX	M	For making a value suitable for the EDMA IDX parameter	4-61
EDMA_MK_OPT	M	For making a value suitable for the EDMA OPT parameter	4-62
EDMA_MK_RLD	M	For making a value suitable for the EDMA RLD parameter	4-64
EDMA_MK_SRC	M	For making a value suitable for the EDMA SRC parameter	4-65
EDMA_Open	F	Opens an EDMA channel	4-66
EDMA_Reset	F	Resets the given EDMA channel	4-67

Note: F = Function; C = Constant; M = Macro; S = Structure; T = Typedef

Table 3–6. EDMA (Continued)

Syntax	Type	Description	Page
EDMA_SetChannel	F	Triggers an EDMA channel by writing to the appropriate bit in the event set register (ESR)	4-68
EDMA_SUPPORT	C	A compile time constant whose value is 1 if the device supports the EDMA module	4-68
EDMA_TABLE_CNT	C	A compile time constant that holds the total number of parameter table entries in the EDMA PRAM	4-68

Note: F = Function; C = Constant; M = Macro; S = Structure; T = Typedef

Table 3–7. EMIF

Syntax	Type	Description	Page
EMIF_CONFIG	T	Structure used to set up the EMIF peripheral	4-69
EMIF_ConfigA	F	Sets up the EMIF using the configuration structure	4-69
EMIF_ConfigB	F	Sets up the EMIF using the register value arguments	4-70
EMIF_MK_CECTL	M	For making a value suitable for an EMIF CE space control register	4-71
EMIF_MK_GBLCTL	M	For making a value suitable for the EMIF global control register	4-73
EMIF_MK_SDCTL	M	For making a value suitable for the EMIF SDRAM control register	4-75
EMIF_MK_SDEXT	M	For making a value suitable for the EMIF SDRAM extension register	4-77
EMIF_MK_SDTIM	M	For making a value suitable for the EMIF SDRAM timing register	4-79
EMIF_SUPPORT	C	A compile time constant that has a value of 1 if the device supports the EMIF module	4-80

Note: F = Function; C = Constant; M = Macro; S = Structure; T = Typedef

Table 3–8. HPI

Syntax	Type	Description	Page
HPI_GetDspint	F	Reads the DSPINT bit from the HPIC register	4-81
HPI_GetEventId	F	Obtain the IRQ event associated with the HPI device	4-81
HPI_GetFetch	F	Reads the FETCH flag from the HPIC register and returns its value.	4-81
HPI_GetHint	F	Returns the value of the HINT bit of the HPIC register	4-81
HPI_GetHrdy	F	Returns the value of the HRDY bit of the HPIC register	4-82
HPI_GetHwob	F	Returns the value of the HWOB bit of the HPIC register	4-82
HPI_SetDspint	F	Writes the value to the DSPINT field of the HPIC register	4-82
HPI_SetHint	F	Writes the value to the HINT field of the HPIC register	4-82
HPI_SUPPORT	C	A compile time constant whose value is 1 if the device supports the HPI module	4-83

Note: F = Function; C = Constant; M = Macro; S = Structure; T = Typedef

Table 3–9. IRQ

Syntax	Type	Description	Page
IRQ_Clear	F	Clears the event flag from the IFR register	4-84
IRQ_Disable	F	Disables the specified event	4-84
IRQ_Enable	F	Enables the specified event	4-84
IRQ_EVT_NNNN	C	These are the IRQ events	4-85
IRQ_Map	F	Maps an event to a physical interrupt number by configuring the interrupt selector MUX registers	4-85
IRQ_Set	F	Sets the specified event by writing to the appropriate ISR register bit	4-86
IRQ_SUPPORT	C	A compile time constant whose value is 1 if the device supports the IRQ module	4-86
IRQ_Test	F	Allows testing an event to see if its flag is set in the IFR register	4-86

Note: F = Function; C = Constant; M = Macro; S = Structure; T = Typedef

Table 3–10. MCBSP

Syntax	Type	Description	Page
MCBSP_Close	F	Closes a MCBSP port previously opened via MCBSP_Open()	4-87
MCBSP_CONFIG	S	Used to setup a MCBSP port	4-87
MCBSP_ConfigA	F	Sets up the MCBSP port using the configuration structure	4-88
MCBSP_ConfigB	F	Sets up the MCBSP port using the register values passed in	4-89
MCBSP_EnableFsync	F	Enables the frame sync generator for the given port	4-90
MCBSP_EnableRcv	F	Enables the receiver for the given port	4-90
MCBSP_EnableSrgr	F	Enables the sample rate generator for the given port	4-90
MCBSP_EnableXmt	F	Enables the transmitter for the given port	4-91
MCBSP_GetPins	F	Reads the values of the port pins when configured as general purpose I/Os	4-91
MCBSP_GetRcvAddr	F	Returns the address of the data receive register (DRR)	4-92
MCBSP_GetRcvEventId	F	Retrieves the receive event ID for the given port	4-92
MCBSP_GetXmtAddr	F	Returns the address of the data transmit register, DXR	4-92
MCBSP_GetXmtEventId	F	Retrieves the transmit event ID for the given port	4-93
MCBSP_MK_MCR	M	Makes a value suitable for the multichannel control register	4-93
MCBSP_MK_PCR	M	Makes a value suitable for the pin control register	4-95
MCBSP_MK_RCER	M	Makes a value suitable for the receive channel enable register	4-97
MCBSP_MK_RCR	M	Makes a value suitable for the receive control register	4-98
MCBSP_MK_SPCR	M	Makes a value suitable for the serial port control register	4-101
MCBSP_MK_SRGR	M	Makes a value suitable for the sample rate generator register	4-103
MCBSP_MK_XCER	M	Makes a value suitable for the transmit channel enable register	4-105
MCBSP_MK_XCR	M	Makes a value suitable for the transmit control register	4-106
MCBSP_Open	F	Opens a McBSP port for use	4-108
MCBSP_PORT_CNT	C	A compile time constant that holds the number of serial ports present on the current device	4-109

Note: F = Function; C = Constant; M = Macro; S = Structure; T = Typedef

Table 3–10. MCBSP (Continued)

Syntax	Type	Description	Page
MCBSP_Read	F	Performs a direct 32-bit read of the data receive register DRR	4-109
MCBSP_Reset	F	Resets the given serial port	4-109
MCBSP_Rfull	F	Reads the RFULL bit of the serial port control register	4-110
MCBSP_Rrdy	F	Reads the RRDY status bit of the SPCR register	4-110
MCBSP_RsyncErr	F	Reads the RSYNCERR status bit of the SPCR register	4-111
MCBSP_SetPins	F	Sets the state of the serial port pins when configured as general purpose IO	4-111
MCBSP_SUPPORT	C	A compile time constant whose value is 1 if the device supports the MCBSP module	4-112
MCBSP_Write	F	Writes a 32-bit value directly to the serial port data transmit register, DXR	4-112
MCBSP_Xempty	F	Reads the XEMPTY bit from the SPCR register	4-112
MCBSP_Xrdy	F	Reads the XRDY status bit of the SPCR register	4-113
MCBSP_XsyncErr	F	Reads the XSYNCERR status bit of the SPCR register	4-113

Note: F = Function; C = Constant; M = Macro; S = Structure; T = Typedef

Table 3–11. PWR

Syntax	Type	Description	Page
PWR_ConfigB	F	Sets up the power-down logic using the register value passed in	4-114
PWR_MK_PDCTL	M	Makes a value suitable for the power-down control register	4-114
PWR_PowerDown	F	Forces the DSP to enter a power-down state	4-115
PWR_SUPPORT	C	A compile time constant whose value is 1 if the device supports the PWR module	4-116

Note: F = Function; C = Constant; M = Macro; S = Structure; T = Typedef

Note: F = Function; C = Constant; M = Macro; S = Structure; T = Typedef

Table 3–12. TIMER

Syntax	Type	Description	Page
TIMER_Close	F	Closes a previously opened timer device	4-119
TIMER_CONFIG	S	Structure used to set up a timer device	4-119
TIMER_ConfigA	F	Configure timer using configuration structure	4-120
TIMER_ConfigB	F	Sets up the timer using the register values passed in	4-120
TIMER_DEVICE_CNT	C	A compile time constant; number of timer devices present	4-121
TIMER_GetCount	F	Returns the current timer count value	4-121
TIMER_GetDatin	F	Reads the value of the TINP pin	4-121
TIMER_GetEventId	F	Obtains the event ID for the timer device	4-122
TIMER_GetPeriod	F	Returns the period of the timer device	4-122
TIMER_GetTstat	F	Reads the timer status; value of timer output	4-122
TIMER_MK_CTL	M	For making a value suitable for the timer control register	4-123
TIMER_Open	F	Opens a TIMER device for use	4-125
TIMER_Pause	F	Pauses the timer	4-125
TIMER_Reset	F	Resets the timer device	4-126
TIMER_Resume	F	Resumes the timer after a pause	4-126
TIMER_SetCount	F	Sets the count value of the timer	4-126
TIMER_SetDatout	F	Sets the data output value	4-127
TIMER_SetPeriod	F	Sets the timer period	4-127
TIMER_Start	F	Starts the timer device running	4-127
TIMER_SUPPORT	C	A compile time constant whose value is 1 if the device supports the TIMER module	4-128

Note: F = Function; C = Constant; M = Macro; S = Structure; T = Typedef

Chapter 4

CSL API Reference

This chapter provides an alphabetical list of the chip support library (CSL) service layer API functions, enumerations, type definitions, macros, structures, constants, and global variables.

Topic **Page**

4.1 CSL API Reference Introduction

Not all CSL API modules are supported on all devices. For example, the EDMA API module is not supported on the ’6201 because the ’6201 does not have EDMA hardware. When an API module is not supported, all of its header file information is conditionally compiled out, meaning the declarations will not exist. Because of this, calling an EDMA API function on devices not supporting EDMA will result in a compiler and/or linker error.

Table 4–1 shows which devices each API module is supported on:

Table 4–1. CSL API Module Support for TMS320C6000 Devices

Module	’6201	’6202	’6203	’6204	’6205	’6211	’6701	’6711
CACHE	X	X	X	X	X	X	X	X
DAT	X	X	X	X	X	X	X	X
CHIP	X	X	X	X	X	X	X	X
DMA	X	X	X	X	X		X	
EDMA						X		X
EMIF	X	X	X	X	X	X	X	X
HPI	X					X	X	X
IRQ	X	X	X	X	X	X	X	X
MCBSP	X	X	X	X	X	X	X	X
PWR	X	X	X	X	X	X	X	X
TIMER	X	X	X	X	X	X	X	X

4.2 CACHE

4.2.1 CACHE_Clean

Cleans a range of L2 cache

Function

```
void CACHE_Clean(
  CACHE_REGION Region,
  UINT32 Addr,
  UINT32 WordCt
);
```

Arguments

`Region`	Specifies which cache region to clean; must be one of the following: • `CACHE_L2` • `CACHE_L2ALL`
`Addr`	Beginning address of range to clean; word aligned
`WordCt`	Number of 4-byte words to clean

Return Value

`none`

Description

Cleans a range of L2 cache. All lines within the range defined by `Addr` and `WordCt` are cleaned out of L2. If `CACHE_L2ALL` is specified, then all of L2 is cleaned, `Addr` and `WordCt` are ignored. A clean operation involves writing back all dirty cache lines then invalidation of those lines. This routine waits until the operation completes before returning.

Note: This function does nothing on devices without L2 cache.

Example

If you want to clean a 4K-byte range that starts at 0x80000000 out of L2 use:

```
CACHE_Clean(CACHE_L2,0x80000000,0x00000400);
```

If you want to clean all lines out of L2 use:

```
CACHE_Clean(CACHE_L2All,0x00000000,0x00000000);
```

4.2.2 CACHE_EnableCaching *Specifies block of ext. memory for caching*

Function

```
void CACHE_EnableCaching(
  UINT32 Block
);
```

Arguments

`Block` Specifies a block of external memory to enable caching for; must one of the following:

- `CACHE_CE33 –(0xB3000000 to 0xB3FFFFFF)`
- `CACHE_CE32 –(0xB2000000 to 0xB2FFFFFF)`
- `CACHE_CE31 –(0xB1000000 to 0xB1FFFFFF)`
- `CACHE_CE30 –(0xB0000000 to 0xB0FFFFFF)`
- `CACHE_CE23 –(0xA3000000 to 0xA3FFFFFF)`
- `CACHE_CE22 –(0xA2000000 to 0xA2FFFFFF)`
- `CACHE_CE21 –(0xA1000000 to 0xA1FFFFFF)`
- `CACHE_CE20 –(0xA0000000 to 0xA0FFFFFF)`
- `CACHE_CE13 –(0x93000000 to 0x93FFFFFF)`
- `CACHE_CE12 –(0x92000000 to 0x92FFFFFF)`
- `CACHE_CE11 –(0x91000000 to 0x91FFFFFF)`
- `CACHE_CE10 –(0x90000000 to 0x90FFFFFF)`
- `CACHE_CE03 –(0x83000000 to 0x83FFFFFF)`
- `CACHE_CE02 –(0x82000000 to 0x82FFFFFF)`
- `CACHE_CE01 –(0x81000000 to 0x81FFFFFF)`
- `CACHE_CE00 –(0x80000000 to 0x80FFFFFF)`

Return Value

`none`

Description

Enables caching for the specified block of memory. This is accomplished by setting the CE bit in the appropriate memory attribute register (MAR). By default, caching is disabled for all memory spaces.

Note: This function does nothing on devices without L2 cache.

Example

To enable caching for the range of memory from 0x80000000 to 0x80FFFFFF use:

```
CACHE_EnableCaching(CACHE_CE00);
```

4.2.3 CACHE_Flush

Flushes a region of cache

Function

```
void CACHE_Flush(
  CACHE_REGION Region,
  UINT32 Addr,
  UINT32 WordCt
);
```

Arguments

`Region` Specifies which cache region to flush from; must be one of the following:

- `CACHE_L2`
- `CACHE_L2ALL`
- `CACHE_L1D`

`Addr` Starting address of memory range to flush

`WordCt` Beginning address of range to flush; word aligned

Return Value

`none` Number of 4-byte words to flush

Description

Flushes a range of L2 cache. All lines within the range defined by `Addr` and `WordCt` are flushed out of L2. If `CACHE_L2ALL` is specified, then all of L2 is flushed; `Addr` and `WordCt` are ignored. A flush operation involves writing back all dirty cache lines, but the lines are not invalidated. This routine waits until the operation completes before returning.

Note: This function does nothing on devices without L2 cache.

Example

If you want to flush a 4K-byte range that starts at 0x80000000 out of L2, use:

```
CACHE_Flush(CACHE_L2,0x80000000,0x00000400);
```

If you want to flush all lines out of L2, use:

```
CACHE_Flush(CACHE_L2ALL,0x00000000,0x00000000);
```

4.2.4 CACHE_GetL2SramSize

Returns current L2 size configured as SRAM

Function

```
UINT32 CACHE_GetL2SramSize();
```

Arguments

`none`

Return Value

`size` Returns number of bytes of on-chip SRAM

Description

This function returns the current size of L2 that is configured as SRAM.

Note: This function does nothing on devices without L2 cache.

Example

```
SramSize = CACHE_GetL2SramSize();
```

4.2.5 CACHE_Invalidate

Invalidates a region of cache

Function

```
void CACHE_Invalidate(
  CACHE_REGION Region,
  UINT32 Addr,
  UINT32 ByteCt
);
```

Arguments

`Region` Specifies which cache region to invalidate; must be one of the following:

- `CACHE_L1P` Invalidate L1P
- `CACHE_L1PALL` Invalidate all of L1P
- `CACHE_L1DALL` Invalidate all of L1D

`Addr` Beginning address of range to invalidate; word aligned

`ByteCt` Number of 4-byte words to invalidate

Return Value

`none`

Description

Invalidates a range from cache. All lines within the range defined by `Addr` and `WordCt` are invalidated from `Region`. If `CACHE_L1PALL` is specified, then all of L1P is invalidated; `Addr` and `WordCt` are ignored. Likewise, if `CACHE_L1DALL` is specified, then all of L1D is invalidated; `Addr` and `WordCt` are ignored. This routine waits until the operation completes before returning.

Note: This function does nothing on devices without L2 cache.

Example

If you want to invalidate a 4K-byte range that starts at 0x80000000 from L1P, use:

```
CACHE_Flush(CACHE_L1P,0x80000000,0x00000400);
```

If you want to invalidate all lines from L1D, use:

```
CACHE_Flush(CACHE_L1DALL,0x00000000,0x00000000);
```

4.2.6 CACHE_Reset

Resets cache to power-on default

Function `void CACHE_Reset();`

Arguments `none`

Return Value `none`

Description Resets cache to power-on default.

Devices with L2 Cache:

- PCC and DCC fields of CSR are set to zero (mapped)
- All MAR bits are cleared
- L2 mode set to all SRAM

Devices without L2 Cache:

- PCC field of CSR set to zero (mapped)

Note: If you reset the cache, any dirty data will be lost. If you want to preserve this data, flush it out first.

Example

```
CACHE_Reset();
```

4.2.7 CACHE_SetL2Mode

Sets L2 cache mode

Function

```
CACHE_L2MODE CACHE_SetL2Mode(
  CACHE_L2MODE NewMode
);
```

Arguments `NewMode` New L2 cache mode; must be one of the following:

- `CACHE_0KSRAM`
- `CACHE_16KSRAM`
- `CACHE_32KSRAM`
- `CACHE_48KSRAM`
- `CACHE_64KSRAM`
- `CACHE_0KCACHE`
- `CACHE_16KCACHE`
- `CACHE_32KCACHE`
- `CACHE_48KCACHE`
- `CACHE_64KCACHE`

Return Value `OldMode` Returns old cache mode; will be one of the following:

- `CACHE_0KSRAM`
- `CACHE_16KSRAM`
- `CACHE_32KSRAM`
- `CACHE_48KSRAM`
- `CACHE_64KSRAM`
- `CACHE_0KCACHE`
- `CACHE_16KCACHE`
- `CACHE_32KCACHE`
- `CACHE_48KCACHE`
- `CACHE_64KCACHE`

Description Sets the mode of the L2 cache. There are three conditions that may occur as a result of changing cache modes:

1. A decrease in cache size
2. An increase in cache size
3. No change in cache size

If the cache size decreases, all of L2 is flushed; then the mode is changed. If the cache size increases, the part of SRAM that is about to be turned into cache is flushed out of L1; then the mode is changed. Nothing happens when there is no change.

Increasing cache size means that some of the SRAM is lost. If you have data in SRAM that you do not want lost, you must preserve it yourself before changing cache modes.

Some of the cache modes are identical. For example, on the ’6211 there are 64K-bytes of L2; hence, `CACHE_16KSRAM` is equivalent to `CACHE_48KCACHE`. However, if the L2 size changes on a future device, this will not be the case.

Note: This function does nothing on devices without L2 cache.

Example

```
CACHE_L2MODE OldMode;

OldMode = CACHE_SetL2Mode(CACHE_32KCACHE);
```

4.2.8 CACHE_SetPccMode

Sets program cache mode

Function

```
CACHE_PCC CACHE_SetPccMode(
  CACHE_PCC NewMode
);
```

Arguments

`NewMode` New program cache mode; must be one of the following:

- `CACHE_PCCMAPPED`
- `CACHE_PCCENABLE`
- `CACHE_PCCFREEZE`
- `CACHE_PCCBYPASS`

Return Value

`OldMode` Returns the old program cache mode; will be one of the following:

- `CACHE_PCCMAPPED`
- `CACHE_PCCENABLE`
- `CACHE_PCCFREEZE`
- `CACHE_PCCBYPASS`

Description

This function sets the program cache mode for devices that don't have an L2 cache. For devices that do have an L2 cache such as the '6211, this function does nothing. See the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for the meaning of the cache modes.

Example

To enable the program cache in normal mode, use:

```
CACHE_PCC OldMode;

OldMode = CACHE_SetPccMode(CACHE_PCCENABLE);
```

4.2.9 CACHE_SUPPORT

A compile time constant whose value is 1 if the device supports the CACHE module

Constant

`CACHE_SUPPORT`

Description

Compile time constant that has a value of 1 if the device supports the CACHE module and 0 otherwise. You are not required to use this constant.

Currently, all devices support this module.

Example

```
#if (CACHE_SUPPORT)
   /* user cache configuration */
#endif
```

4.3 CSL

4.3.1 CSL_Init

Calls the initialization function of all CSL API modules

Function `void CSL_Init();`

Arguments `none`

Return Value `none`

Description The CSL module is the top-level API module whose primary purpose is to initialize the library. Only one function is exported:

- `CSL_Init()`

The *CSL_Init()* function must be called once at the beginning of your program before calling any other CSL API functions.

Example

```
CSL_Init();
```

4.4 DAT

4.4.1 DAT_Close

Closes the DAT module

Function `void DAT_Close();`

Arguments `none`

Return Value `none`

Description Closes the DAT module. First, any pending requests are allowed to complete; then if applicable, any DMA channels are closed.

Example

```
DAT_Close();
```

4.4.2 DAT_Copy

Copies a linear block of data from Src to Dst using DMA or EDMA hardware

Function

```
UINT32 DAT_Copy(
  void *Src,
  void *Dst,
  UINT16 ByteCnt
);
```

Arguments

`Src`	Pointer to source data
`Dst`	Pointer to destination location
`ByteCnt`	Number of bytes to copy

Return Value `UINT32` Transfer ID

Description Copies a linear block of data from `Src` to `Dst` using DMA or EDMA hardware depending on the device. The arguments are checked for alignment and the DMA is submitted accordingly. For best performance, you should ensure that the source and destination addresses are aligned on a 4-byte boundary and the transfer length is a multiple of 4. A maximum of 65,535 bytes may be copied. A ByteCnt of zero has unpredictable results.

If the DMA channel is busy with one or more previous requests, the function will block and wait for completion before submitting this request.

The DAT module must be opened before calling this function. See `DAT_Open()`.

The return value is a transfer identifier that may be used later on to wait for completion. See `DAT_Wait()`.

Example

```
#define DATA_SIZE 256
UINT32 BuffA[DATA_SIZE/sizeof(UINT32)];
UINT32 BuffB[DATA_SIZE/sizeof(UINT32)];
…
DAT_Open(DAT_CHAANY,DAT_PRI_LOW,0);
DAT_Copy(BuffA,BuffB,DATA_SIZE);
…
```

4.4.3 DAT_Copy2D

Perform 2-dimensional data copy

Function

```
UINT32 DAT_Copy2D(
  UINT32 Type,
  void *Src,
  void *Dst,
  UINT16 LineLen,
  UINT16 LineCnt,
  UINT16 LinePitch
);
```

Arguments

`Type`	Transfer type: ❑ DAT_1D2D ❑ DAT_2D1D ❑ DAT_2D2D
`Src`	Pointer to source data
`Dst`	Pointer to destination location
`LineLen`	Number of bytes per line
`LineCnt`	Number of lines
`LinePitch`	Number of bytes between start of one line to start of next line

Return Value `UINT32` Transfer ID

Description

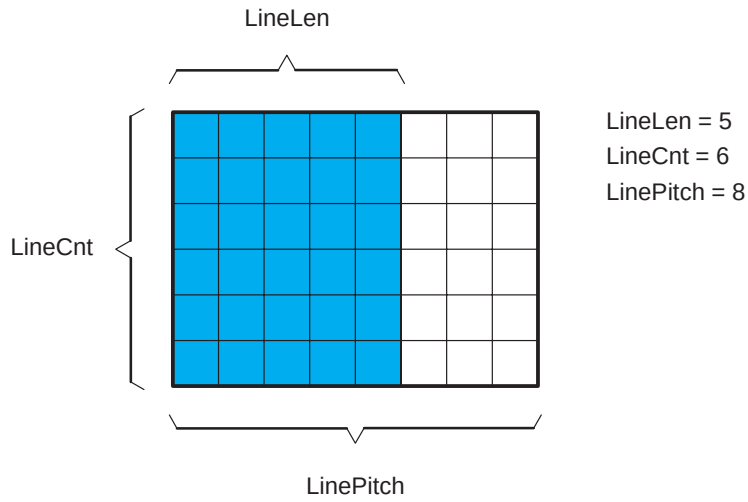
Performs a 2-dimensional data copy using DMA or EDMA hardware depending on the device. The arguments are checked for alignment and the hardware configured accordingly. For best performance, ensure that the source address and destination address are aligned on a 4-byte boundary and also ensure that the LineLen and LinePitch are multiples of 4-bytes.

If the channel is busy with previous requests, this function will block (spin) and wait till it frees up before submitting this request.

Note: The DAT module must be opened with the DAT_OPEN_2D flag before calling this function. See DAT_Open().

There are 3 ways to submit a 2D transfer, 1D to 2D, 2D to 1D, and 2D to 2D. This is specified using the type argument. In all cases, the number of bytes copied is LineLen X LineCnt. The 1D part of the transfer is just a linear block of data. The 2D part is illustrated below:

Figure 4–1. 2D Transfer



If a 2D to 2D transfer is specified, both the source and destination have the same LineLen, LineCnt, and LinePitch.

The return value is a transfer identifier that may be used later on to wait for completion. See DAT_Wait().

Example

```
DAT_Copy2D (DAT_1D2D, buffA, buffB, 16, 8, 32);
```

4.4.4 DAT_Fill

Fills a linear block of memory with the specified fill value using DMA hardware

Function

```
UINT32 DAT_Fill(
  void *Dst,
  UINT16 ByteCnt,
  UINT32 *Value
);
```

Arguments	`Dst`	Pointer to destination location
	`ByteCnt`	Number of bytes to fill
	`Value`	Pointer to fill value
Return Value	`none`	Transfer ID

Description

Fills a linear block of memory with the specified fill value using DMA hardware. The arguments are checked for alignment and the DMA is submitted accordingly. For best performance, you should ensure that the destination address is aligned on a 4-byte boundary and the transfer length is a multiple of 4. A maximum of 65,535 bytes may be filled.

The fill value is 8-bits in size but must be contained in a 32-bit word. This is due to the way the DMA hardware works. If the arguments are 32-bit aligned, then the DMA transfer element size is set to 32-bits to maximize performance. This means that the source of the transfer, the fill value, must be 32-bits in size. So, the 8-bit fill value must be repeated to fill the 32-bit value. For example, if you want to fill a region of memory with the value 0xA5, the fill value should contain 0xA5A5A5A5 before calling this function. If the arguments are 16-bit aligned, a 16-bit element size is used. Finally, if any of the arguments are 8-bit aligned, an 8-bit element size is used. It is a good idea to always fill in the entire 32-bit fill value, this eliminates any endian issues.

If the DMA channel is busy with a previous request, the function will block and wait for completion before submitting this request.

The DAT module must be opened before calling this function. See `DAT_Open()`.

The return value is a transfer identifier that may be used later on to wait for completion. See `DAT_Wait()`.

Note: You should be aware that if the fill value is in cache, the DMA always uses the external address and not the value that is in cache. It is up to you to ensure that the fill value is flushed before calling this function. Also, since the user specifies a pointer to the fill value, it is important not to write to it while the fill is in progress.

Example

```
UINT32 BUFF_SIZE 256;
UINT32 Buff[BUFF_SIZE/sizeof(UINT32)];
UINT32 FillValue = 0xA5A5A5A5;
…
DAT_Open(DAT_CHAANY,DAT_PRI_LOW,0);
DAT_Fill(Buff,BUFF_SIZE,&FillValue);
```

4.4.5 DAT_Open

Opens the DAT module

Function

```
void DAT_Open(
  int ChaNum,
  int Priority,
  UINT32 Flags
);
```

Arguments

`ChaNum` — Specifies which DMA channel to allocate; must be one of the following:

- `DAT_CHAANY`
- `DAT_CHA0`
- `DAT_CHA1`
- `DAT_CHA2`
- `DAT_CHA3`

`Priority` — Specifies the priority of the DMA channel; must be one of the following:

- `DAT_PRI_LOW`
- `DAT_PRI_HIGH`

`Flags` — Miscellaneous open flags

- `DAT_OPEN_2D`

Return Value `none`

Description

This function opens up the DAT module and must be called before calling any of the other DAT API functions. The `ChaNum` argument specifies which DMA channel to open for exclusive use by the DAT module. For devices with EDMA, the `ChaNum` argument is ignored because the quick DMA is used which doesn't have a channel associated with it. Currently, there are no flags defined and the argument should be set to zero.

For DMA Devices:

- `ChaNum` specifies which DMA channel to use
- `DAT_PRI_LOW` sets the DMA channel up for CPU priority
- `DAT_PRI_HIGH` sets the DMA channel up for DMA priority

For EDMA Devices:

- `ChaNum` is ignored
- `DAT_PRI_LOW` sets LOW priority
- `DAT_PRI_HIGH` sets HIGH priority

Once the DAT module is opened, any resources allocated, such as DMA channels, remain allocated. You can call `DAT_Close()` that frees these resources.

If 2D transfers are planned via DAT_Copy2D, the DAT_OPEN_2D flag must be specified. Specifying this flag for devices with the DMA peripheral will cause allocation of one global count reload register and one global index register. These global registers are freed when DAT_Close() is called.

Example

To open the DAT module using any available DMA channel, use:

```
DAT_Open(DAT_CHAANY,DAT_PRI_LOW,0);
```

To open the DAT module using DMA channel 2 in high priority mode, use:

```
DAT_Open(DAT_CHA2,DAT_PRI_HIGH,0);
```

To open the DAT module for 2D copies, use:

```
DAT_Open (DAT_CHAANY, DAT_PRI_HIGH, DAT_OPEN_2D);
```

4.4.6 DAT_Support

A compile time constant whose value is 1 if the device supports the DAT module

Constant `DAT_SUPPORT`

Description Compile time constant that has a value of 1 if the device supports the DAT module and 0 otherwise. You are not required to use this constant.

Currently, all devices support this module.

Example

```
#if (DAT_SUPPORT)
  /* user DAT configuration */
#endif
```

4.4.7 DAT_Wait

Waits for a previous transfer to complete identification by the transfer ID

Function

```
void DAT_Wait(
  UINT32 Id
);
```

Arguments

`Id`	Transfer identifier, returned by one of the DAT copy or DAT fill routines.

Return Value `none`

Description This function waits for a previous transfer to complete, identified by the transfer ID. If the transfer has already completed, this function returns immediately. Interrupts are disabled during the wait.

Example

```
UINT32 TransferId;
...
DAT_Open(DAT_CHAANY, DAT_PRI_LOW, 0);
...
TransferId = DAT_Copy(src,dst,len);
/* user DAT configuration */
DAT_Wait(TransferId);
```

4.5 CHIP

4.5.1 CHIP_6XXX

Current chip identification symbols

Constant

```
CHIP_6201
CHIP_6202
CHIP_6203
CHIP_6204
CHIP_6205
CHIP_6211
CHIP_6211X
CHIP_6701
CHIP_6711
```

Description

These are the current chip identification symbols. They are used throughout the CSL code to make compile-time decisions. When using the CSL, it is up to the user to define one and only one of these symbols. This is necessary because one common set of CSL header files is used for all versions of the library. These header files might define things differently, depending on the device. So, before including any CSL header files, you must define which device is being used. This can be done at the compiler command line by using the *–d* option, or if using *Code Composer Studio*, in the "*Project–>Options–>Compiler–>Preprocessor–>Define Symbols*" dialog.

You may also use these symbols to perform conditional compilation.

i.e.

```
#if (CHIP_6201)
  /* user CHIP configuration for 6201 /
#elif (CHIP_6211)
  / user CHIP configuration for 6211 */
#endif
```

Example

```
cl6x –dCHIP_6201 mycode.c
```

4.5.2 CHIP_GetCpuId

Returns the CPU ID field of the CSR register

Function `UINT32 CHIP_GetCpuId();`

Arguments `none`

Return Value `CPU ID` Returns the CPU ID

Description This function returns the CPU ID field of the CSR register.

Example

```
UINT32 CpuId;
CpuId = CHIP_GetCpuId();
```

4.5.3 CHIP_GetEndian

Returns the current endian mode of the device

Function `int CHIP_GetEndian();`

Arguments `none`

Return Value `endian mode` Returns the current endian mode of the device; will be one of the following:

- `CHIP_ENDIAN_BIG`
- `CHIP_ENDIAN_LITTLE`

Description Returns the current endian mode of the device as determined by the EN bit of the CSR register.

Example

```
UINT32 Endian;
…
Endian = CHIP_GetEndian();
if (Endian == CHIP_ENDIAN_BIG) {
  /* user big endian configuration /
} else {
  / user little endian configuration */
}
```

4.5.4 CHIP_GetMapMode

Returns the current map mode of the device

Function `int CHIP_GetMapMode();`

Arguments `none`

Return Value

`map mode` Returns current device MAP mode; will be one of the following:

- `CHIP_MAP_0`
- `CHIP_MAP_1`

Description Returns the current MAP mode of the device as determined by the MAP bit of the EMIF global control register.

Example

```
UINT32 MapMode;
…
MapMode = CHIP_GetMapMode();
if (MapMode == CHIP_MAP_0) {
  /* user map 0 configuration /
} else {
  / user map 1 configuration */
}
```

4.5.5 CHIP_GetRevId

Returns the CPU revision ID

Function `UINT32 CHIP_GetRevId();`

Arguments `none`

Return Value `revision ID` Returns CPU revision ID

Description This function returns the CPU revision ID as determined by the *Revision ID* field of the CSR register.

Example

```
UINT32 RevId;
RevId = CHIP_GetRevId();
```

4.5.6 CHIP_SUPPORT

A compile time constant whose value is 1 if the device supports the CHIP module

Constant `CHIP_SUPPORT`

Description Compile time constant that has a value of 1 if the device supports the CHIP module and 0 otherwise. You are not required to use this constant.

Currently, all devices support this module.

Example

```
#if (CHIP_SUPPORT)
  /* user CHIP configuration */
#endif
```

4.6 DMA

4.6.1 DMA_AllocGlobalReg

Allocates a global DMA register

Function

```
UINT32 DMA_AllocGlobalReg(
  DMA_GBL RegType,
  UINT32 InitVal
);
```

Arguments

`RegType`	Global register type; must be one of the following: • `DMA_GBL_ADDRRLD` • `DMA_GBL_INDEX` • `DMA_GBL_CNTRLD` • `DMA_GBL_SPLIT`
`InitVal`	Value to initialize the register to

Return Value

`Global Register ID`	Unique ID number for the global register

Description

Since the DMA global registers are shared, they must be controlled using resource management. This is done using `DMA_AllocGlobalReg()` and `DMA_FreeGlobalReg()` functions. Allocating a register ensures that it will not be allocated again until it is freed. The register ID may then be used to get or set the register value by calling `DMA_GetGlobalReg()` and `DMA_SetGlobalReg()` respectively. If the register cannot be allocated, a register id of 0 is returned.

The register ID may be directly used with the `DMA_MK_PRICTL` macro.

- **`DMA_GBL_ADDRRLD`**
  Allocate global address register for use as DMA DST RELOAD or DMA SRC RELOAD. Will allocate one of the following DMA registers:
  - Global Address Register B
  - Global Address Register C
  - Global Address Register D
- **`DMA_GBL_INDEX`**
  Allocate global index register for use as DMA INDEX. Will allocate one of the following DMA registers:
  - Global Index Register A
  - Global Index Register B

- **DMA_GBL_CNTRLD**
  Allocate global count reload register for use as DMA CNT RELOAD. Will allocate one of the following DMA registers:
  - Global Count Reload Register A
  - Global Count Reload Register B
- **DMA_GBL_SPLIT**
  Allocate global address register for use as DMA SPLIT. Will allocated one of the following DMA registers:
  - Global Address Register A
  - Global Address Register B
  - Global Address Register C

Example

```
UINT32 RegId;
...
/* allocate global index register and initialize it
*/
RegId = DMA_AllocGlobalReg(DMA_GBL_
INDEX,0x00200040);
```

4.6.2 DMA_AutoStart

Starts a DMA channel with autoinitialization

Function

```
void DMA_AutoStart(
  DMA_HANDLE hDma
);
```

Arguments `hDma` Handle to DMA channel, see `DMA_Open()`

Return Value `none`

Description Starts a DMA channel running with autoinitialization by setting the START bits in the primary control register accordingly. See also `DMA_Pause()`, `DMA_Stop()`, and `DMA_Start()`.

Example

```
DMA_AutoStart(hDma);
```

4.6.3 DMA_CHA_CNT

Number of DMA channels for the current device

Constant `DMA_CHA_CNT`

Description This constant holds the number of physical DMA channels for the current device.

4.6.4 DMA_CLEAR_CONDITION

Clears one of the condition flags in the DMA secondary control register

Macro

```
DMA_CLEAR_CONDITION(
  hDma,
  COND
);
```

Arguments

`hDma` Handle to DMA channel, see `DMA_Open()`

`COND` Condition to clear, must be one of the following:

- `DMA_SECCTL_SXCOND`
- `DMA_SECCTL_FRAMECOND`
- `DMA_SECCTL_LASTCOND`
- `DMA_SECCTL_BLOCKCOND`
- `DMA_SECCTL_RDROPCOND`
- `DMA_SECCTL_WDROPCOND`
- `DMA_SECCTL_RSYNCSTAT`
- `DMA_SECCTL_WSYNCSTAT`

Return Value `none`

Description This macro clears one of the condition flags in the DMA secondary control register. See the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for a description of the condition flags.

Example

```
DMA_CLEAR_CONDITION(hDma,DMA_SECCTL_BLOCKCOND);
```

4.6.5 DMA_Close

Closes a DMA channel opened via `DMA_Open()`

Function

```
void DMA_Close(
  DMA_HANDLE hDma
);
```

Arguments

`hDma` Handle to DMA channel, see `DMA_Open()`

Return Value `none`

Description This function closes a DMA channel previously opened via `DMA_Open()`. The registers for the DMA channel are set to their power-on defaults and the completion interrupt is disabled and cleared.

Example

```
DMA_Close(hDma);
```

4.6.6 DMA_CONFIG

The DMA configuration structure used to set up a DMA channel

Structure `DMA_CONFIG`

Members

`UINT32 prictl`	DMA primary control register value
`UINT32 secctl`	DMA secondary control register value
`UINT32 src`	DMA source address register value
`UINT32 dst`	DMA destination address register value
`UINT32 xfrcnt`	DMA transfer count register value

Description This is the DMA configuration structure used to set up a DMA channel. You create and initialize this structure then pass its address to the `DMA_ConfigA()` function. You can use literal values or the *DMA_MK* macros to create the structure member values.

Example

```
DMA_CONFIG MyConfig = {
  0x00000050, /* prictl */
  0x00000080, /* secctl */
  0x80000000, /* src    */
  0x80010000, /* dst    */
  0x00200040  /* xfrcnt */
};
...
DMA_ConfigA(hDma,&MyConfig);
```

4.6.7 DMA_ConfigA

Sets up the DMA channel using the configuration structure

Function

```
void DMA_ConfigA(
  DMA_HANDLE hDma,
  DMA_CONFIG *Config
);
```

Arguments

`hDma`	Handle to DMA channel. See `DMA_Open()`
`Config`	Pointer to an initialized configuration structure

Return Value `None`

Description Sets up the DMA channel using the configuration structure. The values of the structure are written to the DMA registers. The primary control register (*prictl*) is written last. See also `DMA_ConfigB()` and `DMA_CONFIG`.

Example

```
DMA_CONFIG MyConfig = {
  0x00000050, /* prictl */
  0x00000080, /* secctl */
  0x80000000, /* src    */
  0x80010000, /* dst    */
  0x00200040  /* xfrcnt */
};
…
DMA_ConfigA(hDma,&MyConfig);
```

4.6.8 DMA_ConfigB

Sets up the DMA channel using the register values passed in

Function

```
void DMA_ConfigB(
  DMA_HANDLE hDma,
  UINT32 prictl,
  UINT32 secctl,
  UINT32 src,
  UINT32 dst,
  UINT32 xfrcnt
);
```

Arguments

`hDma`	Handle to DMA channel. See `DMA_Open()`
`prictl`	Primary control register value
`secctl`	Secondary control register value
`src`	Source address register value
`dst`	Destination address register value
`xfrcnt`	Transfer count register value

Return Value `none`

Description Sets up the DMA channel using the register values passed in. The register values are written to the DMA registers. The primary control register (*prictl*) is written last. See also `DMA_ConfigA()`.

You may use literal values for the arguments or for readability. You may use the *DMA_MK* macros to create the register values based on field values.

Example

```
DMA_ConfigB(hDma,
  0x00000050, /* prictl */
  0x00000080, /* secctl */
  0x80000000, /* src    */
  0x80010000, /* dst    */
  0x00200040  /* xfrcnt */
);
```

4.6.9 DMA_FreeGlobalReg

Frees a global DMA register previously allocated by calling `DMA_AllocGlobalReg()`

Function

```
void DMA_FreeGlobalReg(
  UINT32 RegId
);
```

Arguments

`RegId`	Global register ID obtained from `DMA_AllocGlobalReg()`.

Return Value

`none`

Description

This function frees a global DMA register that was previously allocated by calling `DMA_AllocGlobalReg()`. Once freed, the register is available for allocation again.

Example

```
UINT32 RegId;
…
/* allocate global index register and initialize it /
RegId = DMA_AllocGlobalReg(DMA_GBL_INDEX,0x00200040);
…
/ some time later on when you’re done with it */
DMA_FreeGlobalReg(RegId);
```

4.6.10 DMA_GET_CONDITION

Gets one of the condition flags in the DMA secondary control register

Macro

```
DMA_GET_CONDITION(
  hDma,
  COND
);
```

Arguments

`hDma`	Handle to DMA channel. See `DMA_Open()`
`COND`	Condition to get; must be one of the following:

- `DMA_SECCTL_SXCOND`
- `DMA_SECCTL_FRAMECOND`
- `DMA_SECCTL_LASTCOND`
- `DMA_SECCTL_BLOCKCOND`
- `DMA_SECCTL_RDROPCOND`
- `DMA_SECCTL_WDROPCOND`
- `DMA_SECCTL_RSYNCSTAT`
- `DMA_SECCTL_WSYNCSTAT`

Return Value `Condtion` Condition, 0 if clear, 1 if set

Description This macro gets one of the condition flags in the DMA secondary control register. See the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for a description of the condition flags.

Example

```
if (DMA_GET_CONDITION(hDma,DMA_SECCTL_BLOCKCOND)) {
  /* user DMA configuration */
}
```

4.6.11 DMA_GetEventId

Returns the IRQ event ID for the DMA completion interrupt

Function

```
UINT32 DMA_GetEventId(
  DMA_HANDLE hDma
);
```

Arguments

`hDma`	Handle to DMA channel. See `DMA_Open()`

Return Value `Event ID` IRQ Event ID for DMA Channel

Description Returns the IRQ Event ID for the DMA completion interrupt. Use this ID to manage the event using the IRQ module.

Example

```
EventId = DMA_GetEventId(hDma);
IRQ_Enable(EventId);
```

4.6.12 DMA_GetGlobalReg

Reads a global DMA register that was previously allocated by calling DMA_AllocGlobalReg()

Function

```
UINT32 DMA_GetGlobalReg(
  UINT32 RegId
);
```

Arguments

RegId	Global register ID obtained from DMA_AllocGlobalReg().

Return Value

Register Value	Value read from register

Description

This function returns the register value of the global DMA register that was previously allocated by calling DMA_AllocGlobalReg().

If you prefer not to use the alloc/free paradigm for the global register management, the predefined register IDs may be used. You should be aware that use of predefined register IDs precludes the use of alloc/free. The list of predefined IDs are shown below:

- DMA_GBL_ADDRRLDB
- DMA_GBL_ADDRRLDC
- DMA_GBL_ADDRRLDD
- DMA_GBL_INDEXA
- DMA_GBL_INDEXB
- DMA_GBL_CNTRLDA
- DMA_GBL_CNTRLDB
- DMA_GBL_SPLITA
- DMA_GBL_SPLITB
- DMA_GBL_SPLITC

Note: DMA_GBL_ADDRRLDB denotes the same physical register as DMA_GBL_SPLITB and DMA_GBL_ADDRRLDC denotes the same physical register as DMA_GBL_SPLITC.

Example

```
UINT32 RegId;
UINT32 RegValue;
…
/* allocate global index register and initialize it
/
RegId = DMA_AllocGlobalReg(DMA_GBL_
INDEX,0x00200040);
…
RegValue = DMA_GetGlobalReg(RegId);
```

4.6.13 DMA_GetStatus

Reads the status bits of the DMA channel

Function

```
UINT32 DMA_GetStatus(
  DMA_HANDLE hDma
);
```

Arguments

`hDma`	Handle to DMA channel, see `DMA_Open()`

Return Value

`Status Value`	Current DMA channel status: • `DMA_STATUS_STOPPED` • `DMA_STATUS_RUNNING` • `DMA_STATUS_PAUSED` • `DMA_STATUS_AUTORUNNING`

Description

This function reads the STATUS bits of the DMA channel

Example

```
while (DMA_Status(hDma)==DMA_STATUS_RUNNING);
```

4.6.14 DMA_MK_AUXCTL

Makes a value suitable for the auxiliary control register

Macro

```
DMA_MK_AUXCTL(
  chpri,
  auxpri
)
```

Arguments

`chpri`	DMA channel priority: • `DMA_AUXCTL_CHPRI_HIGHEST` • `DMA_AUXCTL_CHPRI_2ND` • `DMA_AUXCTL_CHPRI_3RD` • `DMA_AUXCTL_CHPRI_4TH` • `DMA_AUXCTL_CHPRI_LOWEST`
`auxpri`	Auxiliary channel priority mode: • `DMA_AUXCTL_AUXPRI_CPU` • `DMA_AUXCTL_AUXPRI_DMA`

Return Value

`AUXCTL Value`	Constructed register value

Description

Use this macro to make a value suitable for the auxiliary control register.

The power-on default value is `DMA_AUXCTL_DEFAULT`.

Use of the *DMA_MK* macros makes it simpler to construct register values based on field values. You have a choice of using integer constants, integer variables, or the symbolic constants for arguments. All field values are right justified.

Refer to the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for descriptions of the arguments.

Example

```
UINT32 AuxCtl;

AuxCtl = DMA_MK_AUXCTL(0,1);

AuxCtl = DMA_MK_AUXCTL(
  DMA_AUXCTL_CHPRI_HIGHEST,
  DMA_AUXCTL_AUXPRI_DMA
);
```

4.6.15 DMA_MK_DST

Makes a value suitable for the destination address register

Macro

```
DMA_MK_DST(
  dst
)
```

Arguments

`dst`	Destination address: • `DMA_DST_DST_OF(x)`

Return Value

`DST Value`	Constructed register value

Description

Use this macro to make a value suitable for the destination address register. Although not really necessary, this macro is included for orthogonality and code readability.

The power-on default value is `DMA_DST_DEFAULT`.

Use of the *DMA_MK* macros makes it simpler to construct register values based on field values. You have a choice of using integer constants, integer variables, or the symbolic constants for arguments. All field values are right justified.

Refer to the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for descriptions of the arguments.

Example

```
UINT32 Dst;

Dst = DMA_MK_DST(0x80000000);

Dst = DMA_MK_DST(
 DMA_DST_DST_OF(0x80000000)
);
```

4.6.16 DMA_MK_GBLADDR

Makes a value suitable for a global address register

Macro

```
DMA_MK_GBLADDR(
  gbladdr
)
```

Arguments

`gbladdr`	Global address value: • `DMA_GBLADDR_GBLADDR_OF(x)`

Return Value

`GBLADDR Value`	Constructed register value

Description

Use this macro to make a value suitable for a global address register. Although not really necessary, this macro is included for orthogonality and code readability.

Use of the *DMA_MK* macros makes it simpler to construct register values based on field values. You have a choice of using integer constants, integer variables, or the symbolic constants for arguments. All field values are right justified.

Refer to the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for descriptions of the arguments.

Example

```
UINT32 GblAddr;

GblAddr = DMA_MK_GBLADDR(0x80000000);

GblAddr = DMA_MK_GBLADDR(
  DMA_GBLADDR_GBLADDR_OF(0x80000000)
);
```

4.6.17 DMA_MK_GBLCNT

Makes a value suitable for a global count reload register

Macro

```
DMA_MK_GBLCNT(
  elecnt,
  frmcnt
)
```

Arguments

`elecnt` Element count:

- `DMA_GBLCNT_ELECNT_OF(x)`

`frmcnt` Frame count:

- `DMA_GBLCNT_FRMCNT_OF(x)`

Return Value

`GBLCNT Value` Constructed register value

Description

Use this macro to make a value suitable for a global count reload register.

Use of the *DMA_MK* macros makes it simpler to construct register values based on field values. You have a choice of using integer constants, integer variables, or the symbolic constants for arguments. All field values are right justified.

Refer to the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for descriptions of the arguments.

Example

```
UINT32 GblCnt;

GblCnt = DMA_MK_GBLCNT(0x0100,0x0020);

GblCnt = DMA_MK_GBLCNT(
  DMA_GBLCNT_ELECNT_OF(0x0100),
  DMA_GBLCNT_FRMCNT_OF(0x0020)
);
```

4.6.18 DMA_MK_GBLIDX

Makes a value suitable for a global index register

Macro

```
DMA_MK_GBLIDX(
  eleidx,
  frmidx
)
```

Arguments

`eleidx` Element index:

- `DMA_GBLCNT_ELEIDX_OF(x)`

`frmidx` Frame index:

- `DMA_GBLCNT_FRMIDX_OF(x)`

Return Value

`GBLIDX Value` Constructed register value

Description

Use this macro to make a value suitable for a global index register.

Use of the *DMA_MK* macros makes it simpler to construct register values based on field values. You have a choice of using integer constants, integer variables, or the symbolic constants for arguments. All field values are right justified.

Refer to the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for descriptions of the arguments.

Example

```
UINT32 GblIdx;

GblIdx = DMA_MK_GBLIDX(0x0100,0x0020);

GblIdx = DMA_MK_GBLIDX(
  DMA_GBLCNT_ELEIDX_OF(0x0100),
  DMA_GBLCNT_FRMIDX_OF(0x0020)
);
```

4.6.19 DMA_MK_PRICTL

Makes a value suitable for a primary control register

Macro

```
DMA_MK_PRICTL(
  start,
  srcdir,
  dstdir,
  esize,
  split,
  cntrld,
  index,
  rsync,
  wsync,
  pri,
  tcint,
  fs,
  emod,
  srcrld,
  dstrld
)
```

Arguments

`start` Start bits:

- `DMA_PRICTL_START_STOP`
- `DMA_PRICTL_START_NORMAL`
- `DMA_PRICTL_START_PAUSE`
- `DMA_PRICTL_START_AUTOINIT`

`srcdir` Source modification:

- `DMA_PRICTL_SRCDIR_NONE`
- `DMA_PRICTL_SRCDIR_INC`
- `DMA_PRICTL_SRCDIR_DEC`
- `DMA_PRICTL_SRCDIR_IDX`

`dstdir` Destination modification:

- `DMA_PRICTL_DSTDIR_NONE`
- `DMA_PRICTL_DSTDIR_INC`
- `DMA_PRICTL_DSTDIR–DEC`
- `DMA_PRICTL_DSTDIR_IDX`

`esize` Element size:

- `DMA_PRICTL_ESIZE_32BIT`
- `DMA_PRICTL_ESIZE_16BIT`
- `DMA_PRICTL_ESIZE_8BIT`

split	Split channel mode: • DMA_PRICTL_SPLIT_DISABLE • DMA_PRICTL_SPLIT_A • DMA_PRICTL_SPLIT_B • DMA_PRICTL_SPLIT_C • DMA_GBL_SPLIT register ID. See DMA_AllocGlobalReg().
cntrld	Count reload: • DMA_PRICTL_CNTRLD_NA • DMA_PRICTL_CNTRLD_A • DMA_PRICTL_CNTRLD_B • DMA_GBL_CNTRLD register ID. See DMA_AllocGlobalReg().
index	Index: • DMA_PRICTL_INDEX_NA • DMA_PRICTL_INDEX_A • DMA_PRICTL_INDEX_B • DMA_GBL_INDEX register ID. See DMA_AllocGlobalReg().

`rsync` Read synchronization:

- `DMA_PRICTL_RSYNC_NONE`
- `DMA_PRICTL_RSYNC_TINT0`
- `DMA_PRICTL_RSYNC_TINT1`
- `DMA_PRICTL_RSYNC_SDINT`
- `DMA_PRICTL_RSYNC_EXTINT4`
- `DMA_PRICTL_RSYNC_EXTINT5`
- `DMA_PRICTL_RSYNC_EXTINT6`
- `DMA_PRICTL_RSYNC_EXTINT7`
- `DMA_PRICTL_RSYNC_DMAINT0`
- `DMA_PRICTL_RSYNC_DMAINT1`
- `DMA_PRICTL_RSYNC_DMAINT2`
- `DMA_PRICTL_RSYNC_DMAINT3`
- `DMA_PRICTL_RSYNC_XEVT0`
- `DMA_PRICTL_RSYNC_REVT0`
- `DMA_PRICTL_RSYNC_XEVT1`
- `DMA_PRICTL_RSYNC_REVT1`
- `DMA_PRICTL_RSYNC_DSPINT`
- `DMA_PRICTL_RSYNC_XEVT2`
- `DMA_PRICTL_RSYNC_REVT2`

wsync	Write synchronization: • DMA_PRICTL_WSYNC_NONE • DMA_PRICTL_WSYNC_TINT0 • DMA_PRICTL_WSYNC_TINT1 • DMA_PRICTL_WSYNC_SDINT • DMA_PRICTL_WSYNC_EXTINT4 • DMA_PRICTL_WSYNC_EXTINT5 • DMA_PRICTL_WSYNC_EXTINT6 • DMA_PRICTL_WSYNC_EXTINT7 • DMA_PRICTL_WSYNC_DMAINT0 • DMA_PRICTL_WSYNC_DMAINT1 • DMA_PRICTL_WSYNC_DMAINT2 • DMA_PRICTL_WSYNC_DMAINT3 • DMA_PRICTL_WSYNC_XEVT0 • DMA_PRICTL_WSYNC_REVT0 • DMA_PRICTL_WSYNC_XEVT1 • DMA_PRICTL_WSYNC_REVT1 • DMA_PRICTL_WSYNC_DSPINT • DMA_PRICTL_WSYNC_XEVT2 • DMA_PRICTL_WSYNC_REVT2
pri	Priority: • DMA_PRICTL_PRI_CPU • DMA_PRICTL_PRI_DMA
tcint	Transfer complete interrupt: • DMA_PRICTL_TCINT_DISABLE • DMA_PRICTL_TCINT_ENABLE
fs	Frame sync: • DMA_PRICTL_FS_DISABLE • DMA_PRICTL_FS_RSYNC
emod	Emulation mode: • DMA_PRICTL_EMOD_NOHALT • DMA_PRICTL_EMOD_HALT

	srcrld	Source reload: • DMA_PRICTL_SRCRLD_NONE • DMA_PRICTL_SRCRLD_B • DMA_PRICTL_SRCRLD_C • DMA_PRICTL_SRCRLD_D • DMA_GBL_ADDRRLD register ID. See DMA_AllocGlobalReg().
	dstrld	Destination reload: • DMA_PRICTL_DSTRLD_NONE • DMA_PRICTL_DSTRLD_B • DMA_PRICTL_DSTRLD_C • DMA_PRICTL_DSTRLD_D • DMA_GBL_ADDRRLD register ID. See DMA_AllocGlobalReg().
Return Value	PRICTL Value	Constructed register value

Description

Use this macro to make a value suitable for a primary control register.

The power-on default value is DMA_PRICTL_DEFAULT.

Use of the *DMA_MK* macros makes it simpler to construct register values based on field values. You have a choice of using integer constants, integer variables, or the symbolic constants for arguments. All field values are right justified.

Refer to the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for descriptions of the arguments.

Example

```
UINT32 PriCtl;

/* you can do this /
PriCtl =
DMA_MK_PRICTL(0,1,1,0,0,0,0,0,0,0,0,0,0,0,0);

/ or to be more readable, you can do this */
PriCtl = DMA_MK_PRICTL(
  DMA_PRICTL_START_STOP,
  DMA_PRICTL_SRCDIR_INC,
  DMA_PRICTL_DSTDIR_INC,
  DMA_PRICTL_ESIZE_32BIT,
  DMA_PRICTL_SPLIT_DISABLE,
  DMA_PRICTL_CNTRLD_NA,
  DMA_PRICTL_INDEX_NA,
  DMA_PRICTL_RSYNC_NONE,
  DMA_PRICTL_WSYNC_NONE,
  DMA_PRICTL_PRI_CPU,
  DMA_PRICTL_TCINT_DISABLE,
  DMA_PRICTL_FS_DISABLE,
  DMA_PRICTL_EMOD_NOHALT,
  DMA_PRICTL_SRCRLD_NONE,
  DMA_PRICTL_DSTRLD_NONE
);
```

4.6.20 DMA_MK_SECCTL

Makes a value suitable for a secondary control register

Macro

```
DMA_MK_SECCTL(
  sxie,
  frameie,
  lastie,
  blockie,
  rdropie,
  wdropie,
  dmacen,
  fsig,
  rspol,
  wspol
)
```

Arguments

`sxie` Split transmit overrun receive interrupt enable:

- `DMA_SECCTL_SXIE_DISABLE`
- `DMA_SECCTL_SXIE_ENABLE`

`frameie`	Frame complete interrupt enable: • `DMA_SECCTL_FRAMEIE_DISABLE` • `DMA_SECCTL_FRAMEIE_ENABLE`
`lastie`	Last frame interrupt enable: • `DMA_SECCTL_LASTIE_DISABLE` • `DMA_SECCTL_LASTIE_ENABLE`
`blockie`	Block transfer complete interrupt enable: • `DMA_SECCTL_BLOCKIE_DISABLE` • `DMA_SECCTL_BLOCKIE_ENABLE`
`rdropie`	Dropped read sync interrupt enable: • `DMA_SECCTL_RDROPIE_DISABLE` • `DMA_SECCTL_RDROPIE_ENABLE`
`wdropie`	Dropped write sync interrupt enable: • `DMA_SECCTL_WDROPIE_DISABLE` • `DMA_SECCTL_WDROPIE_ENABLE`
`dmacen`	DMAC pin control: • `DMA_SECCTL_DMACEN_LOW` • `DMA_SECCTL_DMACEN_HIGH` • `DMA_SECCTL_DMACEN_RSYNCSTAT` • `DMA_SECCTL_DMACEN_WSYNCSTAT` • `DMA_SECCTL_DMACEN_FRAMECOND` • `DMA_SECCTL_DMACEN_BLOCKCOND`
`fsig`	Frame sync ignore: • `DMA_SECCTL_FSIG_NA` • `DMA_SECCTL_FSIG_NORMAL` • `DMA_SECCTL_FSIG_IGNORE`
`rspol`	Read sync polarity: • `DMA_SECCTL_RSPOL_NA` • `DMA_SECCTL_RSPOL_ACTIVEHIGH` • `DMA_SECCTL_RSPOL_ACTIVELOW`

	wspol	Write sync polarity: • DMA_SECCTL_WSPOL_NA • DMA_SECCTL_WSPOL_ACTIVEHIGH • DMA_SECCTL_WSPOL_ACTIVELOW
Return Value	SECCTL Value	Constructed register value

Description

Use this macro to make a value suitable for a secondary control register.

The power-on default value is DMA_SECCTL_DEFAULT.

Use of the *DMA_MK* macros makes it simpler to construct register values based on field values. You have a choice of using integer constants, integer variables, or the symbolic constants for arguments. All field values are right justified.

Refer to the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for descriptions of the arguments.

Example

```
UINT32 SecCtl;

/* you can do this /
SecCtl = DMA_MK_SECCTL(0,0,0,1,0,0,0,0,0,0);

/ or to be more readable, you can do this */
SecCtl = DMA_MK_SECCTL(
  DMA_SECCTL_SXIE_DISABLE,
  DMA_SECCTL_FRAMEIE_DISABLE,
  DMA_SECCTL_LASTIE_DISABLE,
  DMA_SECCTL_BLOCKIE_ENABLE,
  DMA_SECCTL_RDROPIE_DISABLE,
  DMA_SECCTL_WDROPIE_DISABLE,
  DMA_SECCTL_DMACEN_LOW,
  DMA_SECCTL_FSIG_NORMAL,
  DMA_SECCTL_RSPOL_ACTIVEHIGH,
  DMA_SECCTL_WSPOL_ACTIVEHIGH
);
```

4.6.21 DMA_MK_SRC

Makes a value suitable for the source address register

Macro

```
DMA_MK_SRC(
  src
)
```

Arguments

`src`	Source address: • `DMA_SRC_SRC_OF(x)`

Return Value

`SRC Value`	Constructed register value

Description

Use this macro to make a value suitable for the source address register. Although not really necessary, this macro is included for orthogonality and code readability.

The power-on default value is `DMA_SRC_DEFAULT`.

Use of the *DMA_MK* macros makes it simpler to construct register values based on field values. You have a choice of using integer constants, integer variables, or the symbolic constants for arguments. All field values are right justified.

Refer to the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for descriptions of the arguments.

Example

```
UINT32 Src;

Src = DMA_MK_SRC(0x80000000);

Src = DMA_MK_SRC(
 DMA_SRC_SRC_OF(0x80000000)
);
```

4.6.22 DMA_MK_XFRCNT

Makes a value suitable for a transfer count register

Macro

```
DMA_MK_XFRCNT(
  elecnt,
  frmcnt
)
```

Arguments

`elecnt` Element count:

- `DMA_XFRCNT_ELECNT_OF(x)`

`frmcnt` Frame count:

- `DMA_GBLCNT_FRMCNT_OF(x)`

Return Value

`XFRCNT Value` Constructed register value

Description

Use this macro to make a value suitable for a transfer count register.

The power-on default value is `DMA_XFRCNT_DEFAULT`.

Use of the *DMA_MK* macros makes it simpler to construct register values based on field values. You have a choice of using integer constants, integer variables, or the symbolic constants for arguments. All field values are right justified.

Refer to the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for descriptions of the arguments.

Example

```
UINT32 XfrCnt;

XfrCnt = DMA_MK_XFRCNT(0x0100,0x0020);

XfrCnt = DMA_MK_XFRCNT(
  DMA_XFRCNT_ELECNT_OF(0x0100),
  DMA_XFRCNT_FRMCNT_OF(0x0020)
);
```

4.6.23 DMA_Open

Opens a DMA channel for use

Function

```
DMA_HANDLE DMA_Open(
  int ChaNum,
  UINT32 Flags
);
```

Arguments

`ChaNum`	DMA channel to open: • `DMA_CHAANY` • `DMA_CHA0` • `DMA_CHA1` • `DMA_CHA2` • `DMA_CHA3`
`Flags`	Open flags (logical OR of any of the following): • `DMA_OPEN_RESET`

Return Value

`Device Handle`	Handle to newly opened device

Description

Before a DMA channel can be used, it must first be opened by this function. Once opened, it cannot be opened again until closed. See `DMA_Close()`. You have the option of either specifying exactly which physical channel to open or you can let the library pick an unused one for you by specifying `DMA_CHAANY`. The return value is a unique device handle that you use in subsequent DMA API calls. If the open fails, `INV` is returned.

If the `DMA_OPEN_RESET` is specified, the DMA channel registers are set to their power-on defaults and the channel interrupt is disabled and cleared.

Example

```
DMA_HANDLE hDma;
...
hDma = DMA_Open(DMA_CHAANY,DMA_OPEN_RESET);
```

4.6.24 DMA_Pause

Pauses the DMA channel by setting the START bits in the primary control register accordingly

Function

```
void DMA_Pause(
  DMA_HANDLE hDma
);
```

Arguments `hDma` Handle to DMA channel. See `DMA_Open()`

Return Value `none`

Description This function pauses the DMA channel by setting the START bits in the primary control register accordingly. See also `DMA_Start()`, `DMA_Stop()`, and `DMA_AutoStart()`.

Example

```
DMA_Pause(hDma);
```

4.6.25 DMA_Reset

Resets the DMA channel by setting its registers to power-on defaults

Function

```
void DMA_Reset(
  DMA_HANDLE hDma
);
```

Arguments `hDma` Handle to DMA channel. See `DMA_Open()`

Return Value `none`

Description Resets the DMA channel by setting its registers to power-on defaults and disabling and clearing the channel interrupt. You may use `INV` as the device handle to reset all channels.

Example

```
/* reset an open DMA channel /
DMA_Reset(hDma);

/ reset all DMA channels */
DMA_Reset(INV);
```

4.6.26 DMA_SetAuxCtl

Sets the DMA AUXCTL register

Function

```
void DMA_SetAuxCtl(
  UINT32 AuxCtl
);
```

Arguments

`AuxCtl`	Value to set AUXCTL register to

Return Value

`none`

Description

This function sets the DMA AUXCTL register. You may use the `DMA_MK_AUXCTL` macro to construct the register value based on field values. The default value for this register is `DMA_AUXCTL_DEFAULT`.

Example

```
DMA_SetAuxCtl(0x00000000);
```

4.6.27 DMA_SetGlobalReg

Sets value of a global DMA register previously allocated by calling `DMA_AllocGlobalReg()`

Function

```
void DMA_SetGlobalReg(
  UINT32 RegId,
  UINT32 Val
);
```

Arguments

`RegId`	Global register ID obtained from `DMA_AllocGlobalReg()`.
`Val`	Value to set register to

Return Value

`none`

Description

This function sets the value of a global DMA register that was previously allocated by calling `DMA_AllocGlobalReg()`.

Example

```
UINT32 RegId;
 …
/* allocate global index register and initialize it
/
RegId = DMA_AllocGlobalReg(DMA_GBL_IN-
DEX,0x00200040);
…
 DMA_SetGlobalReg(RegId,0x12345678);
```

4.6.28 DMA_Start

Starts a DMA channel running without autoinitialization

Function

```
void DMA_Start(
  DMA_HANDLE hDma
);
```

Arguments `hDma` Handle to DMA channel, see `DMA_Open()`

Return Value `none`

Description Starts a DMA channel running without autoinitialization by setting the START bits in the primary control register accordingly. See also `DMA_Pause()`, `DMA_Stop()`, and `DMA_AutoStart()`.

Example

```
DMA_Start(hDma);
```

4.6.29 DMA_Stop

Stops a DMA channel by setting the START bits in the primary control register accordingly

Function

```
void DMA_Stop(
  DMA_HANDLE hDma
);
```

Arguments `hDma` Handle to DMA channel. See `DMA_Open()`

Return Value `none`

Description Stops a DMA channel by setting the START bits in the primary control register accordingly. See also `DMA_Pause()`, `DMA_Start()`, and `DMA_AutoStart()`.

Example

```
DMA_Stop(hDma);
```

4.6.30 DMA_SUPPORT

A compile time constant whose value is 1 if the device supports the DMA module

Constant `DMA_SUPPORT`

Description Compile time constant that has a value of 1 if the device supports the DMA module and 0 otherwise. You are not required to use this constant.

Note: The DMA module is not supported on devices that do not have the DMA peripheral. In these cases, the EDMA module is supported instead.

Example

```
#if (DMA_SUPPORT)
  /* user DMA configuration /
#elif (EDMA_SUPPORT)
  / user EDMA configuration */
#endif
```

4.6.31 DMA_Wait

Enters a spin loop that polls the DMA status bits until the DMA completes

Function

```
void DMA_Wait(
  DMA_HANDLE hDma
);
```

Arguments

`hDma` Handle to DMA channel. See `DMA_Open()`

Return Value

`none`

Description

This function enters a spin loop that polls the DMA status bits until the DMA completes. Interrupts are not disabled during this loop. This function is equivalent to the following line of code:

```
while (DMA_Status(hDma)&DMA_STATUS_RUNNING);
```

Example

```
DMA_Wait(hDma);
```

4.7 EDMA

4.7.1 EDMA_AllocTable

Allocates a parameter RAM table from PRAM

Function

```
EDMA_HANDLE EDMA_AllocTable(
  int TableNum
);
```

Arguments

`TableNum`	Table number to allocate. Valid values are 0 to `DMA_TABLE_CNT`–1; –1 for any.

Return Value

`Device Handle`	Returns a device handle

Description

This function allocates a parameter RAM table from PRAM. You use PRAM tables for linking transfers together. You can either specify a table number or specify –1 and the function will pick an unused one for you. The return value is a device handle and may be used for APIs that require a device handle. If the table could not be allocated, then `EDMA_HINV` is returned.

If you finish with the table and wish to free it up again, call `EDMA_FreeTable()`.

Example

```
EDMA_HANDLE hEdmaTable;
...
hEdmaTabel = EDMA_AllocTable(–1);
```

4.7.2 EDMA_CHA_CNT

Number of EDMA channels

Constant

`EDMA_CHA_CNT`

Description

Compile time constant that holds the number of EDMA channels.

4.7.3 EDMA_ClearChannel

Clears the EDMA event flag in the EDMA channel event register

Function
```
void EDMA_ClearChannel(
  EDMA_HANDLE hEdma
);
```

Arguments `hEdma` Device handle, see `EDMA_Open()`.

Return Value `none`

Description This function clears the EDMA event flag in the EDMA channel event register by writing to the appropriate bit in the EDMA event clear register (ECR).

This function accepts the following devices handles:

- From `EDMA_Open()`

Example
```
EDMA_ClearChannel(hEdma);
```

4.7.4 EDMA_Close

Closes a previously opened EDMA channel

Function
```
void EDMA_Close(
  EDMA_HANDLE hEdma
);
```

Arguments `hEdma` Device handle. See `EDMA_Open()`.

Return Value `none`

Description Closes a previously opened EDMA channel.

This function accepts the following devices handles:

- From `EDMA_Open()`

Example
```
EDMA_Close(hEdma);
```

4.7.5 EDMA_CONFIG

The EDMA configuration structure used to set up an EDMA channel

Structure `EDMA_CONFIG`

Members

`UINT32 opt`	Options
`UINT32 src`	Source address
`UINT32 cnt`	Transfer count
`UINT32 dst`	Destination address
`UINT32 idx`	Index
`UINT32 rld`	Element count reload and link address

Description This is the EDMA configuration structure used to set up an EDMA channel. You create and initialize this structure and then pass its address to the `EDMA_ConfigA()` function. You can use literal values or the *EDMA_MK* macros to create the structure member values.

Example

```
EDMA_CONFIG MyConfig = {
  0x41200000, /* opt */
  0x80000000, /* src */
  0x00000040, /* cnt */
  0x80010000, /* dst */
  0x00000004, /* idx */
  0x00000000  /* rld */
};
…
EDMA_ConfigA(hEdma,&MyConfig);
```

4.7.6 EDMA_ConfigA

Sets up the EDMA channel using the configuration structure

Function

```
void EDMA_ConfigA(
  EDMA_HANDLE hDma,
  EDMA_CONFIG *Config
);
```

Arguments

`hEdma`	Device handle. See `EDMA_Open()` and `EDMA_AllocTable()`.
`Config`	Pointer to an initialized configuration structure

Return Value

`none`

Description

Sets up the EDMA channel using the configuration structure. The values of the structure are written to the EDMA PRAM entries. The options value (*opt*) is written last. See also `EDMA_ConfigB()` and `EDMA_CONFIG`.

Note: The predefined device handle `EDMA_HQDMA` may be used as the *hEdma* argument if you wish to configure the quick DMA registers. In this case, the *rld* structure member is ignored. First, *src*, *cnt*, *dst*, and *idx* structure members are written to the QDMA SRC, CNT, DST, and IDX registers, respectively. Then the *opt* structure member is written to the pseudo-QDMA OPT register.

This function accepts the following devices handles:

- From `EDMA_Open()`
- From `EDMA_AllocTable()`
- Quick DMA predefined handle

Example

```
EDMA_CONFIG MyConfig = {
  0x41200000, /* opt */
  0x80000000, /* src */
  0x00000040, /* cnt */
  0x80010000, /* dst */
  0x00000004  /* idx */
  0x00000000  /* rld */
};
...
EDMA_ConfigA(hEdma,&MyConfig);
```

4.7.7 EDMA_ConfigB

Sets up the EDMA channel using the EDMA parameter arguments

Function

```
void EDMA_ConfigB(
  EDMA_HANDLE hEdma,
  UINT32 opt,
  UINT32 src,
  UINT32 cnt,
  UINT32 dst,
  UINT32 idx,
  UINT32 rld
);
```

Arguments

`hEdma`	Device handle. See `EDMA_Open()` and `EDMA_AllocTable()`.
`opt`	Options
`src`	Source address
`cnt`	Transfer count
`dst`	Destination address
`idx`	Index
`rld`	Element count reload and link address

Return Value

`none`

Description

Sets up the EDMA channel using the EDMA parameter arguments. The values of the arguments are written to the EDMA PRAM entries. The options value (*opt*) is written last. See also `EDMA_ConfigA()`.

Note: The predefined device handle `EDMA_HQDMA` may be used as the *hEdma* argument if you wish to configure the quick DMA registers. In this case, the *rld* argument is ignored. First, *src*, *cnt*, *dst*, and *idx* are written to the QDMA SRC, CNT, DST, and IDX registers respectively. Then *opt* is written to the pseudo-QDMA OPT register.

This function accepts the following devices handles:

- From `EDMA_Open()`
- From `EDMA_AllocTable()`
- Quick DMA predefined handle

Example

```
EDMA_ConfigB(hEdma,
  0x41200000, /* opt */
  0x80000000, /* src */
  0x00000040, /* cnt */
  0x80010000, /* dst */
  0x00000004  /* idx */
  0x00000000  /* rld */
);
```

4.7.8 EDMA_DisableChannel

Disables an EDMA channel

Function

```
void EDMA_DisableChannel(
  EDMA_HANDLE hEdma
);
```

Arguments `hEdma` Device handle, see `EDMA_Open()`.

Return Value `none`

Description Disables an EDMA channel by clearing the corresponding bit in the EDMA event enable register. See also `EDMA_EnableChannel()`.

This function accepts the following devices handles:

- From `EDMA_Open()`

Example

```
EDMA_DisableChannel(hEdma);
```

4.7.9 EDMA_EnableChannel

Enables an EDMA channel

Function

```
void EDMA_EnableChannel(
  EDMA_HANDLE hEdma
);
```

Arguments `hEdma` Device handle, see `EDMA_Open()`.

Return Value `none`

Description Enables an EDMA channel by setting the corresponding bit in the EDMA event enable register. See also `EDMA_DisableChannel()`. When you open an EDMA channel it is disabled so you must enabled it explicitly.

This function accepts the following devices handles:

- From `EDMA_Open()`

Example

```
EDMA_EnableChannel(hEdma);
```

4.7.10 EDMA_FreeTable

Frees up a PRAM table previously allocated

Function

```
void EDMA_FreeTable(
  EDMA_HANDLE hEdma
);
```

Arguments

hEdma	Device handle. See EDMA_AllocTable().

Return Value

none

Description

This function frees up a PRAM table previously allocated via EDMA_AllocTable().

This function accepts the following device handles:

- From EDMA_AllocTable()

Example

```
EDMA_FreeTable(hEdmaTable);
```

4.7.11 EDMA_GetChannel

Returns the current state of the channel event

Function

```
UINT32 EDMA_GetChannel(
  EDMA_HANDLE hEdma
);
```

Arguments

hEdma	Device handle. See EDMA_Open().

Return Value

Channel Flag	Channel event flag: 0 – event not detected 1 – event detected

Description

Returns the current state of the channel event by reading the event flag from the EDMA channel event register (EER).

This function accepts the following devices handles:

- From EDMA_Open()

Example

```
flag = EDMA_GetChannel(hEdma);
```

4.7.12 EDMA_GetPriQStatus

Returns the value of the priority queue status register (PQSR)

Function `UINT32 EDMA_GetPriQStatus();`

Arguments `none`

Return Value `Status` Returns status of the priority queue

Description Returns the value of the priority queue status register (PQSR). May be the logical OR of any of the following:

- 0x00000001 – PQ0
- 0x00000002 – PQ1
- 0x00000004 – PQ2

Example
```
PqStat = EDMA_GetPriQStatus();
```

4.7.13 EDMA_GetScratchAddr

Returns the starting address of the EDMA PRAM used as non-cacheable on-chip SRAM (scratch area)

Function `UINT32 EDMA_GetScratchAddr();`

Arguments `none`

Return Value `Scratch Address` 32-bit starting address of PRAM scratch area

Description There is a small portion of the EDMA PRAM that is not used for parameter tables and is free for use as non-cacheable on-chip SRAM. This function returns the starting address of this scratch area. See also `EDMA_GetScratchSize()`.

Example
```
UINT32 *ScratchWord;
ScratchWord = (UINT32*)EDMA_GetScratchAddr();
```

4.7.14 EDMA_GetScratchSize

Returns the size (in bytes) of the EDMA PRAM used as non-cacheable on-chip SRAM (scratch area)

Function `UINT32 EDMA_GetScratchSize();`

Arguments `none`

Return Value `Scratch Size` Size of PRAM scratch area in bytes

Description There is a small portion of the EDMA PRAM that is not used for parameter tables and is free for use as non-cacheable on-chip SRAM. This function returns the size of this scratch area in bytes. See also `EDMA_GetScratchAddr()`.

Example
```
ScratchSize = EDMA_GetScratchSize();
```

4.7.15 EDMA_GetTableAddress

Returns the 32-bit absolute address of the table

Function

```
UINT32 GetTableAddress(
  EDMA_HANDLE hEdma
);
```

Arguments

`hEdma`	Device handle obtained by `EDMA_AllocTable()`.

Return Value

`Table Address`	32-bit address of table

Description

Given a device handle obtained from `EDMA_AllocTable()`, this function returns the 32-bit absolute address of the table.

This function accepts the following device handles:

- From `EDMA_AllocTable()`

Example

```
Addr = EDMA_GetTableAddress(hEdmaTable);
```

4.7.16 EDMA_MK_CNT

Makes a value suitable for the EDMA CNT parameter

Macro

```
EDMA_MK_CNT(
  elecnt,
  frmcnt
)
```

Arguments

`elecnt`	Element count: • `EDMA_CNT_ELECNT_OF(x)`
`frmcnt`	Frame count: • `EDMA_CNT_FRMCNT_OF(x)`

Return Value

`CNT Value`	Constructed parameter value

Description

Use this macro to make a value suitable for the EDMA CNT parameter.

The default CNT parameter value is `EDMA_CNT_DEFAULT`.

Use of the *EDMA_MK* macros makes it simpler to construct parameter values based on field values. You have a choice of using integer constants, integer variables, or the symbolic constants for arguments. All field values are right justified.

Refer to the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for descriptions of the arguments.

Example

```
UINT32 Cnt;

Cnt = DMA_MK_CNT(0x0100,0x0020);

Cnt = DMA_MK_CNT(
  DMA_CNT_ELECNT_OF(0x0100),
  DMA_CNT_FRMCNT_OF(0x0020)
);
```

4.7.17 EDMA_MK_DST

Makes a value suitable for the EDMA DST parameter

Macro

```
EDMA_MK_DST(
  dst
)
```

Arguments

`dst`	Destination address: • `EDMA_DST_DST_OF(x)`

Return Value

`DST Value`	Constructed parameter value

Description

Use this macro to make a value suitable for the EDMA DST parameter. Although not really necessary, this macro is included for orthogonality and code readability.

The default DST parameter value is `EDMA_DST_DEFAULT`.

Use of the *EDMA_MK* macros makes it simpler to construct parameter values based on field values. You have a choice of using integer constants, integer variables, or the symbolic constants for arguments. All field values are right justified.

Refer to the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for descriptions of the arguments.

Example

```
UINT32 Dst;

Dst = DMA_MK_DST(0x80000000);

Dst = DMA_MK_DST(
  EDMA_DST_DST_OF(0x80000000)
);
```

4.7.18 EDMA_MK_IDX

Makes a value suitable for the EDMA IDX parameter

Macro

```
EDMA_MK_IDX(
  eleidx,
  frmidx
)
```

Arguments

`eleidx` Element index:

- `EDMA_IDX_ELEIDX_OF(x)`

`frmidx` Frame index:

- `EDMA_IDX_FRMIDX_OF(x)`

Return Value

`IDX Value` Constructed parameter value

Description

Use this macro to make a value suitable for the EDMA IDX parameter.

The default IDX parameter value is `EDMA_IDX_DEFAULT`.

Use of the *EDMA_MK* macros makes it simpler to construct parameter values based on field values. You have a choice of using integer constants, integer variables, or the symbolic constants for arguments. All field values are right justified.

Refer to the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for descriptions of the arguments.

Example

```
UINT32 Idx;

Idx = DMA_MK_IDX(0x0100,0x0020);

Idx = DMA_MK_IDX(
  DMA_IDX_ELEIDX_OF(0x0100),
  DMA_IDX_FRMIDX_OF(0x0020)
);
```

4.7.19 EDMA_MK_OPT

Makes a value suitable for the EDMA OPT parameter

Macro

```
EDMA_MK_OPT(
  fs,
  link,
  tcc,
  tcint,
  dum,
  d2d,
  sum,
  s2d,
  esize,
  pri
)
```

Arguments

`fs` Frame sync:

- `EDMA_OPT_FS_NO`
- `EDMA_OPT_FS_YES`

`link` Link flag:

- `EDMA_OPT_LINK_NO`
- `EDMA_OPT_LINK_YES`

`tcc` Transfer complete code:

- `EDMA_OPT_TCC_OF(x)`

`tcint` Transfer complete interrupt:

- `EDMA_OPT_TCINT_NO`
- `EDMA_OPT_TCINT_YES`

`dum` Destination address update mode:

- `EDMA_OPT_DUM_NONE`
- `EDMA_OPT_DUM_INC`
- `EDMA_OPT_DUM_DEC`
- `EDMA_OPT_DUM_IDX`

`d2d` 2-Dimensional destination:

- `EDMA_OPT_2DD_NO`
- `EDMA_OPT_2DD_YES`

	`sum`	Source address update mode: • `EDMA_OPT_SUM_NONE` • `EDMA_OPT_SUM_INC` • `EDMA_OPT_SUM_DEC` • `EDMA_OPT_SUM_IDX`
	`s2d`	2-Dimensional source: • `EDMA_OPT_2DS_NO` • `EDMA_OPT_2DS_YES`
	`esize`	Element size: • `EDMA_OPT_ESIZE_32BIT` • `EDMA_OPT_ESIZE_16BIT` • `EDMA_OPT_ESIZE_8BIT`
	`pri`	Priority: • `EDMA_OPT_PRI_HIGH` • `EDMA_OPT_PRI_LOW`
Return Value	`OPT Value`	Constructed parameter value

Description

Use this macro to make a value suitable for the EDMA OPT parameter.

The default OPT parameter value is `EDMA_OPT_DEFAULT`.

Use of the *EDMA_MK* macros makes it simpler to construct parameter values based on field values. You have a choice of using integer constants, integer variables, or the symbolic constants for arguments. All field values are right justified.

Refer to the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for descriptions of the arguments.

Example

```
UINT32 Opt;

/* you can do this /
Opt = EDMA_MK_OPT(2,0,0,0,1,0,1,0,0,0);

/ or to be more readable, you can do this */
Opt = EDMA_MK_OPT(
  EDMA_OPT_FS_NO,
  EDMA_OPT_LINK_NO,
  EDMA_OPT_TCC_OF(0),
  EDMA_OPT_TCINT_NO,
  EDMA_OPT_DUM_INC,
  EDMA_OPT_2DD_NO,
  EDMA_OPT_SUM_INC,
  EDMA_OPT_2DS_NO,
  EDMA_OPT_ESIZE_32BIT,
  EDMA_OPT_PRI_LOW
);
```

4.7.20 EDMA_MK_RLD

Makes a value suitable for the EDMA RLD parameter

Macro

```
EDMA_MK_RLD(
  link,
  elerld
)
```

Arguments

`link` Link address:

- `EDMA_RLD_LINK_OF(x)`

`elerld` Element count reload:

- `EDMA_RLD_ELERLD_OF(x)`

Return Value

`RLD Value` Constructed parameter value

Description

Use this macro to make a value suitable for the EDMA RLD parameter. You may directly use the device handle returned by `EDMA_AllocTable()` as the *link* argument.

The default RLD parameter value is `EDMA_RLD_DEFAULT`.

Use of the *EDMA_MK* macros makes it simpler to construct parameter values based on field values. You have a choice of using integer constants, integer variables, or the symbolic constants for arguments. All field values are right justified.

Refer to the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for descriptions of the arguments.

Example

```
UINT32 Rld;
EDMA_HANDLE hEdmaTable;

hEdmaTable = EDMA_AllocTable(–1);

Rld = DMA_MK_RLD(hEdmaTable,0x0020);

Rld = DMA_MK_RLD(
  EDMA_RLD_LINK_OF(hEdmaTable),
  EDMA_RLD_ELERLD_OF(0x0020)
);
```

4.7.21 EDMA_MK_SRC

Makes a value suitable for the EDMA SRC parameter

Macro

```
EDMA_MK_SRC(
  src
)
```

Arguments

`src` Source address:

- `EDMA_SRC_SRC_OF(x)`

Return Value

`SRC Value` Constructed parameter value

Description

Use this macro to make a value suitable for the EDMA SRC parameter. Although not really necessary, this macro is included for orthogonality and code readability.

The default SRC parameter value is `EDMA_SRC_DEFAULT`.

Use of the *EDMA_MK* macros makes it simpler to construct parameter values based on field values. You have a choice of using integer constants, integer variables, or the symbolic constants for arguments. All field values are right justified.

Refer to the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for descriptions of the arguments.

Example

```
UINT32 Src;

Src = DMA_MK_SRC(0x80000000);

Src = DMA_MK_SRC(
  EDMA_SRC_SRC_OF(0x80000000)
);
```

4.7.22 EDMA_Open

Opens an EDMA channel

Function

```
EDMA_HANDLE EDMA_Open(
  int ChaNum,
  UINT32 Flags
);
```

Arguments

`ChaNum` — EDMA channel to open:

- `EDMA_CHA_ANY`
- `EDMA_CHA_DSPINT`
- `EDMA_CHA_TINT0`
- `EDMA_CHA_TINT1`
- `EDMA_CHA_SDINT`
- `EDMA_CHA_EXTINT4`
- `EDMA_CHA_EXTINT5`
- `EDMA_CHA_EXTINT6`
- `EDMA_CHA_EXTINT7`
- `EDMA_CHA_TCC8`
- `EDMA_CHA_TCC9`
- `EDMA_CHA_TCC10`
- `EDMA_CHA_TCC11`
- `EDMA_CHA_XEVT0`
- `EDMA_CHA_REVT0`
- `EDMA_CHA_XEVT1`
- `EDMA_CHA_REVT1`

`Flags` — Open flags, logical OR of any of the following:

- `EDMA_OPEN_RESET`
- `EDMA_OPEN_ENABLE`

Return Value

`Device Handle` — Device handle to be used by other EDMA API function calls.

Description Before an EDMA channel can be used, it must first be opened by this function. Once opened, it cannot be opened again until closed. See `EDMA_Close()`. You have the option of either specifying exactly which physical channel to open or you can let the library pick an unused one for you by specifying `EDMA_CHA_ANY`. The return value is a unique device handle that you use in subsequent EDMA API calls. If the open fails, `EDMA_HINV` is returned.

If the `EDMA_OPEN_RESET` is specified, the EDMA channel is reset, the channel interrupt is disabled and cleared. If the EDMA_OPEN_ENABLE flag is specified, the channel will be enabled.

If the channel cannot be opened, `EDMA_HINV` is returned.

Refer to the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for details regarding the EDMA channels.

- `EDMA_HQDMA`

Example

```
EDMA_HANDLE hEdma;
...
hEdma = EDMA_Open(EDMA_CHA_TINT0,EDMA_OPEN_RESET);
...
```

4.7.23 EDMA_Reset

Resets the given EDMA channel

Function

```
void EDMA_Reset(
  EDMA_HANDLE hEdma
);
```

Arguments `hEdma` Device handle obtained by `EDMA_Open()`.

Return Value `none`

Description Resets the given EDMA channel. If you pass `EDMA_HINV` or `INV` as the device handle, all channels are reset.

The following steps are taken:

- The channel is disabled
- The channel event flag is cleared

This function accepts the following devices handles:

- From `EDMA_Open()`

Example

```
EDMA_Reset(hEdma);
EDMA_Reset(INV);
```

4.7.24 EDMA_SetChannel

Triggers an EDMA channel by writing to appropriate bit in the event set register (ESR)

Function

```
void EDMA_SetChannel(
  EDMA_HANDLE hEdma
);
```

Arguments `hEdma` Device handle obtained by `EDMA_Open()`.

Return Value `none`

Description Software triggers an EDMA channel by writing to the appropriate bit in the EDMA event set register (ESR).

This function accepts the following device handles:

- From `EDMA_Open()`

Example

```
EDMA_SetChannel(hEdma);
```

4.7.25 EDMA_SUPPORT

A compile time constant whose value is 1 if the device supports the EDMA module

Constant `EDMA_SUPPORT`

Description Compile time constant that has a value of 1 if the device supports the EDMA module and 0 otherwise. You are not required to use this constant.

Note: The EDMA module is not supported on devices that do not have the EDMA peripheral. In these cases, the DMA module is supported instead.

Example

```
#if (EDMA_SUPPORT)
  /* user EDMA configuration /
#elif (DMA_SUPPORT)
  / user DMA configuration */
#endif
```

4.7.26 EDMA_TABLE_CNT

A compile time constant that holds the total number of parameter table entries in the EDMA PRAM

Constant `EDMA_TABLE_CNT`

Description Compile time constant that holds the total number of parameter table entries in the EDMA PRAM.

4.8 EMIF

4.8.1 EMIF_CONFIG

Structure used to set up the EMIF peripheral

Structure `EMIF_CONFIG`

Members

`UINT32 gblctl`	EMIF global control register value
`UINT32 ce0ctl`	CE0 space control register value
`UINT32 ce1ctl`	CE1 space control register value
`UINT32 ce2ctl`	CE2 space control register value
`UINT32 ce3ctl`	CE3 space control register value
`UINT32 sdctl`	SDRAM control register value
`UINT32 sdtim`	SDRAM timing register value
`UINT32 sdext`	SDRAM extension register value

Description This is the EMIF configuration structure used to setup the EMIF peripheral. You create and initialize this structure and then pass its address to the `EMIF_ConfigA()` function. You can use literal values or the *EMIF_MK* macros to create the structure member values.

Example

```
EMIF_CONFIG MyConfig = {
  0x00003060, /* gblctl */
  0x00000040, /* ce0ctl */
  0x404F0323, /* ce1ctl */
  0x00000030, /* ce2ctl */
  0x00000030, /* ce3ctl */
  0x72270000, /* sdctl  */
  0x00000410, /* sdtim  */
  0x00000000  /* sdext  */
};
...
EMIF_ConfigA(&MyConfig);
```

4.8.2 EMIF_ConfigA

Sets up the EMIF using the configuration structure

Function

```
void EMIF_ConfigA(
  EMIF_CONFIG *Config
);
```

Arguments `Config` Pointer to an initialized configuration structure

Return Value `none`

Description Sets up the EMIF using the configuration structure. The values of the structure are written to the EMIF registers. See also `EMIF_ConfigB()` and `EMIF_CONFIG`.

Example

```
EMIF_CONFIG MyConfig = {
  0x00003060, /* gblctl */
  0x00000040, /* ce0ctl */
  0x404F0323, /* ce1ctl */
  0x00000030, /* ce2ctl */
  0x00000030, /* ce3ctl */
  0x72270000, /* sdctl  */
  0x00000410, /* sdtim  */
  0x00000000  /* sdext  */
};
…
EMIF_ConfigA(&MyConfig);
```

4.8.3 EMIF_ConfigB

Sets up the EMIF using the register value arguments

Function

```
void EMIF_ConfigB(
  UINT32 gblctl,
  UINT32 ce0ctl,
  UINT32 ce1ctl,
  UINT32 ce2ctl,
  UINT32 ce3ctl,
  UINT32 sdctl,
  UINT32 sdtim,
  UINT32 sdext
);
```

Arguments

`gblctl`	EMIF global control register value
`ce0ctl`	CE0 space control register value
`ce1ctl`	CE1 space control register value
`ce2ctl`	CE2 space control register value
`ce3ctl`	CE3 space control register value
`sdctl`	SDRAM control register value
`sdtim`	SDRAM timing register value
`sdext`	SDRAM extension register value

Return Value `none`

Description Sets up the EMIF using the register value arguments. The arguments are written to the EMIF registers. See also `EMIF_ConfigA()`.

Example

```
EMIF_ConfigB(
  0x00003060, /* gblctl */
  0x00000040, /* ce0ctl */
  0x404F0323, /* ce1ctl */
  0x00000030, /* ce2ctl */
  0x00000030, /* ce3ctl */
  0x72270000, /* sdctl  */
  0x00000410, /* sdtim  */
  0x00000000  /* sdext  */
);
```

4.8.4 EMIF_MK_CECTL

Makes a value suitable for an EMIF CE space control register

Macro

```
EMIF_MK_CECTL(
  rdhld,
  mtype,
  rdstrb,
  ta,
  rdsetup,
  wrhld,
  wrstrb,
  wrsetup
)
```

Arguments

`rdhld` Read hold:

- `EMIF_CECTL_RDHLD_OF(x)`

`mtype` Memory type:

- `EMIF_CECTL_MTYPE_ASYNC8`
- `EMIF_CECTL_MTYPE_ASYNC16`
- `EMIF_CECTL_MTYPE_ASYNC32`
- `EMIF_CECTL_MTYPE_SDRAM32`
- `EMIF_CECTL_MTYPE_SBSRAM32`
- `EMIF_CECTL_MTYPE_SDRAM8`
- `EMIF_CECTL_MTYPE_SDRAM16`
- `EMIF_CECTL_MTYPE_SBSRAM8`
- `EMIF_CECTL_MTYPE_SBSRAM16`

`rdstrb` Read strobe:

- `EMIF_CECTL_RDSTRB_OF(x)`

	`ta`	Turn around time: • `EMIF_CECTL_TA_NA` • `EMIF_CECTL_TA_OF(x)`
	`rdsetup`	Read setup: • `EMIF_CECTL_RDSETUP_OF(x)`
	`wrhld`	Write hold: • `EMIF_CECTL_WRHLD_OF(x)`
	`wrstrb`	Write strobe: • `EMIF_CECTL_WRSTRB_OF(x)`
	`wrsetup`	Write setup: • `EMIF_CECTL_WRSETUP_OF(x)`
Return Value	`CECTL Value`	Constructed register value

Description

Use this macro to make a value suitable for an EMIF CE space control register.

The default CE space control register value is `EMIF_CECTL_DEFAULT`.

Use of the *EMIF_MK* macros makes it simpler to construct register values based on field values. You have a choice of using integer constants, integer variables, or the symbolic constants for arguments. All field values are right justified.

Refer to the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for descriptions of the arguments.

Example

```
UINT32 CeCtl;

/* you can do this /
CeCtl = EMIF_MK_CECTL(3,0,0,0,0,0,0,0);

/ or to be more readable, you can do this */
CeCtl = EMIF_MK_CECTL(
  EMIF_CECTL_RDHLD_OF(0),
  EMIF_CECTL_MTYPE_SDRAM32,
  EMIF_CECTL_RDSTRB_OF(0),
  EMIF_CECTL_TA_NA,
  EMIF_CECTL_RDSETUP_OF(0),
  EMIF_CECTL_WRHLD_OF(0),
  EMIF_CECTL_WRSTRB_OF(0),
  EMIF_CECTL_WRSETUP_OF(0)
);
```

4.8.5 EMIF_MK_GBLCTL

Makes a value suitable for the EMIF global control register

Macro

```
EMIF_MK_GBLCTL(
  rbtr8,
  sscrt,
  clk2en,
  clk1en,
  sscen,
  sdcen,
  nohold
)
```

Arguments

`rbtr8` Requester arbitration mode:

- `EMIF_GBLCTL_RBTR8_NA`
- `EMIF_GBLCTL_RBTR8_HPRI`
- `EMIF_GBLCTL_RBTR8_8ACC`

`sscrt` SBSRAM clock rate select:

- `EMIF_GBLCTL_SSCRT_NA`
- `EMIF_GBLCTL_SSCRT_CPUOVR2`
- `EMIF_GBLCTL_SSCRT_CPU`

`clk2en` CLKOUT2 enable:

- `EMIF_GBLCTL_CLK2EN_NA`
- `EMIF_GBLCTL_CLK2EN_DISABLE`
- `EMIF_GBLCTL_CLK2EN_ENABLE`

`clk1en` CLKOUT1 enable:

- `EMIF_GBLCTL_CLK1EN_DISABLE`
- `EMIF_GBLCTL_CLK1EN_ENABLE`

`sscen` SSCLK enable:

- `EMIF_GBLCTL_SSCEN_NA`
- `EMIF_GBLCTL_SSCEN_DISABLE`
- `EMIF_GBLCTL_SSCEN_ENABLE`

`sdcen` SDCLK enable:

- `EMIF_GBLCTL_SDCEN_NA`
- `EMIF_GBLCTL_SDCEN_DISABLE`
- `EMIF_GBLCTL_SDCEN_ENABLE`

	nohold	External hold disable: • EMIF_GBLCTL_NOHOLD_0 • EMIF_GBLCTL_NOHOLD_1
Return Value	GBLCTL Value	Constructed register value

Description

Use this macro to make a value suitable for the EMIF global control register.

The default global control register value is EMIF_GBLCTL_DEFAULT.

Use of the *EMIF_MK* macros makes it simpler to construct register values based on field values. You have a choice of using integer constants, integer variables, or the symbolic constants for arguments. All field values are right justified.

Refer to the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for descriptions of the arguments.

Example

```
UINT32 GblCtl;

/* you can do this /
GblCtl = EMIF_MK_GBLCTL(0,0,1,1,1,0,0);

/ or to be more readable, you can do this */
GblCtl = EMIF_MK_GBLCTL(
  EMIF_GBLCTL_RBTR8_HPRI,
  EMIF_GBLCTL_SSCRT_CPUOVR2,
  EMIF_GBLCTL_CLK2EN_ENABLE,
  EMIF_GBLCTL_CLK1EN_ENABLE,
  EMIF_GBLCTL_SSCEN_ENABLE,
  EMIF_GBLCTL_SDCEN_DISABLE,
  EMIF_GBLCTL_NOHOLD_0
);
```

4.8.6 EMIF_MK_SDCTL

Makes a value suitable for the EMIF SDRAM control register

Macro

```
EMIF_MK_SDCTL(
  trc,
  trp,
  trcd,
  init,
  rfen,
  sdwid,
  sdcsz,
  sdrsz,
  sdbsz
)
```

Arguments

`trc` SDRAM Trc value:

- `EMIF_SDCTL_TRC_OF(x)`

`trp` SDRAM Trp value:

- `EMIF_SDCTL_TRP_OF(x)`

`trcd` SDRAM Trcd value:

- `EMIF_SDCTL_TRCD_OF(x)`

`init` Forces initialization of all SDRAM:

- `EMIF_SDCTL_INIT_NO`
- `EMIF_SDCTL_INIT_YES`

`rfen` Refresh enable:

- `EMIF_SDCTL_RFEN_DISABLE`
- `EMIF_SDCTL_RFEN_ENABLE`

`sdwid` SDRAM width select:

- `EMIF_SDCTL_SDWID_NA`
- `EMIF_SDCTL_SDWID_4X8BIT`
- `EMIF_SDCTL_SDWID_2X16BIT`

`sdcsz` SDRAM column size:

- `EMIF_SDCTL_SDCSZ_NA`
- `EMIF_SDCTL_SDCSZ_9COL`
- `EMIF_SDCTL_SDCSZ_8COL`
- `EMIF_SDCTL_SDCSZ_10COL`

	`sdrsz`	SDRAM row size: • `EMIF_SDCTL_SDRSZ_NA` • `EMIF_SDCTL_SDRSZ_11ROW` • `EMIF_SDCTL_SDRSZ_12ROW` • `EMIF_SDCTL_SDRSZ_13ROW`
	`sdbsz`	SDRAM bank size: • `EMIF_SDCTL_SDBSZ_NA` • `EMIF_SDCTL_SDBSZ_2BANKS` • `EMIF_SDCTL_SDBSZ_4BANKS`
Return Value	`SDCTL Value`	Constructed register value

Description Use this macro to make a value suitable for the EMIF SDRAM control register.

The default SDRAM control register value is `EMIF_SDCTL_DEFAULT`.

Use of the *EMIF_MK* macros makes it simpler to construct register values based on field values. You have a choice of using integer constants, integer variables, or the symbolic constants for arguments. All field values are right justified.

Refer to the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for descriptions of the arguments.

Example

```
UINT32 SdCtl;

/* you can do this /
SdCtl = EMIF_MK_SDCTL(0,0,0,0,0,0,0,0,0);

/ or to be more readable, you can do this */
SdCtl = EMIF_MK_SDCTL(
  EMIF_SDCTL_TRC_OF(0),
  EMIF_SDCTL_TRP_OF(0),
  EMIF_SDCTL_TRCD_OF(0),
  EMIF_SDCTL_INIT_NO,
  EMIF_SDCTL_RFEN_DISABLE,
  EMIF_SDCTL_SDWID_4X8BIT,
  EMIF_SDCTL_SDCSZ_NA,
  EMIF_SDCTL_SDRSZ_11ROW,
  EMIF_SDCTL_SDBSZ_NA
);
```

4.8.7 EMIF_MK_SDEXT

Makes a value suitable for the EMIF SDRAM extension register

Macro

```
EMIF_MK_SDEXT(
  tcl,
  tras,
  trrd,
  twr,
  thzp,
  rd2rd,
  rd2deac,
  rd2wr,
  r2wdqm,
  wr2wr,
  wr2deac,
  wr2rd
)
```

Arguments

`tcl`	SDRAM CAS latency: • `EMIF_SDEXT_TCL_OF(x)`
`tras`	SDRAM Tras value: • `EMIF_SDEXT_TRAS_OF(x)`
`trrd`	SDRAM Trrd value: • `EMIF_SDEXT_TRRD_OF(x)`
`twr`	SDRAM Twr value: • `EMIF_SDEXT_TWR_OF(x)`
`thzp`	SDRAM Thzp value: • `EMIF_SDEXT_THZP_OF(x)`
`rd2rd`	Read to read clocks: • `EMIF_SDEXT_RD2RD_OF(x)`
`rd2deac`	Read to DEAC clocks: • `EMIF_SDEXT_RD2DEAC_OF(x)`
`rd2wr`	Read to write cycles: • `EMIF_SDEXT_RD2WR_OF(x)`
`r2wdqm`	BEx high clocks: • `EMIF_SDEXT_R2WDQM_OF(x)`
`wr2wr`	Write to write clocks: • `EMIF_SDEXT_WR2WR_OF(x)`

	wr2deac	Write to DEAC cycles: • EMIF_SDEXT_WR2DEAC_OF(x)
	wr2rd	Write to read cycles: • EMIF_SDEXT_WR2RD_OF(x)
Return Value	SDEXT Value	Constructed register value

Description

Use this macro to make a value suitable for the EMIF SDRAM extension register.

The default SDRAM extension register value is EMIF_SDEXT_DEFAULT.

Use of the *EMIF_MK* macros makes it simpler to construct register values based on field values. You have a choice of using integer constants, integer variables, or the symbolic constants for arguments. All field values are right justified.

Refer to the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for descriptions of the arguments.

Example

```
UINT32 SdExt;

/* you can do this /
SdExt = EMIF_MK_SDEXT(0,0,0,0,0,0,0,0,0,0,0,0);

/ or to be more readable, you can do this */
SdExt = EMIF_MK_SDEXT(
  EMIF_SDEXT_TCL_OF(0),
  EMIF_SDEXT_TRAS_OF(0),
  EMIF_SDEXT_TRRD_OF(0),
  EMIF_SDEXT_TWR_OF(0),
  EMIF_SDEXT_THZP_OF(0),
  EMIF_SDEXT_RD2RD_OF(0),
  EMIF_SDEXT_RD2DEAC_OF(0),
  EMIF_SDEXT_RD2WR_OF(0),
  EMIF_SDEXT_R2WDQM_OF(0),
  EMIF_SDEXT_WR2WR_OF(0),
  EMIF_SDEXT_WR2DEAC_OF(0),
  EMIF_SDEXT_WR2RD_OF(0)
);
```

4.8.8 EMIF_MK_SDTIM

Makes a value suitable for the EMIF SDRAM timing register

Macro

```
EMIF_MK_SDTIM(
  period,
  xrfr
)
```

Arguments

`period`	Refresh period: • `EMIF_SDTIM_PERIOD_OF(x)`
`xrfr`	Extra refreshes: • `EMIF_SDTIM_XRFR_NA` • `EMIF_SDTIM_XRFR_OF(x)`

Return Value

`SDTIM Value`	Constructed register value

Description

Use this macro to make a value suitable for the EMIF SDRAM timing register.

The default SDRAM timing register value is `EMIF_SDTIM_DEFAULT`.

Use of the *EMIF_MK* macros makes it simpler to construct register values based on field values. You have a choice of using integer constants, integer variables, or the symbolic constants for arguments. All field values are right justified.

Refer to the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for descriptions of the arguments.

Example

```
UINT32 SdTim;

/* you can do this /
SdTim = EMIF_MK_SDTIM(0,0);

/ or to be more readable, you can do this */
SdTim = EMIF_MK_SDTIM(
  EMIF_SDTIM_PERIOD_OF(0),
  EMIF_SDTIM_XRFR_OF(0)
);
```

4.8.9 EMIF_SUPPORT

A compile time constant that has a value of 1 if the device supports the EMIF module

Constant EDMA_SUPPORT

Description Compile time constant that has a value of 1 if the device supports the EMIF module and 0 otherwise. You are not required to use this constant.

Currently, all devices support this module.

Example

```
#if (EMIF_SUPPORT)
  /* user EMIF configuration /
#endif
```

4.9 HPI

4.9.1 HPI_GetDspint

Reads the DSPINT bit from HPIC register

Function `UINT32 HPI_GetDspint();`

Arguments `none`

Return Value `DSPINT` Returns the value of the DSPINT bit, 0 or 1

Description This function reads the DSPINT bit from the HPIC register.

Example

```
if (HPI_GetDspint()) {
}
```

4.9.2 HPI_GetEventId

Obtain the IRQ even associated with the HPI device

Function `UINT32 HPI_GetEventId();`

Arguments `none`

Return Value `Event ID` Returns the IRQ event for the HPI device

Description Use this function to obtain the IRQ event associated with the HPI device. Currently this is `IRQ_EVT_DSPINT`.

Example

```
HpiEventId = HPI_GetEventId();
```

4.9.3 HPI_GetFetch

Reads the FETCH flag from the HPIC register and returns its value.

Function `UINT32 HPI_GetFetch();`

Arguments `none`

Return Value `FETCH` Returns the value of the FETCH flag, 0 or 1

Description This function reads the FETCH flag from the HPIC register and returns its value.

Example

```
flag = HPI_GetFetch();
```

4.9.4 HPI_GetHint

Returns the value of the HINT bit of the HPIC register

Function `UINT32 HPI_GetHint();`

Arguments `none`

Return Value `HINT` Returns the value of the HINT bit, 0 or 1

Description This function returns the value of the HINT bit of the HPIC register.

Example

```
hint = HPI_GetHint();
```

4.9.5 HPI_GetHrdy

Returns the value of the HRDY bit of the HPIC register

Function	`UINT32 HPI_GetHrdy();`	
Arguments	`none`	
Return Value	`HRDY`	Returns the value of the HRDY bit, 0 or 1
Description	This function returns the value of the HRDY bit of the HPIC register.	
Example	`hrdy = HPI_GetHrdy();`	

4.9.6 HPI_GetHwob

Returns the value of the HWOB bit of the HPIC register

Function	`UINT32 HPI_GetHwob();`	
Arguments	`none`	
Return Value	`HWOB`	Returns the value of the HWOB bit, 0 or 1
Description	This function returns the value of the HWOB bit of the HPIC register.	
Example	`hwob = HPI_GetHwob();`	

4.9.7 HPI_SetDspint

Writes the value to the DSPINT field of the HPIC register

Function	`void HPI_SetDspint(`
	`  UINT32 Val`
	`);`	

Arguments	`Val`	Value to write to DSPINT: 0 or 1
Return Value	`none`	
Description	This function writes the value to the DSPINT file of the HPIC register	
Example	`HPI_SetDspint(0);` `HPI_SetDspint(1);`	

4.9.8 HPI_SetHint

Writes the value to the HINT field of the HPIC register

Function	`void HPI_SetHint(`
	`  UINT32 Val`
	`);`	

Arguments	`Val`	
Return Value	`none`	Value to write to HINT: 0 or 1

Description This function writes the value to the HINT file of the HPIC register

Example

```
HPI_SetHint(0);
HPI_SetHint(1);
```

4.9.9 HPI_SUPPORT

A compile time constant whose value is 1 if the device supports the HPI module

Constant HPI_SUPPORT

Description Compile time constant that has a value of 1 if the device supports the HPI module and 0 otherwise. You are not required to use this constant.

Example

```
#if (HPI_SUPPORT)
  /* user HPI configuration /
#endif
```

4.10 IRQ

4.10.1 IRQ_Clear

Clears the event flag from the IFR register

Function

```
void IRQ_Clear(
  UINT32 EventId
);
```

Arguments

`EventId` Event ID. See IRQ_EVT_NNNN for a complete list of events.

Return Value `none`

Description Clears the event flag from the IFR register

Example

```
IRQ_Clear(IRQ_EVT_TINT0);
```

4.10.2 IRQ_Disable

Disables the specified event

Function

```
void IRQ_Disable(
  UINT32 EventId
);
```

Arguments

`EventId` Event ID. See IRQ_EVT_NNNN for a complete list of events.

Return Value `none`

Description Disables the specified event.

Example

```
IRQ_Disable(IRQ_EVT_TINT0);
```

4.10.3 IRQ_Enable

Enables the specified event

Function

```
void IRQ_Enable(
  UINT32 EventId
);
```

Arguments

`EventId` Event ID. See IRQ_EVT_NNNN for a complete list of events.

Return Value `none`

Description Enables the specified event.

Example

```
IRQ_Enable(IRQ_EVT_TINT0);
```

4.10.4 IRQ_EVT_NNNN

These are the IRQ events

Constant

```
IRQ_EVT_DSPINT
IRQ_EVT_TINT0
IRQ_EVT_TINT1
IRQ_EVT_SDINT
IRQ_EVT_EXTINT4
IRQ_EVT_EXTINT5
IRQ_EVT_EXTINT6
IRQ_EVT_EXTINT7
IRQ_EVT_DMAINT0
IRQ_EVT_DMAINT1
IRQ_EVT_DMAINT2
IRQ_EVT_DMAINT3
IRQ_EVT_EDMAINT
IRQ_EVT_XINT0
IRQ_EVT_RINT0
IRQ_EVT_XINT1
IRQ_EVT_RINT1
IRQ_EVT_XINT2
IRQ_EVT_RINT2
```

Description These are the IRQ events. Refer to the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for more details regarding the events.

4.10.5 IRQ_Map

Maps an event to a physical interrupt number by configuring the interrupt selector MUX registers

Function

```
void IRQ_Map(
  UINT32 EventId,
  int IntNumber
);
```

Arguments

`EventId`	Event ID. See IRQ_EVT_NNNN for a complete list of events.
`IntNumber`	Interrupt number, 0 to 15

Return Value `none`

Description This function maps an event to a physical interrupt number by configuring the interrupt selector MUX registers. For most cases, the default map is sufficient and does not need to be changed.

Example

```
IRQ_Map(IRQ_EVT_TINT0,12);
```

4.10.6 IRQ_Set

Sets the specified event by writing to the appropriate ISR register

Function

```
void IRQ_Set(
  UINT32 EventId
);
```

Arguments

`EventId`	Event ID. See IRQ_EVT_NNNN for a complete list of events.

Return Value `none`

Description Sets the specified event by writing to the appropriate ISR register. This basically allows software triggering of events.

Example

```
IRQ_Set(IRQ_EVT_TINT0);
```

4.10.7 IRQ_SUPPORT

A compile time constant whose value is 1 if the device supports the IRQ module

Constant `IRQ_SUPPORT`

Description Compile time constant that has a value of 1 if the device supports the IRQ module and 0 otherwise. You are not required to use this constant.

Currently, all devices support this module.

Example

```
#if (IRQ_SUPPORT)
  /* user IRQ configuration /
#endif
```

4.10.8 IRQ_Test

Allows testing an event to see if its flag is set in the IFR register

Function

```
BOOL IRQ_Test(
  UINT32 EventId
);
```

Arguments

`EventId`	Event ID. See IRQ_EVT_NNNN for a complete list of events.

Return Value

`Flag`	Returns event flag; 0 or 1

Description Use this function to test an event to see if its flag is set in the IFR register.

Example

```
while (!IRQ_Test(IRQ_EVT_TINT0);
```

4.11 MCBSP

4.11.1 MCBSP_Close

Closes a MCBSP port previously opened via MCBSP_Open()

Function

```
void MCBSP_Close(
  MCBSP_HANDLE hMcbsp
);
```

Arguments

hMcbsp	Handle to MCBSP port, see MCBSP_Open()

Return Value

none

Description

This function closes a MCBSP port previously opened via MCBSP_Open(). The registers for the MCBSP port are set to their power-on defaults. Any associated interrupts are disabled and cleared.

Example

```
MCBSPClose(hMcbsp);
```

4.11.2 MCBSP_CONFIG

Used to setup a MCBSP port

Structure MCBSP_CONFIG

Members

UINT32 spcr	Serial port control register value
UINT32 rcr	Receive control register value
UINT32 xcr	Transmit control register value
UINT32 srgr	Sample rate generator register value
UINT32 mcr	Multichannel control register value
UINT32 rcer	Receive channel enable register value
UINT32 xcer	Transmit channel enable register value
UINT32 pcr	Pin control register value

Description

This is the MCBSP configuration structure used to set up a MCBSP port. You create and initialize this structure and then pass its address to the MCBSP_ConfigA() function. You can use literal values or the *MCBSP_MK* macros to create the structure member values.

Example

```
MCBSP_CONFIG MyConfig = {
  0x00012001, /* spcr */
  0x00010140, /* rcr  */
  0x00010140, /* xcr  */
  0x00000000, /* srgr */
  0x00000000, /* mcr  */
  0x00000000, /* rcer */
  0x00000000, /* xcer */
  0x00000000  /* pcr  */
};
...
MCBSP_ConfigA(hMcbsp,&MyConfig);
```

4.11.3 MCBSP_ConfigA

Sets up the MCBSP port using the configuration structure

Function

```
void MCBSP_ConfigA(
  MCBSP_HANDLE hMcbsp,
  MCBSP_CONFIG *Config
);
```

Arguments

`hMcbsp`	Handle to MCBSP port. See `MCBSP_Open()`
`Config`	Pointer to an initialized configuration structure

Return Value `none`

Description Sets up the MCBSP port using the configuration structure. The values of the structure are written to the port registers. The serial port control register (*spcr*) is written last. See also `MCBSP_ConfigB()` and `MCBSP_CONFIG`.

Example

```
MCBSP_CONFIG MyConfig = {
  0x00012001, /* spcr */
  0x00010140, /* rcr  */
  0x00010140, /* xcr  */
  0x00000000, /* srgr */
  0x00000000, /* mcr  */
  0x00000000, /* rcer */
  0x00000000, /* xcer */
  0x00000000  /* pcr  */
};
...
MCBSP_ConfigA(hMcbsp,&MyConfig);
```

4.11.4 MCBSP_ConfigB

Sets up the MCBSP port using the register values passed in

Function

```
void MCBSP_ConfigB(
  MCBSP_HANDLE hMcbsp,
  UINT32 spcr,
  UINT32 rcr,
  UINT32 xcr,
  UINT32 srgr,
  UINT32 mcr,
  UINT32 rcer,
  UINT32 xcer,
  UINT32 pcr
);
```

Arguments

`hMcbsp`	Handle to MCBSP port. See `MCBSP_Open()`
`spcr`	Serial port control register value
`rcr`	Receive control register value
`xcr`	Transmit control register value
`srgr`	Sample rate generator register value
`mcr`	Multichannel control register value
`rcer`	Receive channel enable register value
`xcer`	Transmit channel enable register value
`pcr`	Pin control register value

Return Value

`none`

Description

Sets up the MCBSP port using the register values passed in. The register values are written to the port registers. The serial port control register (*spcr*) is written last. See also `MCBSP_ConfigA()`.

You may use literal values for the arguments or for readability. You may use the *MCBSP_MK* macros to create the register values based on field values.

Example

```
MCBSP_ConfigB(hMcbsp,
  0x00012001, /* spcr */
  0x00010140, /* rcr  */
  0x00010140, /* xcr  */
  0x00000000, /* srgr */
  0x00000000, /* mcr  */
  0x00000000, /* rcer */
  0x00000000, /* xcer */
  0x00000000  /* pcr  */
);
```

4.11.5 MCBSP_EnableFsync

Enables the frame sync generator for the given port

Function
```
void MCBSP_EnableFsync(
  MCBSP_HANDLE hMcbsp
);
```

Arguments `hMcbsp` Handle to MCBSP port. See `MCBSP_Open()`

Return Value `none`

Description Use this function to enable the frame sync generator for the given port.

Example
```
MCBSP_EnableFsync(hMcbsp);
```

4.11.6 MCBSP_EnableRcv

Enables the receiver for the given port

Function
```
void MCBSP_EnableRcv(
  MCBSP_HANDLE hMcbsp
);
```

Arguments `hMcbsp` Handle to MCBSP port. See `MCBSP_Open()`

Return Value `none`

Description Use this function to enable the receiver for the given port.

Example
```
MCBSP_EnableRcv(hMcbsp);
```

4.11.7 MCBSP_EnableSrgr

Enables the sample rate generator for the given port

Function
```
void MCBSP_EnableSrgr(
  MCBSP_HANDLE hMcbsp
);
```

Arguments `hMcbsp` Handle to MCBSP port. See `MCBSP_Open()`

Return Value `none`

Description Use this function to enable the sample rate generator for the given port.

Example
```
MCBSP_EnableSrgr(hMcbsp);
```

4.11.8 MCBSP_EnableXmt

Enables the transmitter for the given port

Function

```
void MCBSP_EnableXmt(
  MCBSP_HANDLE hMcbsp
);
```

Arguments

`hMcbsp`	Handle to MCBSP port. See `MCBSP_Open()`

Return Value

`none`

Description

Use this function to enable the transmitter for the given port.

Example

```
MCBSP_EnableXmt(hMcbsp);
```

4.11.9 MCBSP_GetPins

Reads the values of the port pins when configured as general purpose I/Os

Function

```
UINT32 MCBSP_GetPins(
  MCBSP_HANDLE hMcbsp
);
```

Arguments

`hMcbsp`	Handle to MCBSP port. See `MCBSP_Open()`

Return Value

`Pin Mask` Bit-Mask of pin values

- `MCBSP_PIN_CLKX`
- `MCBSP_PIN_FSX`
- `MCBSP_PIN_DX`
- `MCBSP_PIN_CLKR`
- `MCBSP_PIN_FSR`
- `MCBSP_PIN_DR`
- `MCBSP_PIN_CLKS`

Description

This function reads the values of the port pins when configured as general purpose input/outputs.

Example

```
UINT32 PinMask;
...
PinMask = MCBSP_GetPins(hMcbsp);
if (PinMask & MCBSP_PIN_DR) {
  ...
}
```

4.11.10 MCBSP_GetRcvAddr

Returns the address of the data receive register (DRR)

Function

```
UINT32 MCBSP_GetRcvAddr(
  MCBSP_HANDLE hMcbsp
);
```

Arguments	`hMcbsp`	Handle to MCBSP port. See `MCBSP_Open()`
Return Value	`Receive Address`	DRR register address

Description Returns the address of the data receive register, DRR. This value is needed when setting up DMA transfers to read from the serial port. See also `MCBSP_GetXmtAddr()`.

Example

```
Addr = MCBSP_GetRcvAddr(hMcbsp);
```

4.11.11 MCBSP_GetRcvEventId

Retrieves the transmit event ID for the given port

Function

```
UINT32 MCBSP_GetRcvEventId(
  MCBSP_HANDLE hMcbsp
);
```

Arguments	`hMcbsp`	Handle to MCBSP port. See `MCBSP_Open()`
Return Value	`Receive Event ID`	Receiver event ID

Description Retrieves the receive event ID for the given port.

Example

```
UINT32 RecvEventId;
...
RecvEventId = MCBSP_GetRcvEventId(hMcbsp);
IRQ_Enable(RecvEventId);
```

4.11.12 MCBSP_GetXmtAddr

Returns the address of the data transmit register, DXR

Function

```
UINT32 MCBSP_GetXmtEventId(
  MCBSP_HANDLE hMcbsp
);
```

Arguments	`hMcbsp`	Handle to MCBSP port. See `MCBSP_Open()`
Return Value	`Transmit Address`	DXR register address

Description Returns the address of the data transmit register, DXR. This value is needed when setting up DMA transfers to write to the serial port. See also `MCBSP_GetRcvAddr()`.

Example

```
Addr = MCBSP_GetXmtAddr(hMcbsp);
```

4.11.13 MCBSP_GetXmtEventId

Retrieves the transmit event ID for the given port

Function

```
UINT32 MCBSP_GetXmtEventId(
  MCBSP_HANDLE hMcbsp
);
```

Arguments

`hMcbsp`	Handle to MCBSP port. See `MCBSP_Open()`

Return Value

`Transmit Event ID`	Event ID of transmitter

Description

Retrieves the transmit event ID for the given port.

Example

```
UINT32 XmtEventId;
...
XmtEventId = MCBSP_GetXmtEventId(hMcbsp);
IRQ_Enable(XmtEventId);
```

4.11.14 MCBSP_MK_MCR

Makes a value suitable for the multichannel control register

Macro

```
MCBSP_MK_MCR(
  rmcm,
  rpablk,
  rpbblk,
  xmcm,
  xpablk,
  xpbblk
)
```

Arguments

`rmcm` Receive multichannel selection enable:

- `MCBSP_MCR_RMCM_CHENABLE`
- `MCBSP_MCR_RMCM_ELDISABLE`

`rpablk` Receive partition A subframe:

- `MCBSP_MCR_RPABLK_SF0`
- `MCBSP_MCR_RPABLK_SF2`
- `MCBSP_MCR_RPABLK_SF4`
- `MCBSP_MCR_RPABLK_SF6`

`rpbblk` Receive partition B subframe:

- `MCBSP_MCR_RPBBLK_SF1`
- `MCBSP_MCR_RPBBLK_SF3`
- `MCBSP_MCR_RPBBLK_SF5`
- `MCBSP_MCR_RPBBLK_SF7`

	xmcm	Transmit multichannel selection enable: • MCBSP_MCR_XMCM_ENNOMASK • MCBSP_MCR_XMCM_DISXP • MCBSP_MCR_XMCM_ENMASK • MCBSP_MCR_XMCM_DISRP
	xpablk	Transmit partition A subframe: • MCBSP_MCR_XPABLK_SF0 • MCBSP_MCR_XPABLK_SF2 • MCBSP_MCR_XPABLK_SF4 • MCBSP_MCR_XPABLK_SF6
	xpbblk	Transmit partition B subframe: • MCBSP_MCR_XPBBLK_SF1 • MCBSP_MCR_XPBBLK_SF3 • MCBSP_MCR_XPBBLK_SF5 • MCBSP_MCR_XPBBLK_SF7
Return Value	MCR Value	Constructed register value

Description

Use this macro to make a value suitable for the multichannel control register.

The power-on default value is MCBSP_MCR_DEFAULT.

Use of the *MCBSP_MK* macros makes it simpler to construct register values based on field values. You have a choice of using integer constants, integer variables, or the symbolic constants for arguments. All field values are right justified.

Refer to the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for descriptions of the arguments.

Example

```
UINT32 Mcr;

/* you can do this /
Mcr = MCBSP_MK_MCR(0,0,0,0,0,0);

/ or to be more readable, you can do this */
Mcr = MCBSP_MK_MCR(
  MCBSP_MCR_RMCM_CHENABLE,
  MCBSP_MCR_RPABLK_SF0,
  MCBSP_MCR_RPBBLK_SF1,
  MCBSP_MCR_XMCM_ENNOMASK,
  MCBSP_MCR_XPABLK_SF0,
  MCBSP_MCR_XPBBLK_SF1
);
```

4.11.15 MCBSP_MK_PCR

Makes a value suitable for the pin control register

Macro

```
MCBSP_MK_PCR(
  clkrp,
  clkxp,
  fsrp,
  fsxp,
  dxstat,
  clksstat,
  clkrm,
  clkxm,
  fsrm,
  fsxm,
  rioen,
  xioen
)
```

Arguments

`clkrp` Receive clock polarity:

- `MCBSP_PCR_CLKRP_FALLING`
- `MCBSP_PCR_CLKRP_RISING`

`clkxp` Transmit clock polarity:

- `MCBSP_PCR_CLKXP_RISING`
- `MCBSP_PCR_CLKXP_FALLING`

`fsrp` Receive frame sync polarity:

- `MCBSP_PCR_FSRP_ACTIVEHIGH`
- `MCBSP_PCR_FSRP_ACTIVELOW`

	`fsxp`	Transmit frame sync polarity: • `MCBSP_PCR_FSXP_ACTIVEHIGH` • `MCBSP_PCR_FSXP_ACTIVELOW`
	`dxstat`	DX pin status: • `MCBSP_PCR_DXSTAT_0` • `MCBSP_PCR_DXSTAT_1`
	`clksstat`	CLKS pin status: • `MCBSP_PCR_CLKSSTAT_0` • `MCBSP_PCR_CLKSSTAT_1`
	`clkrm`	Receiver clock mode: • `MCBSP_PCR_CLKRM_INPUT` • `MCBSP_PCR_CLKRM_OUTPUT`
	`clkxm`	Transmitter clock mode: • `MCBSP_PCR_CLKXM_INPUT` • `MCBSP_PCR_CLKXM_OUTPUT`
	`fsrm`	Receive frame sync mode: • `MCBSP_PCR_FSRM_EXTERNAL` • `MCBSP_PCR_FSRM_INTERNAL`
	`fsxm`	Transmit frame sync mode: • `MCBSP_PCR_FSXM_EXTERNAL` • `MCBSP_PCR_FSXM_INTERNAL`
	`rioen`	Receiver general purpose IO mode: • `MCBSP_PCR_RIOEN_SP` • `MCBSP_PCR_RIOEN_GPIO`
	`xioen`	Transmitter general purpose IO mode: • `MCBSP_PCR_XIOEN_SP` • `MCBSP_PCR_XIOEN_GPIO`
Return Value	`PCR Value`	Constructed register value

Description Use this macro to make a value suitable for the pin control register.

The power-on default value is `MCBSP_PCR_DEFAULT`.

Use of the *MCBSP_MK* macros makes it simpler to construct register values based on field values. You have a choice of using integer constants, integer variables, or the symbolic constants for arguments. All field values are right justified.

Refer to the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for descriptions of the arguments.

Example

```
UINT32 Pcr;

/* you can do this /
Pcr = MCBSP_MK_PCR(0,0,0,0,0,0,0,0,0,0,0,0);

/ or to be more readable, you can do this */
Pcr = MCBSP_MK_PCR(
  MCBSP_PCR_CLKRP_FALLING,
  MCBSP_PCR_CLKXP_RISING,
  MCBSP_PCR_FSRP_ACTIVEHIGH,
  MCBSP_PCR_FSXP_ACTIVEHIGH,
  MCBSP_PCR_DXSTAT_0,
  MCBSP_PCR_CLKSSTAT_0,
  MCBSP_PCR_CLKRM_INPUT,
  MCBSP_PCR_CLKXM_INPUT,
  MCBSP_PCR_FSRM_EXTERNAL,
  MCBSP_PCR_FSXM_EXTERNAL,
  MCBSP_PCR_RIOEN_SP,
  MCBSP_PCR_XIOEN_SP
);
```

4.11.16 MCBSP_MK_RCER

Makes a value suitable for the receive channel enable register

Macro

```
MCBSP_MK_RCER(
  rcea,
  rceb
)
```

Arguments

`rcea` Receive channel enable bit-mask A:

- `MCBSP_RCER_RCEA_OF(x)`

`rceb` Receive channel enable bit-mask B:

- `MCBSP_RCER_RCEB_OF(x)`

Return Value `RCER Value` Constructed register value

Description Use this macro to make a value suitable for the receive channel enable register.

The power-on default value is `MCBSP_RCER_DEFAULT`.

Use of the *MCBSP_MK* macros makes it simpler to construct register values based on field values. You have a choice of using integer constants, integer variables, or the symbolic constants for arguments. All field values are right justified.

Refer to the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for descriptions of the arguments.

Example

```
UINT32 Rcer;

/* you can do this /
Rcer = MCBSP_MK_RCER(0x0000,0x0000);

/ or to be more readable, you can do this */
Rcer = MCBSP_MK_RCER(
  MCBSP_RCER_RCEA_OF(0x0000),
  MCBSP_RCER_RCEB_OF(0x0000)
);
```

4.11.17 MCBSP_MK_RCR

Makes a value suitable for the receive control register

Macro

```
MCBSP_MK_RCR(
  rwdrevrs,
  rwdlen1,
  rfrlen1,
  rphase2,
  rdatdly,
  rfig,
  rcompand,
  rwdlen2,
  rfrlen2,
  rphase
)
```

Arguments

`rwdrevrs` Receive 32-bit reversal:

- `MCBSP_RCR_RWDREVRS_NA`
- `MCBSP_RCR_RWDREVRS_DISABLE`
- `MCBSP_RCR_RWDREVRS_ENABLE`

`rwdlen1` Receive element length in phase 1:

- `MCBSP_RCR_RWDLEN1_8BIT`
- `MCBSP_RCR_RWDLEN1_12BIT`
- `MCBSP_RCR_RWDLEN1_16BIT`
- `MCBSP_RCR_RWDLEN1_20BIT`
- `MCBSP_RCR_RWDLEN1_24BIT`
- `MCBSP_RCR_RWDLEN1_32BIT`

`rfrlen1` Receive frame length in phase 1:

- `MCBSP_RCR_RFRLEN1_OF(x)`

`rphase2` Receive phase 2:

- `MCBSP_RCR_RPHASE2_NA`
- `MCBSP_RCR_RPHASE2_NORMAL`
- `MCBSP_RCR_RPHASE2_OPPOSITE`

`rdatdly` Receive data delay:

- `MCBSP_RCR_RDATDLY_0BIT`
- `MCBSP_RCR_RDATDLY_1BIT`
- `MCBSP_RCR_RDATDLY_2BIT`

`rfig` Receive frame ignore:

MCBSP_RCR_RFIG_YES

MCBSP_RCR_RFIG_NO

`rcompand` Receive companding mode:

- `MCBSP_RCR_RCOMPAND_MSB`
- `MCBSP_RCR_RCOMPAND_8BITLSB`
- `MCBSP_RCR_RCOMPAND_ULAW`
- `MCBSP_RCR_RCOMPAND_ALAW`

`rwdlen2` Receive element length in phase 2:

- `MCBSP_RCR_RWDLEN2_8BIT`
- `MCBSP_RCR_RWDLEN2_12BIT`
- `MCBSP_RCR_RWDLEN2_16BIT`
- `MCBSP_RCR_RWDLEN2_20BIT`
- `MCBSP_RCR_RWDLEN2_24BIT`
- `MCBSP_RCR_RWDLEN2_32BIT`

	`rfrlen2`	Receive frame length in phase 2: • `MCBSP_RCR_RFRLEN2_OF(x)`
	`rphase`	Receive phases: • `MCBSP_RCR_RPHASE_SINGLE` • `MCBSP_RCR_RPHASE_DUAL`
Return Value	`RCR Value`	Constructed register value

Description Use this macro to make a value suitable for the receive control register.

The power-on default value is `MCBSP_RCR_DEFAULT`.

Use of the *MCBSP_MK* macros makes it simpler to construct register values based on field values. You have a choice of using integer constants, integer variables, or the symbolic constants for arguments. All field values are right justified.

Refer to the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for descriptions of the arguments.

Example

```
UINT32 Rcr;

/* you can do this /
Rcr = MCBSP_MK_RCR(0,0,0,0,0,0,0,0,0,0);

/ or to be more readable, you can do this */
Rcr = MCBSP_MK_RCR(
  MCBSP_RCR_RWDREVRS_NA,
  MCBSP_RCR_RWDLEN1_8BIT,
  MCBSP_RCR_RFRLEN1_OF(0),
  MCBSP_RCR_RPHASE2_NA,
  MCBSP_RCR_RDATDLY_0BIT,
  MCBSP_RCR_RFIG_YES,
  MCBSP_RCR_RCOMPAND_MSB,
  MCBSP_RCR_RWDLEN2_8BIT,
  MCBSP_RCR_RFRLEN2_OF(0),
  MCBSP_RCR_RPHASE_SINGLE
);
```

4.11.18 MCBSP_MK_SPCR

Makes a value suitable for the serial port control register

Macro

```
MCBSP_MK_SPCR(
  rrst,
  rintm,
  dxena,
  clkstp,
  rjust,
  dlb,
  xrst,
  xintm,
  grst,
  frst
)
```

Arguments

`rrst` Receiver reset:

- `MCBSP_SPCR_RRST_YES`
- `MCBSP_SPCR_RRST_NO`

`rintm` Receiver interrupt mode:

- `MCBSP_SPCR_RINTM_RRDY`
- `MCBSP_SPCR_RINTM_EOS`
- `MCBSP_SPCR_RINTM_FRM`
- `MCBSP_SPCR_RINTM_RSYNCERR`

`dxena` DX enabler:

- `MCBSP_SPCR_DXENA_NA`
- `MCBSP_SPCR_DXENA_OFF`
- `MCBSP_SPCR_DXENA_ON`

`clkstp` Clock stop mode:

- `MCBSP_SPCR_CLKSTP_DISABLE`
- `MCBSP_SPCR_CLKSTP_NODELAY`
- `MCBSP_SPCR_CLKSTP_DELAY`

`rjust` Receive data justification mode:

- `MCBSP_SPCR_RJUST_RZF`
- `MCBSP_SPCR_RJUST_RSE`
- `MCBSP_SPCR_RJUST_LZF`

	dlb	Digital loopback mode: • MCBSP_SPCR_DLB_OFF • MCBSP_SPCR_DLB_ON
	xrst	Transmitter reset: • MCBSP_SPCR_XRST_YES • MCBSP_SPCR_XRST_NO
	xintm	Transmitter interrupt mode: • MCBSP_SPCR_XINTM_XRDY • MCBSP_SPCR_XINTM_EOS • MCBSP_SPCR_XINTM_FRM • MCBSP_SPCR_XINTM_XSYNCERR
	grst	Sample rate generator reset: • MCBSP_SPCR_GRST_YES • MCBSP_SPCR_GRST_NO
	frst	Frame sync generator reset: • MCBSP_SPCR_FRST_YES • MCBSP_SPCR_FRST_NO
Return Value	SPCR Value	Constructed register value

Description

Use this macro to make a value suitable for the serial port control register.

The power-on default value is MCBSP_SPCR_DEFAULT.

Use of the *MCBSP_MK* macros makes it simpler to construct register values based on field values. You have a choice of using integer constants, integer variables, or the symbolic constants for arguments. All field values are right justified.

Refer to the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for descriptions of the arguments.

Example

```
UINT32 Spcr;

/* you can do this /
Spcr = MCBSP_MK_SPCR(0,0,0,0,0,0,0,0,0,0);

/ or to be more readable, you can do this */
Spcr = MCBSP_MK_SPCR(
  MCBSP_SPCR_RRST_YES,
  MCBSP_SPCR_RINTM_RRDY,
  MCBSP_SPCR_DXENA_NA,
  MCBSP_SPCR_CLKSTP_DISABLE,
  MCBSP_SPCR_RJUST_RZF,
  MCBSP_SPCR_DLB_OFF,
  MCBSP_SPCR_XRST_YES,
  MCBSP_SPCR_XINTM_XRDY,
  MCBSP_SPCR_GRST_YES,
  MCBSP_SPCR_FRST_YES
);
```

4.11.19 MCBSP_MK_SRGR

Makes a value suitable for the sample rate generator register

Macro

```
MCBSP_MK_SRGR(
  clkgdv,
  fwid,
  fper,
  fsgm,
  clksm,
  clksp,
  gsync
)
```

Arguments

`clkgdv` Clock divider:

- `MCBSP_SRGR_CLKGDV_OF(x)`

`fwid` Frame width:

- `MCBSP_SRGR_FWID_OF(x)`

`fper` Frame period:

- `MCBSP_SRGR_FPER_OF(x)`

`fsgm` Transmit frame sync mode:

- `MCBSP_SRGR_FSGM_DXR2XSR`
- `MCBSP_SRGR_FSGM_FSG`

	clksm	Clock mode: • MCBSP_SRGR_CLKSM_CLKS • MCBSP_SRGR_CLKSM_INTERNAL
	clksp	CLKS polarity clock edge select: • MCBSP_SRGR_CLKSP_RISING • MCBSP_SRGR_CLKSP_FALLING
	gsync	Clock synchronization: • MCBSP_SRGR_GSYNC_FREE • MCBSP_SRGR_GSYNC_SYNC
Return Value	SRGR Value	Constructed register value

Description Use this macro to make a value suitable for the sample rate generator register.

The power-on default value is MCBSP_SRGR_DEFAULT.

Use of the *MCBSP_MK* macros makes it simpler to construct register values based on field values. You have a choice of using integer constants, integer variables, or the symbolic constants for arguments. All field values are right justified.

Refer to the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for descriptions of the arguments.

Example

```
UINT32 Srgr;

/* you can do this /
Srgr = MCBSP_MK_SRGR(0,0,0,0,0,0,0);

/ or to be more readable, you can do this */
Srgr = MCBSP_MK_SRGR(
  MCBSP_SRGR_CLKGDV_OF(0),
  MCBSP_SRGR_FWID_OF(0),
  MCBSP_SRGR_FPER_OF(0),
  MCBSP_SRGR_FSGM_DXR2XSR,
  MCBSP_SRGR_CLKSM_CLKS,
  MCBSP_SRGR_CLKSP_RISING,
  MCBSP_SRGR_GSYNC_FREE
);
```

4.11.20 MCBSP_MK_XCER

Makes a value suitable for the transmit channel enable register

Macro

```
MCBSP_MK_XCER(
  xcea,
  xceb
)
```

Arguments

`xcea`	Transmit channel enable bit-mask A: • `MCBSP_XCER_XCEA_OF(x)`
`xceb`	Transmit channel enable bit-mask B: • `MCBSP_XCER_XCEB_OF(x)`

Return Value

`XCER Value`	Constructed register value

Description

Use this macro to make a value suitable for the transmit channel enable register.

The power-on default value is `MCBSP_XCER_DEFAULT`.

Use of the *MCBSP_MK* macros makes it simpler to construct register values based on field values. You have a choice of using integer constants, integer variables, or the symbolic constants for arguments. All field values are right justified.

Refer to the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for descriptions of the arguments.

Example

```
UINT32 Xcer;

/* you can do this /
Xcer = MCBSP_MK_XCER(0x0000,0x0000);

/ or to be more readable, you can do this */
Xcer = MCBSP_MK_XCER(
  MCBSP_XCER_XCEA_OF(0x0000),
  MCBSP_XCER_XCEB_OF(0x0000)
);
```

4.11.21 MCBSP_MK_XCR

Makes a value suitable for the transmit control register

Macro

```
MCBSP_MK_XCR(
  xwdrevrs,
  xwdlen1,
  xfrlen1,
  xphase2,
  xdatdly,
  xfig,
  xcompand,
  xwdlen2,
  xfrlen2,
  xphase
)
```

Arguments

`xwdrevrs` Transmit 32-bit reversal:

- `MCBSP_XCR_XWDREVRS_NA`
- `MCBSP_XCR_XWDREVRS_DISABLE`
- `MCBSP_XCR_XWDREVRS_ENABLE`

`xwdlen1` Transmit element length in phase 1:

- `MCBSP_XCR_XWDLEN1_8BIT`
- `MCBSP_XCR_XWDLEN1_12BIT`
- `MCBSP_XCR_XWDLEN1_16BIT`
- `MCBSP_XCR_XWDLEN1_20BIT`
- `MCBSP_XCR_XWDLEN1_24BIT`
- `MCBSP_XCR_XWDLEN1_32BIT`

`xfrlen1` Transmit frame length in phase 1:

- `MCBSP_XCR_XFRLEN1_OF(x)`

`xphase2` Transmit phase 2:

- `MCBSP_XCR_XPHASE2_NA`
- `MCBSP_XCR_XPHASE2_NORMAL`
- `MCBSP_XCR_XPHASE2_OPPOSITE`

`xdatdly` Transmit data delay:

- `MCBSP_XCR_XDATDLY_0BIT`
- `MCBSP_XCR_XDATDLY_1BIT`
- `MCBSP_XCR_XDATDLY_2BIT`

	xfig	Transmit frame ignore: • MCBSP_XCR_XFIG_YES • MCBSP_XCR_XFIG_NO
	xcompand	Transmit companding mode: • MCBSP_XCR_XCOMPAND_MSB • MCBSP_XCR_XCOMPAND_8BITLSB • MCBSP_XCR_XCOMPAND_ULAW • MCBSP_XCR_XCOMPAND_ALAW
	xwdlen2	Transmit element length in phase 2: • MCBSP_XCR_XWDLEN2_8BIT • MCBSP_XCR_XWDLEN2_12BIT • MCBSP_XCR_XWDLEN2_16BIT • MCBSP_XCR_XWDLEN2_20BIT • MCBSP_XCR_XWDLEN2_24BIT • MCBSP_XCR_XWDLEN2_32BIT
	xfrlen2	Transmit frame length in phase 2: • MCBSP_XCR_XFRLEN2_OF(x)
	xphase	Transmit phases: • MCBSP_XCR_XPHASE_SINGLE • MCBSP_XCR_XPHASE_DUAL
Return Value	XCR Value	Constructed register value

Description

Use this macro to make a value suitable for the transmit control register.

The power-on default value is MCBSP_XCR_DEFAULT.

Use of the *MCBSP_MK* macros makes it simpler to construct register values based on field values. You have a choice of using integer constants, integer variables, or the symbolic constants for arguments. All field values are right justified.

Refer to the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for descriptions of the arguments.

Example

```
UINT32 Xcr;

/* you can do this /
Xcr = MCBSP_MK_XCR(0,0,0,0,0,0,0,0,0,0);

/ or to be more readable, you can do this */
Xcr = MCBSP_MK_XCR(
  MCBSP_XCR_XWDREVRS_NA,
  MCBSP_XCR_XWDLEN1_8BIT,
  MCBSP_XCR_XFRLEN1_OF(0),
  MCBSP_XCR_XPHASE2_NA,
  MCBSP_XCR_XDATDLY_0BIT,
  MCBSP_XCR_XFIG_YES,
  MCBSP_XCR_XCOMPAND_MSB,
  MCBSP_XCR_XWDLEN2_8BIT,
  MCBSP_XCR_XFRLEN2_OF(0),
  MCBSP_XCR_XPHASE_SINGLE
);
```

4.11.22 MCBSP_Open

Opens a McBSP port for use

Function

```
MCBSP_HANDLE MCBSP_Open(
  int DevNum,
  UINT32 Flags
);
```

Arguments

`DevNum`	MCBSP device (port) number: • `MCBSP_DEV0` • `MCBSP_DEV1` • `MCBSP_DEV2*`
`Flags`	Open flags; may be logical OR of any of the following: • `MCBSP_OPEN_RESET`

Return Value

`Device Handle`	Returns a device handle

Description

Before a MCBSP port can be used, it must first be opened by this function. Once opened, it cannot be opened again until closed. See `MCBSP_Close()`.The return value is a unique device handle that you use in subsequent MCBSP API calls. If the open fails, `INV` is returned.

If the `MCBSP_OPEN_RESET` is specified, the MCBSP port registers are set to their power-on defaults and any associated interrupts are disabled and cleared.

Example

```
MCBSP_HANDLE hMcbsp;
...
hMcbsp =
MCBSP_Open(MCBSP_DEV0,MCBSP_OPEN_RESET);
```

4.11.23 MCBSP_PORT_CNT

Compile time constant that holds the number of serial ports present on the current device

Constant `MCBSP_PORT_CNT`

Description Compile time constant that holds the number of serial ports present on the current device.

Example

```
#if (MCBSP_PORT_CNT==3)
  ...
#endif
```

4.11.24 MCBSP_Read

Performs a direct 32-bit read of the data receive register DRR

Function

```
UINT32 MCBSP_Read(
  MCBSP_HANDLE hMcbsp
);
```

Arguments `hMcbsp` Handle to MCBSP port. See `MCBSP_Open()`

Return Value `Data`

Description This function performs a direct 32-bit read of the data receive register DRR.

Example

```
Data = MCBSP_Read(hMcbsp);
```

4.11.25 MCBSP_Reset

Resets the given serial port

Function

```
void MCBSP_Reset(
  MCBSP_HANDLE hMcbsp
);
```

Arguments `hMcbsp` Handle to MCBSP port. See `MCBSP_Open()`

Return Value `none`

Description Resets the given serial port. If you use `INV` for `hMcbsp`, all serial ports are reset.

Actions Taken:

- All serial port registers are set to their power-on defaults.
- All associated interrupts are disabled and cleared

Example

```
MCBSP_Reset(hMcbsp);
MCBSP_Reset(INV);
```

4.11.26 MCBSP_Rfull

Reads the RFULL bit of the serial port control register

Function

```
BOOL MCBSP_Rfull(
  MCBSP_HANDLE hMcbsp
);
```

Arguments `hMcbsp` Handle to MCBSP port. See `MCBSP_Open()`

Return Value `RFULL` Returns RFULL status bit of SPCR register; 0 or 1

Description This function reads the RFULL bit of the serial port control register. A 1 indicates a receive shift register full error.

Example

```
if (MCBSP_Rfull(hMcbsp)) {
  …
}
```

4.11.27 MCBSP_Rrdy

Reads the RRDY status bit of the SPCR register

Function

```
BOOL MCBSP_Rrdy(
  MCBSP_HANDLE hMcbsp
);
```

Arguments `hMcbsp` Handle to MCBSP port. See `MCBSP_Open()`

Return Value `RRDY` Returns RRDY status bit of SPCR; 0 or 1

Description Reads the RRDY status bit of the SPCR register. A 1 indicates the receiver is ready with data to be read.

Example

```
if (MCBSP_Rrdy(hMcbsp)) {
  …
}
```

4.11.28 MCBSP_RsyncErr

Reads the RSYNCERR status bit of the SPCR register

Function

```
BOOL MCBSP_RsyncErr(
  MCBSP_HANDLE hMcbsp
);
```

Arguments

hMcbsp	Handle to MCBSP port. See MCBSP_Open()

Return Value

RSYNCERR	Returns RSYNCERR bit of the SPCR register; 0 or 1

Description

Reads the RSYNCERR status bit of the SPCR register. A 1 indicates a receiver frame sync error.

Example

```
if (MCBSP_ RsyncErr(hMcbsp)) {
  …
}
```

4.11.29 MCBSP_SetPins

Sets the state of the serial port pins when configured as general purpose IO

Function

```
void MCBSP_SetPins(
  MCBSP_HANDLE hMcbsp,
  UINT32 Pins
);
```

Arguments

hMcbsp	Handle to MCBSP port. See MCBSP_Open()
Pins	Bit-mask of pin values (logical OR)

- MCBSP_PIN_CLKX
- MCBSP_PIN_FSX
- MCBSP_PIN_DX
- MCBSP_PIN_CLKR
- MCBSP_PIN_FSR
- MCBSP_PIN_DR
- MCBSP_PIN_CLKS

Return Value

none

Description

Use this function to set the state of the serial port pins when configured as general purpose IO.

Example

```
MCBSP_SetPins(hMcbsp,
  MCBSP_PIN_FSX |
  MCBSP_PIN_DX
);
```

4.11.30 MCBSP_SUPPORT

A compile time constant whose value is 1 if the device supports the MCBSP module

Constant `MCBSP_SUPPORT`

Description Compile time constant that has a value of 1 if the device supports the MCBSP module and 0 otherwise. You are not required to use this constant.

Currently, all devices support this module.

Example

```
#if (MCBSP_SUPPORT)
  /* user MCBSP configuration /
#endif
```

4.11.31 MCBSP_Write

Writes a 32-bit value directly to the serial port data transmit register, DXR

Function

```
void MCBSP_Write(
  MCBSP_HANDLE hMcbsp,
  UINT32 Val
);
```

Arguments

`hMcbsp`	Handle to MCBSP port. See `MCBSP_Open()`
`Val`	32-bit data value

Return Value `none`

Description Use this function to directly write a 32-bit value to the serial port data transmit register, DXR.

Example

```
MCBSP_Write(hMcbsp,0x12345678);
```

4.11.32 MCBSP_Xempty

Reads the XEMPTY bit from the SPCR register

Function

```
BOOL MCBSP_Xempty(
  MCBSP_HANDLE hMcbsp
);
```

Arguments

`hMcbsp`	Handle to MCBSP port. See `MCBSP_Open()`

Return Value

`XEMPTY`	Returns XEMPTY bit of SPCR register; 0 or 1

Description Reads the XEMPTY bit from the SPCR register. A 0 indicates the transmit shift (XSR) is empty.

Example

```
if (MCBSP_Xempty(hMcbsp)) {
  …
}
```

4.11.33 MCBSP_Xrdy

Reads the XRDY status bit of the SPCR register

Function

```
BOOL MCBSP_Xrdy(
  MCBSP_HANDLE hMcbsp
);
```

Arguments

hMcbsp	Handle to MCBSP port. See MCBSP_Open()

Return Value

XRDY	Returns XRDY status bit of SPCR; 0 or 1

Description

Reads the XRDY status bit of the SPCR register. A 1 indicates the transmitter is ready to be written to.

Example

```
if (MCBSP_Xrdy(hMcbsp)) {
  …
}
```

4.11.34 MCBSP_XsyncErr

Reads the XSYNCERR status bit of the SPCR register

Function

```
BOOL MCBSP_XsyncErr(
  MCBSP_HANDLE hMcbsp
);
```

Arguments

hMcbsp	Handle to MCBSP port. See MCBSP_Open()

Return Value

XSYNCERR	Returns XSYNCERR bit of the SPCR register; 0 or 1

Description

Reads the XSYNCERR status bit of the SPCR register. A 1 indicates a transmitter frame sync error.

Example

```
if (MCBSP_ RsyncErr(hMcbsp)) {
  …
}
```

4.12 PWR

4.12.1 PWR_ConfigB

Sets up the power-down logic using the register value passed in

Function

```
void PWR_ConfigB(
  UINT32 pdctl
);
```

Arguments

`pdctl` Power-down control register value

Return Value

`none`

Description

Sets up the power-down logic using the register value passed in.

You may use literal values for the argument or for readability. You may use the *PWR_MK_PDCTL* macro to create the register value based on field values.

Example

```
PWR_ConfigB(0x00000000);
```

4.12.2 PWR_MK_PDCTL

Makes a value suitable for the power-down control register

Macro

```
PWR_MK_PDCTL(
  dma,
  emif,
  mcbsp0,
  mcbsp1,
  mcbsp2
)
```

Arguments

`dma` DMA clock enable:

- `PWR_PDCTL_DMA_CLKON`
- `PWR_PDCTL_DMA_CLKOFF`

`emif` EMIF clock enable:

- `PWR_PDCTL_EMIF_CLKON`
- `PWR_PDCTL_EMIF_CLKOFF`

`mcbsp0` MCBSP0 clock enable:

- `PWR_PDCTL_MCBSP0_CLKON`
- `PWR_PDCTL_MCBSP0_CLKOFF`

`mcbsp1` MCBSP1 clock enable:

- `PWR_PDCTL_MCBSP1_CLKON`
- `PWR_PDCTL_MCBSP1_CLKOFF`

	mcbsp2	MCBSP2 clock enable: • PWR_PDCTL_MCBSP2_CLKON • PWR_PDCTL_MCBSP2_CLKOFF
Return Value	PDCTL Value	Constructed register value

Description Use this macro to make a value suitable for the power-down control register.

The power-on default value is PWR_PDCTL_DEFAULT.

Use of the *PWR_MK_PDCTL* macro makes it easier to construct register values based on field values. You have a choice of using integer constants, integer variables, or the symbolic constants for arguments. All field values are right justified.

Refer to the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for descriptions of the arguments.

Example

```
UINT32 PdCtl;

/* you can do this /
PdCtl = PWR_MK_PDCTL(0,0,0,0,0);

/ or to be more readable, you can do this */
PdCtl = PWR_MK_PDCTL(
  PWR_PDCTL_DMA_CLKON,
  PWR_PDCTL_EMIF_CLKON,
  PWR_PDCTL_MCBSP0_CLKON,
  PWR_PDCTL_MCBSP1_CLKON,
  PWR_PDCTL_MCBSP2_CLKON
);
```

4.12.3 PWR_PowerDown

Forces the DSP to enter a power-down state

Function

```
void PWR_PowerDown(
  PWR_MODE mode
);
```

Arguments	mode	Power-down mode: • PWR_NONE • PWR_PD1A • PWR_PD1B • PWR_PD2 • PWR_PD3 • PWR_IDLE

Return Value none

Description Calling this function forces the DSP to enter a power-down state. Refer to the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for a description of the power-down modes.

Example

```
PWR_PowerDown(PWR_PD2);
```

4.12.4 PWR_SUPPORT

A compile time constant whose value is 1 if the device supports the PWR module

Constant PWR_SUPPORT

Description Compile time constant that has a value of 1 if the device supports the PWR module and 0 otherwise. You are not required to use this constant.

Currently, all devices support this module.

Example

```
#if (PWR_SUPPORT)
  /* user PWR configuration /
#endif
```

4.13 STDINC

4.13.1 BOOL

Typedef BOOL

Description

```
typedef unsigned int BOOL;
```

4.13.2 FALSE

Constant FALSE

Description

```
#ifndef FALSE
  #define FALSE  ((BOOL)(0))
#endif
```

4.13.3 INT16

Typedef INT16

Description

```
typedef short UINT16;
```

4.13.4 INT32

Typedef INT32

Description

```
typedef int INT32;
```

4.13.5 INT40

Typedef INT40

Description

```
typedef long INT40;
```

4.13.6 INT8

Typedef INT8

Description

```
typedef char INT8;
```

4.13.7 INV

Constant INV

Description

```
#define INV  ((void*)(–1))
```

4.13.8 NO

Constant NO

Description
```
#define NO   ((BOOL)(0))
```

4.13.9 TRUE

Constant TRUE

Description
```
#ifndef TRUE
  #define TRUE   ((BOOL)(1))
#endif
```

4.13.10 UINT16

Typedef UINT16

Description
```
typedef unsigned short UINT16;
```

4.13.11 UINT32

Typedef UINT32

Description
```
typedef unsigned int UINT32;
```

4.13.12 UINT40

Typedef UINT40

Description
```
typedef unsigned long UINT40;
```

4.13.13 UINT8

Typedef UINT8

Description
```
typedef unsigned char UINT8;
```

4.13.14 YES

Constant YES

Description
```
#define YES  ((BOOL)(1))
```

4.14 TIMER

4.14.1 TIMER_Close

Closes a previously opened timer device

Function

```
void TIMER_Close(
  TIMER_HANDLE hTimer
);
```

Arguments

`hTimer`	Device handle. See `TIMER_Open()`.

Return Value

`none`

Description

Closes a previously opened timer device. See `TIMER_Open()`.

The Following Tasks are Performed:

- The timer event is disabled and cleared
- The timer registers are set to their default values

Example

```
TIMER_Close(hTimer);
```

4.14.2 TIMER_CONFIG

Structure used to setup a timer device

Structure

`TIMER_CONFIG`

Members

`UINT32 ctl`	Control register value
`UINT32 prd`	Period register value
`UINT32 cnt`	Count register value

Description

This is the TIMER configuration structure used to set up a timer device. You create and initialize this structure and then pass its address to the `TIMER_ConfigA()` function. You can use literal values or the *TIMER_MK* macros to create the structure member values.

Example

```
TIMER_CONFIG MyConfig = {
  0x000002C0, /* ctl */
  0x00010000, /* prd */
  0x00000000  /* cnt */
};
...
TIMER_ConfigA(hTimer,&MyConfig);
```

4.14.3 TIMER_ConfigA

Configure timer using configuration structure

Function

```
void TIMER_ConfigA(
  TIMER_HANDLE hTimer,
  TIMER_CONFIG *Config
);
```

Arguments

`hTimer`	Device handle. See `TIMER_Open()`.
`Config`	

Return Value

`none`

Description

Sets up the timer device using the configuration structure. The values of the structure are written to the DMA registers. The timer control register (CTL) is written last. See also `TIMER_ConfigB()` and `TIMER_CONFIG`.

Example

```
TIMER_CONFIG MyConfig = {
  0x000002C0, /* ctl */
  0x00010000, /* prd */
  0x00000000  /* cnt */
};
...
TIMER_ConfigA(hTimer,&MyConfig);
```

4.14.4 TIMER_ConfigB

Sets up the timer using the register values passed in

Function

```
void TIMER_ConfigB(
  TIMER_HANDLE hTimer,
  UINT32 ctl,
  UINT32 prd,
  UINT32 cnt
);
```

Arguments

`hTimer`	Device handle. See `TIMER_Open()`.
`ctl`	Control register value
`prd`	Period register value
`cnt`	Count register value

Return Value

`none`

Description Sets up the timer using the register values passed in. The register values are written to the timer registers. The timer control register (*ctl*) is written last. See also `TIMER_ConfigA().`

You may use literal values for the arguments or for readability. You may use the *TIMER_MK* macros to create the register values based on field values.

Example

```
TIMER_ConfigB (LTimer, 0x000002C0, 0x00010000,
0x00000000);
```

4.14.5 TIMER_DEVICE_CNT

A compile time constant; number of timer devices present.

Constant `TIMER_DEVICE_CNT`

Description Compile time constant; number of timer devices present.

4.14.6 TIMER_GetCount

Returns the current timer count value

Function

```
UINT32 TIMER_GetCount(
  TIMER_HANDLE hTimer
);
```

Arguments

`hTimer`	Device handle. See `TIMER_Open()`.

Return Value `Count Value`

Description Returns the current timer count value.

Example

```
cnt = TIMER_GetCount(hTimer);
```

4.14.7 TIMER_GetDatin

Reads the value of the TINP pin

Function

```
int TIMER_GetDatin(
  TIMER_HANDLE hTimer
);
```

Arguments

`hTimer`	Device handle. See `TIMER_Open()`.

Return Value

`DATIN`	Returns DATIN, value on TINP pin; 0 or 1

Description This function reads the value of the TINP pin.

Example

```
tinp = TIMER_GetDatin();
```

4.14.8 TIMER_GetEventId

Obtains the event ID for the timer device

Function

```
UINT32 TIMER_GetEventId(
  TIMER_HANDLE hTimer
);
```

Arguments	`hTimer`	Device handle. See `TIMER_Open()`.
Return Value	`Event ID`	IRQ Event ID for the timer device

Description Use this function to obtain the event ID for the timer device.

Example

```
TimerEventId = TIMER_GetEventId(hTimer);
IRQ_Enable(TimerEventId);
```

4.14.9 TIMER_GetPeriod

Returns the period of the timer device

Function

```
UINT32 TIMER_GetPeriod(
  TIMER_HANDLE hTimer
);
```

Arguments	`hTimer`	Device handle. See `TIMER_Open()`.
Return Value	`Period Value`	Timer period

Description Returns the period of the timer device.

Example

```
p = TIMER_GetPeriod(hTimer);
```

4.14.10 TIMER_GetTstat

Reads the timer status; value of timer output

Function

```
int TIMER_GetTstat(
  TIMER_HANDLE hTimer
);
```

Arguments	`hTimer`	Device handle. See `TIMER_Open()`.
Return Value	`TSTAT`	Timer status; 0 or 1

Description Reads the timer status; value of timer output.

Example

```
status = TIMER_GetTstat(hTimer);
```

4.14.11 TIMER_MK_CTL

Makes a value suitable for the timer control register

Macro

```
TIMER_MK_CTL(
  func,
  invout,
  datout,
  pwid,
  go,
  hld,
  cp,
  clksrc,
  invinp
)
```

Arguments

`func` — Function of TOUT pin:

- `TIMER_CTL_FUNC_GPIO`
- `TIMER_CTL_FUNC_TOUT`

`invout` — TOUT inverter control:

- `TIMER_CTL_INVOUT_NO`
- `TIMER_CTL_INVOUT_YES`

`datout` — Data output:

- `TIMER_CTL_DATOUT_0`
- `TIMER_CTL_DATOUT_1`

`pwid` — Pulse width:

- `TIMER_CTL_PWID_ONE`
- `TIMER_CTL_PWID_TWO`

`go` — GO bit:

- `TIMER_CTL_GO_NO`
- `TIMER_CTL_GO_YES`

`hld` — Hold:

- `TIMER_CTL_HLD_YES`
- `TIMER_CTL_HLD_NO`

`cp` — Clock/Pulse mode:

- `TIMER_CTL_CP_PULSE`
- `TIMER_CTL_CP_CLOCK`

	`clksrc`	Timer input clock source: • `TIMER_CTL_CLKSRC_EXTERNAL` • `TIMER_CTL_CLKSRC_CPU0VR4`
	`invinp`	TINP inverter control: • `TIMER_CTL_INVINP_NO` • `TIMER_CTL_INVINP_YES`
Return Value	`CTL Value`	Constructed register value

Description Use this macro to make a value suitable for the timer control register.

The power-on default value is `TIMER_CTL_DEFAULT.`

Use of the *TIMER_MK* macros makes it easier to construct register values based on field values. You have a choice of using integer constants, integer variables, or the symbolic constants for arguments. All field values are right justified.

Refer to the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) for descriptions of the arguments.

Example

```
UINT32 Ctl;

/* you can do this /
Ctl = TIMER_MK_CTL(0,0,0,0,0,0,0,0,0);

/ or to be more readable, you can do this */
Ctl = TIMER_MK_CTL(
  TIMER_CTL_FUNC_GPIO,
  TIMER_CTL_INVOUT_NO,
  TIMER_CTL_DATOUT_0,
  TIMER_CTL_PWID_ONE,
  TIMER_CTL_GO_NO,
  TIMER_CTL_HLD_YES,
  TIMER_CTL_CP_PULSE,
  TIMER_CTL_CLKSRC_EXTERNAL,
  TIMER_CTL_INVINP_NO
);
```

4.14.12 TIMER_Open

Opens a TIMER device for use

Function

```
TIMER_HANDLE TIMER_Open(
  int DevNum,
  UINT32 Flags
);
```

Arguments

`DevNum`	Device Number: • `TIMER_DEVANY` • `TIMER_DEV0` • `TIMER_DEV1`
`Flags`	Open flags, logical OR of any of the following: • `TIMER_OPEN_RESET`

Return Value

`Device Handle`	Device handle

Description

Before a TIMER device can be used, it must first be opened by this function. Once opened, it cannot be opened again until closed. See `TIMER_Close()`.The return value is a unique device handle that is used in subsequent TIMER API calls. If the open fails, `INV` is returned.

If the `TIMER_OPEN_RESET` is specified, the timer device registers are set to their power-on defaults and any associated interrupts are disabled and cleared.

Example

```
TIMER_HANDLE hTimer;
...
hTimer = TIMER_Open(TIMER_DEV0,0);
```

4.14.13 TIMER_Pause

Pauses the timer

Function

```
void TIMER_Pause(
  TIMER_HANDLE hTimer
);
```

Arguments

`hTimer`	Device handle. See `TIMER_Open()`.

Return Value

`none`

Description

Pauses the timer. May be restarted using `TIMER_Resume()`.

Example

```
TIMER_Pause(hTimer);
...
TIMER_Resume(hTimer);
```

4.14.14 TIMER_Reset

Resets the timer device

Function

```
void TIMER_Reset(
  TIMER_HANDLE hTimer
);
```

Arguments

`hTimer`	Device handle. See `TIMER_Open()`.

Return Value

`none`

Description

Resets the timer device. Disables and clears the interrupt event and sets the timer registers to default values. If `INV` is specified, all timer devices are reset.

Example

```
TIMER_Reset(hTimer);
TIMER_Reset(INV);
```

4.14.15 TIMER_Resume

Resumes the timer after a pause

Function

```
void TIMER_Resume(
  TIMER_HANDLE hTimer
);
```

Arguments

`hTimer`	Device handle. See `TIMER_Open()`.

Return Value

`none`

Description

Resumes the timer after a pause. See `TIMER_Pause()`.

Example

```
TIMER_Pause(hTimer);
...
TIMER_Resume(hTimer);
```

4.14.16 TIMER_SetCount

Sets the count value of the timer

Function

```
void TIMER_SetCount(
  TIMER_HANDLE hTimer,
  UINT32 Count
);
```

Arguments

`hTimer`	Device handle. See `TIMER_Open()`.
`Count`	Count value

Return Value

`none`

Description

Sets the count value of the timer. The timer is not paused during the update.

Example

```
TIMER_SetCount(hTimer,0x00000000);
```

4.14.17 TIMER_SetDataout

Sets the data output value

Function

```
void TIMER_SetDatout(
  TIMER_HANDLE hTimer,
  int Val
);
```

Arguments

`hTimer`	Device handle. See `TIMER_Open()`.
`Val`	0 or 1

Return Value `none`

Description Sets the data output value.

Example

```
TIMER_SetDatout(hTimer,0);
```

4.14.18 TIMER_SetPeriod

Sets the timer period

Function

```
void TIMER_SetPeriod(
  TIMER_HANDLE hTimer,
  UINT32 Period
);
```

Arguments

`hTimer`	Device handle. See `TIMER_Open()`.
`Period`	Period value

Return Value `none`

Description Sets the timer period. The timer is not paused during the update.

Example

```
TIMER_SetPeriod(hTimer,0x00010000);
```

4.14.19 TIMER_Start

Starts the timer device running

Function

```
void TIMER_Start(
  TIMER_HANDLE hTimer
);
```

Arguments

`hTimer`	Device handle. See `TIMER_Open()`.

Return Value `none`

Description Starts the timer device running. HLD is released and GO is set.

Example

```
TIMER_Start(hTimer);
```

4.14.20 TIMER_SUPPORT

A compile time constant whose value is 1 if the device supports the TIMER module

Constant `TIMER_SUPPORT`

Description Compile time constant that has a value of 1 if the device supports the TIMER module and 0 otherwise. You are not required to use this constant.

Currently, all devices support this module.

Example

```
#if (TIMER_SUPPORT)
  /* user TIMER configuration /
#endif
```

Chapter 5

HAL Reference

This chapter contains an alphabetical reference of the chip support library hardware abstraction layer (CSL HAL). The HAL underlies the service layer and provides it a set of macros and constants for manipulating the peripheral registers at the lowest level.

When using the service layer APIs, it is not advisable to interface directly into the HAL because this could have an adverse effect on the service layer functionality. However, if you decide not to use the service layer, the HAL is available for exclusive use. It is a good idea to have the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190) readily available when viewing the HAL reference.

Topic **Page**

5.1 HAL Reference Introduction

The HAL offers a consistent orthogonal set of constants and macros used for manipulating peripheral registers symbolically. For every register and every field of every register, there are access identifiers as shown below.

Memory Mapped Register Constants

- `HPER_REG_ADDR`

Memory Mapped Register Macros

- `HPER_REG`
- `HPER_REG_GET(RegAddr)`
- `HPER_REG_SET(RegAddr, Val)`
- `HPER_REG_CFG(RegAddr, Field Val0, Field Val1, …)`

Memory Mapped Register Field Constants

- `HPER_REG_FIELD_MASK`
- `HPER_REG_FIELD_SHIFT`

Memory Mapped Register Field Macros

- `HPER_REG_FIELD_GET(RegAddr)`
- `HPER_REG_FIELD_SET(RegAddr, Val)`

HPER – peripheral module name preceded by H, i.e. HDMA.

REG – peripheral register name, i.e. PRICTL

FIELD – peripheral register field name, i.e. ESIZE

HPER_REG_ADDR – constant 32-bit address of the register

HPER_REG – L-value symbol for the register that may be used in expressions, defined as (*(volatile unsigned int*)HPER_REG_ADDR)

HPER_REG_GET() – reads the register

HPER_REG_SET() – sets the register to a given value

HPER_REG_CFG() – sets the register given individual field values

HPER_REG_FILED_MASK – is a bit-mask defining the field within the register

HPER_REG_FIELD_SHIFT – bit position of the field within the register

HPER_REG_FIELD_GET() – extracts the field from the register then right jus tifies it

HPER_REG_FIELD_SET() – sets the field of the register to the given right justified value

Example Usage:

- `HDMA_PRICTL = 0x1234567;`
- `HDMA_PRICTL_RSYNC_SET(HDMA_PRICTL0_ADDR,12);`
- `X = HDMA_PRICTL_RSYNC_GET(HDMA_PRICTL0_ADDR);`
- `X = HDMA_PRICTL_GET(HDMA_PRICTL0_ADDR);`
- `HDMA_PRICTL_SET(HDMA_PRICTL0_ADDR,0x12345678);`
- ```
  HDMA_PRICTL_CFG(HDMA_PRICTL0_ADDR,
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
  );
  ```

In the reference that follows, only the symbolic identifiers for the registers and fields are given. The macros and constants may be assumed as follows:

- `HPER_REG_ADDR – always exists`
- `HPER_REG – always exists`
- `HPER_REG_GET – exists if register is readable`
- `HPER_REG_SET – exists if register is write–able`
- `HPER_REG_CFG – exists if register is write–able, only write-able fields are parameters to this macro`
- `HPER_REG_FIELD_MASK – always exists`
- `HPER_REG_FIELD_SHIFT – always exists`
- `HPER_REG_FIELD_GET – exists if field is readable`
- `HPER_REG_FIELD_SET – exists if field is write–able`

The service layer source code uses the HAL extensively to perform its tasks. One of the things the HAL does for the service layer is abstracts certain differences of registers between different devices. For example, if a field becomes larger in a future device, the MASK/SHIFT macros will be adjusted accordingly and hence, the GET/SET macros still work. It is a good idea to have the *TMS320C6000 Peripherals Reference Guide* handy when viewing the HAL reference.
  ```

5.2 HCACHE

5.2.1 HCACHE_CCFG

cache configuration register

(RW) HCACHE_CCFG*		
Fields	(RW) HCACHE_CCFG_L2MODE	L2 operation mode
	(RW) HCACHE_CCFG_ID	invalidate L1D
	(RW) HCACHE_CCFG_IP	invalidate L1P
	(RW) HCACHE_CCFG_P	L2 requestor priority
*Only supported on devices with L2 cache		

5.2.2 HCACHE_L2FBAR

L2 cache flush base address register

(RW) HCACHE_L2FBAR*		
Fields	(RW) HCACHE_L2FBAR_L2FBAR	L2 flush base address
*Only supported on devices with L2 cache		

5.2.3 HCACHE_L2FWC

L2 cache flush word count register

(RW) HCACHE_L2FWC*		
Fields	(RW) HCACHE_L2FWC_L2FWC	L2 flush word count
*Only supported on devices with L2 cache		

5.2.4 HCACHE_L2CBAR

L2 cache clean base address register

(RW) HCACHE_L2CBAR*		
Fields	(RW) HCACHE_L2CBAR_L2CBAR	L2 clean base address
*Only supported on devices with L2 cache		

5.2.5 HCACHE_L2CWC

L2 cache clean word count

(RW) HCACHE_L2CWC*		
Fields	(RW) HCACHE_L2CWC_L2CWC	L2 clean word count
*Only supported on devices with L2 cache		

5.2.6 HCACHE_L1PFBAR — *L1 program cache flush base address register*

(RW) HCACHE_L1PFBAR*		
Fields	(RW) HCACHE_L1PFBAR_L1PFBAR	L1P flush base address
*Only supported on devices with L2 cache		

5.2.7 HCACHE_L1PFWC — *L1 program cache flush word count register*

(RW) HCACHE_L1PFWC*		
Fields	(RW) HCACHE_L1PFWC_L1PFWC	L1P flush word count
*Only supported on devices with L2 cache		

5.2.8 HCACHE_L1DFBAR — *L1 data cache flush base register*

(RW) HCACHE_L1DFBAR*		
Fields	(RW) HCACHE_L1DFBAR_L1DFBAR	L1D flush base address
*Only supported on devices with L2 cache		

5.2.9 HCACHE_L1DFWC — *L1 data flush word count register*

(RW) HCACHE_L1DFWC*		
Fields	(RW) HCACHE_L1DFWC_L1DFWC	L1D flush word count
*Only supported on devices with L2 cache		

5.2.10 HCACHE_L2FLUSH — *L2 cache flush all register*

(RW) HCACHE_L2FLUSH*		
Fields	(RW) HCACHE_L2FLUSH_F	flush L2
*Only supported on devices with L2 cache		

5.2.11 HCACHE_L2CLEAN — **L2 cache clean all register**

(RW) HCACHE_L2CLEAN*		
Fields	(RW) HCACHE_L2CLEAN_C	clean L2
*Only supported on devices with L2 cache		

5.2.12 HCACHE_MAR

Memory attribute registers 0–15

(RW) HCACHE_MAR0*		
(RW) HCACHE_MAR1*		
(RW) HCACHE_MAR2*		
(RW) HCACHE_MAR3*		
(RW) HCACHE_MAR4*		
(RW) HCACHE_MAR5*		
(RW) HCACHE_MAR6*		
(RW) HCACHE_MAR7*		
(RW) HCACHE_MAR8*		
(RW) HCACHE_MAR9*		
(RW) HCACHE_MAR10*		
(RW) HCACHE_MAR11*		
(RW) HCACHE_MAR12*		
(RW) HCACHE_MAR13*		
(RW) HCACHE_MAR14*		
(RW) HCACHE_MAR15*		
Fields	(RW) HCACHE_MAR_CE	cacheability enable
*Only supported on devices with L2 cache		

5.3 HCHIP

5.3.1 HCHIP_NULL

NULL register

(RW) HCHIP_NULL		
Fields	none	
When a register exists on one device but not on another, the NULL register is used on the non-supporting device. This allows the HAL to still implement the GET/SET macros.		

5.3.2 HCHIP_CSR

Control status register

(RW) HCHIP_CSR **(RW) CSR**		
Fields	(RW) HCHIP_CSR_GIE	global interrupt enable
	(RW) HCHIP_CSR_PGIE	previous GIE
	(RW) HCHIP_CSR_DCC	data cache control
	(RW) HCHIP_CSR_PCC	program cache control
	(R) HCHIP_CSR_EN	endian
	(RC) HCHIP_CSR_SAT	saturate bit
	(RW) HCHIP_CSR_PWRD	control power down modes
	(R) HCHIP_CSR_REVID	revision ID
	(R) HCHIP_CSR_CPUID	CPU ID

5.3.3 HCHIP_IFR — *Interrupt flag register*

(R) HCHIP_IFR **(R) IFR**		
Fields	(R) HCHIP_IFR_NMIF	non-maskable interrupt flag
	(R) HCHIP_IFR_IF4	interrupt 4 flag
	(R) HCHIP_IFR_IF5	interrupt 5 flag
	(R) HCHIP_IFR_IF6	interrupt 6 flag
	(R) HCHIP_IFR_IF7	interrupt 7 flag
	(R) HCHIP_IFR_IF8	interrupt 8 flag
	(R) HCHIP_IFR_IF9	interrupt 9 flag
	(R) HCHIP_IFR_IF10	interrupt 10 flag
	(R) HCHIP_IFR_IF11	interrupt 11 flag
	(R) HCHIP_IFR_IF12	interrupt 12 flag
	(R) HCHIP_IFR_IF13	interrupt 13 flag
	(R) HCHIP_IFR_IF14	interrupt 14 flag
	(R) HCHIP_IFR_IF15	interrupt 15 flag

5.3.4 HCHIP_ISR — *Interrupt set register*

(W) HCHIP_ISR **(W) ISR**		
Fields	(W) HCHIP_ISR_IS4	interrupt 4 set
	(W) HCHIP_ISR_IS5	interrupt 5 set
	(W) HCHIP_ISR_IS6	interrupt 6 set
	(W) HCHIP_ISR_IS7	interrupt 7 set
	(W) HCHIP_ISR_IS8	interrupt 8 set
	(W) HCHIP_ISR_IS9	interrupt 9 set
	(W) HCHIP_ISR_IS10	interrupt 10 set
	(W) HCHIP_ISR_IS11	interrupt 11 set
	(W) HCHIP_ISR_IS12	interrupt 12 set
	(W) HCHIP_ISR_IS13	interrupt 13 set
	(W) HCHIP_ISR_IS14	interrupt 14 set
	(W) HCHIP_ISR_IS15	interrupt 15 set

5.3.5 HCHIP_ICR

Interrupt clear register

(W) HCHIP_ICR **(W) ICR**		
Fields	(W) HCHIP_ICR_IC4	interrupt 4 clear
	(W) HCHIP_ICR_IC5	interrupt 5 clear
	(W) HCHIP_ICR_IC6	interrupt 6 clear
	(W) HCHIP_ICR_IC7	interrupt 7 clear
	(W) HCHIP_ICR_IC8	interrupt 8 clear
	(W) HCHIP_ICR_IC9	interrupt 9 clear
	(W) HCHIP_ICR_IC10	interrupt 10 clear
	(W) HCHIP_ICR_IC11	interrupt 11 clear
	(W) HCHIP_ICR_IC12	interrupt 12 clear
	(W) HCHIP_ICR_IC13	interrupt 13 clear
	(W) HCHIP_ICR_IC14	interrupt 14 clear
	(W) HCHIP_ICR_IC15	interrupt 15 clear

5.3.6 HCHIP_IER

Interrupt enable register

(RW) HCHIP_IER **(RW) IER**		
Fields	(RW) HCHIP_IER_NMIF	non-maskable interrupt enable
	(RW) HCHIP_IER_IE4	interrupt 4 enable
	(RW) HCHIP_IER_IE5	interrupt 5 enable
	(RW) HCHIP_IER_IE6	interrupt 6 enable
	(RW) HCHIP_IER_IE7	interrupt 7 enable
	(RW) HCHIP_IER_IE8	interrupt 8 enable
	(RW) HCHIP_IER_IE9	interrupt 9 enable
	(RW) HCHIP_IER_IE10	interrupt 10 enable
	(RW) HCHIP_IER_IE11	interrupt 11 enable
	(RW) HCHIP_IER_IE12	interrupt 12 enable
	(RW) HCHIP_IER_IE13	interrupt 13 enable
	(RW) HCHIP_IER_IE14	interrupt 14 enable
	(RW) HCHIP_IER_IE15	interrupt 15 enable

5.3.7 HCHIP_ISTP — *Interrupt service table pointer register*

(RW) HCHIP_ISTP **(RW) ISTP**		
Fields	(R) HCHIP_ISTP_HPEINT	highest priority enabled interrupt
	(RW) HCHIP_ISTP_ISTB	interrupt service table base address

5.3.8 HCHIP_IRP — *Interrupt return pointer register*

(RW) HCHIP_IRP **(RW) IRP**		
Fields	(RW) HCHIP_IRP_IRP	interrupt return pointer

5.3.9 HCHIP_NRP — *Non-maskable interrupt return pointer register*

(RW) HCHIP_NRP **(RW) NRP**		
Fields	(RW) HCHIP_NRP_NRP	non-maskable interrupt return pointer

5.3.10 HCHIP_AMR — *Addressing mode register*

(RW) HCHIP_AMR **(RW) AMR**		
Fields	(RW) HCHIP_AMR_A4MODE	A4 mode
	(RW) HCHIP_AMR_A5MODE	A5 mode
	(RW) HCHIP_AMR_A6MODE	A6 mode
	(RW) HCHIP_AMR_A7MODE	A7 mode
	(RW) HCHIP_AMR_B4MODE	B4 mode
	(RW) HCHIP_AMR_B5MODE	B5 mode
	(RW) HCHIP_AMR_B6MODE	B6 mode
	(RW) HCHIP_AMR_B7MODE	B7 mode
	(RW) HCHIP_AMR_BK0	BK0 block size
	(RW) HCHIP_AMR_BK1	BK1 block size

5.3.11 HCHIP_FADCR

Floating-point adder configuration register

(RW) HCHIP_FADCR* (RW) FADCR*		
Fields	(RW) HCHIP_FADCR_L1NAN1	NAN1.L1
	(RW) HCHIP_FADCR_L1NAN2	NAN2.L1
	(RW) HCHIP_FADCR_L1DEN1	DEN1.L1
	(RW) HCHIP_FADCR_L1DEN2	DEN2.L1
	(RW) HCHIP_FADCR_L1INVAL	INVAL.L1
	(RW) HCHIP_FADCR_L1INFO	INFO.L1
	(RW) HCHIP_FADCR_L1OVER	OVER.L1
	(RW) HCHIP_FADCR_L1INEX	INEX.L1
	(RW) HCHIP_FADCR_L1UNDER	UNDER.L1
	(RW) HCHIP_FADCR_L1RMODE	Rmode.L1
	(RW) HCHIP_FADCR_L2NAN1	NAN1.L2
	(RW) HCHIP_FADCR_L2NAN2	NAN2.L2
	(RW) HCHIP_FADCR_L2DEN1	DEN1.L2
	(RW) HCHIP_FADCR_L2DEN2	DEN2.L2
	(RW) HCHIP_FADCR_L2INVAL	INVAL.L2
	(RW) HCHIP_FADCR_L2INFO	INFO.L2
	(RW) HCHIP_FADCR_L2OVER	OVER.L2
	(RW) HCHIP_FADCR_L2INEX	INEX.L2
	(RW) HCHIP_FADCR_L2UNDER	UNDER.L2
	(RW) HCHIP_FADCR_L2RMODE	Rmode.L2
* only supported on devices with floating-point unit		

5.3.12 HCHIP_FAUCR — *Floating-point auxiliary configuration register*

(RW) HCHIP_FAUCR* (RW) FAUCR*		
Fields	(RW) HCHIP_FAUCR_S1NAN1	NAN1.S1
	(RW) HCHIP_FAUCR_S1NAN2	NAN2.S1
	(RW) HCHIP_FAUCR_S1DEN1	DEN1.S1
	(RW) HCHIP_FAUCR_S1DEN2	DEN2.S1
	(RW) HCHIP_FAUCR_S1INVAL	INVAL.S1
	(RW) HCHIP_FAUCR_S1INFO	INFO.S1
	(RW) HCHIP_FAUCR_S1OVER	OVER.S1
	(RW) HCHIP_FAUCR_S1INEX	INEX.S1
	(RW) HCHIP_FAUCR_S1UNDER	UNDER.S1
	(RW) HCHIP_FAUCR_S1UNORD	UNORD.S1
	(RW) HCHIP_FAUCR_S1DIV0	DIV0.S1
	(RW) HCHIP_FAUCR_S2NAN1	NAN1.S2
	(RW) HCHIP_FAUCR_S2NAN2	NAN2.S2
	(RW) HCHIP_FAUCR_S2DEN1	DEN1.S2
	(RW) HCHIP_FAUCR_S2DEN2	DEN2.S2
	(RW) HCHIP_FAUCR_S2INVAL	INVAL.S2
	(RW) HCHIP_FAUCR_S2INFO	INFO.S2
	(RW) HCHIP_FAUCR_S2OVER	OVER.S2
	(RW) HCHIP_FAUCR_S2INEX	INEX.S2
	(RW) HCHIP_FAUCR_S2UNDER	UNDER.S2
	(RW) HCHIP_FAUCR_S2UNORD	UNORD.S2
	(RW) HCHIP_FAUCR_S2DIV0	DIV0.S2
* only supported on devices with floating-point unit		

5.3.13 HCHIP_FMCR

Floating-point multiplier configuration register

(RW) HCHIP_FMCR* (RW) FMCR*		
Fields	(RW) HCHIP_FMCR_M1NAN1	NAN1.M1
	(RW) HCHIP_FMCR_M1NAN2	NAN2.M1
	(RW) HCHIP_FMCR_M1DEN1	DEN1.M1
	(RW) HCHIP_FMCR_M1DEN2	DEN2.M1
	(RW) HCHIP_FMCR_M1INVAL	INVAL.M1
	(RW) HCHIP_FMCR_M1INFO	INFO.M1
	(RW) HCHIP_FMCR_M1OVER	OVER.M1
	(RW) HCHIP_FMCR_M1INEX	INEX.M1
	(RW) HCHIP_FMCR_M1UNDER	UNDER.M1
	(RW) HCHIP_FMCR_M1RMODE	Rmode.M1
	(RW) HCHIP_FMCR_M2NAN1	NAN1.M2
	(RW) HCHIP_FMCR_M2NAN2	NAN2.M2
	(RW) HCHIP_FMCR_M2DEN1	DEN1.M2
	(RW) HCHIP_FMCR_M2DEN2	DEN2.M2
	(RW) HCHIP_FMCR_M2INVAL	INVAL.M2
	(RW) HCHIP_FMCR_M2INFO	INFO.M2
	(RW) HCHIP_FMCR_M2OVER	OVER.M2
	(RW) HCHIP_FMCR_M2INEX	INEX.M2
	(RW) HCHIP_FMCR_M2UNDER	UNDER.M2
	(RW) HCHIP_FMCR_M2RMODE	Rmode.M2
* only supported on devices with floating-point unit		

5.4 HDMA

5.4.1 HDMA_AUXCTL

DMA auxiliary control register

(RW) HDMA_AUXCTL*		
Fields	(RW) HDMA_AUXCTL_CHPRI	DMA channel priority
	(RW) HDMA_AUXCTL_AUXPRI	auxiliary channel priority mode
* only supported on devices with DMA		

5.4.2 HDMA_PRICTL

DMA primary control register

(RW) HDMA_PRICTL0* **(RW) HDMA_PRICTL1*** **(RW) HDMA_PRICTL2*** **(RW) HDMA_PRICTL3***		
Fields	(RW) HDMA_PRICTL_START	start
	(R) HDMA_PRICTL_STATUS	status
	(RW) HDMA_PRICTL_SRCDIR	source address modification
	(RW) HDMA_PRICTL_DSTDIR	destination address modification
	(RW) HDMA_PRICTL_ESIZE	element size
	(RW) HDMA_PRICTL_SPLIT	split channel mode
	(RW) HDMA_PRICTL_CNTRLD	transfer count reload
	(RW) HDMA_PRICTL_INDEX	global data register for programmable index
	(RW) HDMA_PRICTL_RSYNC	read transfer synchronization
	(RW) HDMA_PRICTL_WSYNC	write transfer synchronization
	(RW) HDMA_PRICTL_PRI	priority mode
	(RW) HDMA_PRICTL_TCINT	transfer controller interrupt
	(RW) HDMA_PRICTL_FS	frame synchronization
	(RW) HDMA_PRICTL_EMOD	emulation mode
	(RW) HDMA_PRICTL_SRCRLD	source address reload
	(RW) HDMA_PRICTL_DSTRLD	destination address reload
* only supported on devices with DMA		

5.4.3 HDMA_SECCTL

DMA secondary control register

(RW) HDMA_SECCTL0* (RW) HDMA_SECCTL1* (RW) HDMA_SECCTL2* (RW) HDMA_SECCTL3*		
Fields	(RW) HDMA_SECCTL_SXCOND	split transfer overrun receive condition
	(RW) HDMA_SECCTL_SXIE	split transfer overrun receive interrupt enable
	(RW) HDMA_SECCTL_FRAMECOND	frame complete condition
	(RW) HDMA_SECCTL_FRAMEIE	frame complete interrupt enable
	(RW) HDMA_SECCTL_LASTCOND	last frame condition
	(RW) HDMA_SECCTL_LASTIE	last frame interrupt enable
	(RW) HDMA_SECCTL_BLOCKCOND	block transfer complete condition
	(RW) HDMA_SECCTL_BLOCKIE	block transfer complete interrupt enable
	(RW) HDMA_SECCTL_RDROPCOND	dropped read synchronization condition
	(RW) HDMA_SECCTL_RDROPIE	dropped read synchronization interrupt enable
	(RW) HDMA_SECCTL_WDROPCOND	dropped write synchronization condition
	(RW) HDMA_SECCTL_WDROPIE	dropped write synchronization interrupt enable
	(RW) HDMA_SECCTL_RSYNCSTAT	read synchronization status
	(RW) HDMA_SECCTL_RSYNCCLR	read synchronization status clear
	(RW) HDMA_SECCTL_WSYNCSTAT	write synchronization status
	(RW) HDMA_SECCTL_WSYNCCLR	write synchronization status clear
	(RW) HDMA_SECCTL_DMACEN	DMAC pin control
	(RW) HDMA_SECCTL_FSIG	frame sync ignore
	(RW) HDMA_SECCTL_RSPOL	read sync polarity
	(RW) HDMA_SECCTL_WSPOL	write sync polarity
* only supported on devices with DMA		

5.4.4 HDMA_SRC

DMA source address register

(RW) HDMA_SRC0* (RW) HDMA_SRC1* (RW) HDMA_SRC2* (RW) HDMA_SRC3*		
Fields	(RW) HDMA_SRC_SRC	source address
* only supported on devices with DMA		

5.4.5 HDMA_DST

DMA destination address register

(RW) HDMA_DST0* (RW) HDMA_DST1* (RW) HDMA_DST2* (RW) HDMA_DST3*		
Fields	(RW) HDMA_DST_DST	destination address
* only supported on devices with DMA		

5.4.6 HDMA_XFRCNT

DMA transfer count register

(RW) HDMA_XFRCNT0* (RW) HDMA_XFRCNT1* (RW) HDMA_XFRCNT2* (RW) HDMA_XFRCNT3*		
Fields	(RW) HDMA_XFRCNT_ELECNT	element count
Fields	(RW) HDMA_XFRCNT_FRMCNT	frame count
* only supported on devices with DMA		

5.4.7 HDMA_GBLCNT

DMA global count reload register

(RW) HDMA_GBLCNTA* (RW) HDMA_GBLCNTB*		
Fields	(RW) HDMA_GBLCNT_ELECNT	element count reload
Fields	(RW) HDMA_GBLCNT_FRMCNT	frame count reload
* only supported on devices with DMA		

5.4.8 HDMA_GBLADDR

DMA global address register

(RW) HDMA_GBLADDRA* (RW) HDMA_GBLADDRB* (RW) HDMA_GBLADDRC* (RW) HDMA_GBLADDRD*		
Fields	(RW) HDMA_GBLIDX_ELEIDX	element index
Fields	(RW) HDMA_GBLIDX_FRMIDX	frame index
* only supported on devices with DMA		

5.4.9 HDMA_GBLIDX

DMA global index register

(RW) HDMA_GBLIDXA* (RW) HDMA_GBLIDXB*		
Fields	(RW) HDMA_GBLADDR_GBLADDR	address
* only supported on devices with DMA		

5.5 HEDMA

5.5.1 HEDMA_OPT

quick EDMA options register
quick EDMA options pseudo register
EDMA channel 0 options
EDMA channel 1 options
EDMA channel 2 options
EDMA channel 3 options
EDMA channel 4 options
EDMA channel 5 options
EDMA channel 6 options
EDMA channel 7 options
EDMA channel 8 options
EDMA channel 9 options
EDMA channel 10 options
EDMA channel 11 options
EDMA channel 12 options
EDMA channel 13 options
EDMA channel 14 options
EDMA channel 15 options
EDMA options in parameter RAM

(W) HEDMA_QOPT* **(W) HEDMA_QSOPT*** **(RW) HEDMA_OPT0*** **(RW) HEDMA_OPT1*** **(RW) HEDMA_OPT2*** **(RW) HEDMA_OPT3*** **(RW) HEDMA_OPT4*** **(RW) HEDMA_OPT5*** **(RW) HEDMA_OPT6*** **(RW) HEDMA_OPT7*** **(RW) HEDMA_OPT8*** **(RW) HEDMA_OPT9*** **(RW) HEDMA_OPT10*** **(RW) HEDMA_OPT11*** **(RW) HEDMA_OPT12*** **(RW) HEDMA_OPT13*** **(RW) HEDMA_OPT14*** **(RW) HEDMA_OPT15*** **(RW) *PRAM*+***	

Fields	(RW) HEDMA_OPT_FS	frame synchronization
	(RW) HEDMA_OPT_LINK++	linking events
	(RW) HEDMA_OPT_TCC	transfer complete code
	(RW) HEDMA_OPT_TCINT	transfer complete interrupt
	(RW) HEDMA_OPT_DUM	destination update mode
	(RW) HEDMA_OPT_2DD	2-dimensional destination
	(RW) HEDMA_OPT_SUM	source update mode

Table Continued

	(RW) HEDMA_OPT_2DS	2-dimensional source
	(RW) HEDMA_OPT_ESIZE	element size
	(RW) HEDMA_OPT_PRI	priority level

* only supported on devices with EDMA

+ all of the macros apply for the options member of a PRAM table for both read and write

++ this field ignored for quick DMA registers

5.5.2 HEDMA_SRC

quick EDMA source address register
quick EDMA source address pseudo register
EDMA channel 0 source address
EDMA channel 1 source address
EDMA channel 2 source address
EDMA channel 3 source address
EDMA channel 4 source address
EDMA channel 5 source address
EDMA channel 6 source address
EDMA channel 7 source address
EDMA channel 8 source address
EDMA channel 9 source address
EDMA channel 10 source address
EDMA channel 11 source address
EDMA channel 12 source address
EDMA channel 13 source address
EDMA channel 14 source address
EDMA channel 15 source address
EDMA source address in parameter RAM

(W) HEDMA_QSRC* (W) HEDMA_QSSRC* (RW) HEDMA_SRC0* (RW) HEDMA_SRC1* (RW) HEDMA_SRC2* (RW) HEDMA_SRC3* (RW) HEDMA_SRC4* (RW) HEDMA_SRC5* (RW) HEDMA_SRC6* (RW) HEDMA_SRC7* (RW) HEDMA_SRC8* (RW) HEDMA_SRC9* (RW) HEDMA_SRC10* (RW) HEDMA_SRC11* (RW) HEDMA_SRC12* (RW) HEDMA_SRC13* (RW) HEDMA_SRC14* (RW) HEDMA_SRC15* (RW) *PRAM*+*		
Fields	(RW) HEDMA_SRC_SRC	source address
* only supported on devices with EDMA + all of the macros apply for the options member of a PRAM table for both read and write		

5.5.3 HEDMA_CNT

quick EDMA transfer count register
quick EDMA transfer count pseudo register
EDMA channel 0 transfer count
EDMA channel 1 transfer count
EDMA channel 2 transfer count
EDMA channel 3 transfer count
EDMA channel 4 transfer count
EDMA channel 5 transfer count
EDMA channel 6 transfer count
EDMA channel 7 transfer count
EDMA channel 8 transfer count
EDMA channel 9 transfer count
EDMA channel 10 transfer count
EDMA channel 11 transfer count
EDMA channel 12 transfer count
EDMA channel 13 transfer count
EDMA channel 14 transfer count
EDMA channel 15 transfer count
EDMA transfer count in parameter RAM

(W) HEDMA_QCNT* **(W) HEDMA_QSCNT*** **(RW) HEDMA_CNT0*** **(RW) HEDMA_CNT1*** **(RW) HEDMA_CNT2*** **(RW) HEDMA_CNT3*** **(RW) HEDMA_CNT4*** **(RW) HEDMA_CNT5*** **(RW) HEDMA_CNT6*** **(RW) HEDMA_CNT7*** **(RW) HEDMA_CNT8*** **(RW) HEDMA_CNT9*** **(RW) HEDMA_CNT10*** **(RW) HEDMA_CNT11*** **(RW) HEDMA_CNT12*** **(RW) HEDMA_CNT13*** **(RW) HEDMA_CNT14*** **(RW) HEDMA_CNT15*** **(RW) *PRAM*+***		
Fields	(RW) HEDMA_CNT_ELECNT	element count
	(RW) HEDMA_CNT_FRMCNT	frame count
* only supported on devices with EDMA + all of the macros apply for the options member of a PRAM table for both read and write		

5.5.4 HEDMA_DST

quick EDMA destination address register
quick EDMA destination address pseudo register
EDMA channel 0 destination address
EDMA channel 1 destination address
EDMA channel 2 destination address
EDMA channel 3 destination address
EDMA channel 4 destination address
EDMA channel 5 destination address
EDMA channel 6 destination address
EDMA channel 7 destination address
EDMA channel 8 destination address
EDMA channel 9 destination address
EDMA channel 10 destination address
EDMA channel 11 destination address
EDMA channel 12 destination address
EDMA channel 13 destination address
EDMA channel 14 destination address
EDMA channel 15 destination address
EDMA destination address in parameter RAM

(W) HEDMA_QDST* **(W) HEDMA_QSDST*** **(RW) HEDMA_DST0*** **(RW) HEDMA_DST1*** **(RW) HEDMA_DST2*** **(RW) HEDMA_DST3*** **(RW) HEDMA_DST4*** **(RW) HEDMA_DST5*** **(RW) HEDMA_DST6*** **(RW) HEDMA_DST7*** **(RW) HEDMA_DST8*** **(RW) HEDMA_DST9*** **(RW) HEDMA_DST10*** **(RW) HEDMA_DST11*** **(RW) HEDMA_DST12*** **(RW) HEDMA_DST13*** **(RW) HEDMA_DST14*** **(RW) HEDMA_DST15*** **(RW) *PRAM*+**	

Fields	(RW) HEDMA_DST_DST	destination address

* only supported on devices with EDMA

+ all of the macros apply for the options member of a PRAM table for both read and write

5.5.5 HEDMA_IDX

quick EDMA index register
quick EDMA index pseudo register
EDMA channel 0 index
EDMA channel 1 index
EDMA channel 2 index
EDMA channel 3 index
EDMA channel 4 index
EDMA channel 5 index
EDMA channel 6 index
EDMA channel 7 index
EDMA channel 8 index
EDMA channel 9 index
EDMA channel 10 index
EDMA channel 11 index
EDMA channel 12 index
EDMA channel 13 index
EDMA channel 14 index
EDMA channel 15 index
EDMA index in parameter RAM

(W) HEDMA_QIDX* **(W) HEDMA_QSIDX*** **(RW) HEDMA_IDX0*** **(RW) HEDMA_IDX1*** **(RW) HEDMA_IDX2*** **(RW) HEDMA_IDX3*** **(RW) HEDMA_IDX4*** **(RW) HEDMA_IDX5*** **(RW) HEDMA_IDX6*** **(RW) HEDMA_IDX7*** **(RW) HEDMA_IDX8*** **(RW) HEDMA_IDX9*** **(RW) HEDMA_IDX10*** **(RW) HEDMA_IDX11*** **(RW) HEDMA_IDX12*** **(RW) HEDMA_IDX13*** **(RW) HEDMA_IDX14*** **(RW) HEDMA_IDX15*** **(RW) *PRAM*+***		
Fields	(RW) HEDMA_IDX_ELEIDX	element index
	(RW) HEDMA_IDX_FRMIDX	frame index
* only supported on devices with EDMA + all of the macros apply for the options member of a PRAM table for both read and write		

5.5.6 HEDMA_RLD

EDMA channel 0 element reload & link
EDMA channel 1 element reload & link
EDMA channel 2 element reload & link
EDMA channel 3 element reload & link
EDMA channel 4 element reload & link
EDMA channel 5 element reload & link
EDMA channel 6 element reload & link
EDMA channel 7 element reload & link
EDMA channel 8 element reload & link
EDMA channel 9 element reload & link
EDMA channel 10 element reload & link
EDMA channel 11 element reload & link
EDMA channel 12 element reload & link
EDMA channel 13 element reload & link
EDMA channel 14 element reload & link
EDMA channel 15 element reload & link
EDMA element reload & link in parameter RAM

(RW) HEDMA_RLD0* **(RW) HEDMA_RLD1*** **(RW) HEDMA_RLD2*** **(RW) HEDMA_RLD3*** **(RW) HEDMA_RLD4*** **(RW) HEDMA_RLD5*** **(RW) HEDMA_RLD6*** **(RW) HEDMA_RLD7*** **(RW) HEDMA_RLD8*** **(RW) HEDMA_RLD9*** **(RW) HEDMA_RLD10*** **(RW) HEDMA_RLD11*** **(RW) HEDMA_RLD12*** **(RW) HEDMA_RLD13*** **(RW) HEDMA_RLD14*** **(RW) HEDMA_RLD15*** **(RW) *PRAM*+***		
Fields	(RW) HEDMA_RLD_LINK	link address
	(RW) HEDMA_RLD_ELERLD	element count reload
* only supported on devices with EDMA + all of the macros apply for the options member of a PRAM table for both read and write		

5.5.7 HEDMA_PQSR

Priority queue status register

(R) HEDMA_PQSR*		
Fields	(R) HEDMA_PQSR_PQ0	priority queue 0 status
	(R) HEDMA_PQSR_PQ1	priority queue 1 status
	(R) HEDMA_PQSR_PQ2	priority queue 2 status
* only supported on devices with EDMA		

5.5.8 HEDMA_CIPR *Channel interrupt pending register*

(RW) HEDMA_CIPR*		
Fields	(RW) HEDMA_CIPR_CIP0	interrupt 0 pending
	(RW) HEDMA_CIPR_CIP1	interrupt 1 pending
	(RW) HEDMA_CIPR_CIP2	interrupt 2 pending
	(RW) HEDMA_CIPR_CIP3	interrupt 3 pending
	(RW) HEDMA_CIPR_CIP4	interrupt 4 pending
	(RW) HEDMA_CIPR_CIP5	interrupt 5 pending
	(RW) HEDMA_CIPR_CIP6	interrupt 6 pending
	(RW) HEDMA_CIPR_CIP7	interrupt 7 pending
	(RW) HEDMA_CIPR_CIP8	interrupt 8 pending
	(RW) HEDMA_CIPR_CIP9	interrupt 9 pending
	(RW) HEDMA_CIPR_CIP10	interrupt 10 pending
	(RW) HEDMA_CIPR_CIP11	interrupt 11 pending
	(RW) HEDMA_CIPR_CIP12	interrupt 12 pending
	(RW) HEDMA_CIPR_CIP13	interrupt 13 pending
	(RW) HEDMA_CIPR_CIP14	interrupt 14 pending
	(RW) HEDMA_CIPR_CIP15	interrupt 15 pending
* only supported on devices with EDMA		

5.5.9 HEDMA_CIER *Channel interrupt enable register*

(RW) HEDMA_CIER*		
Fields	(RW) HEDMA_CIER_CIE0	interrupt 0 enable
	(RW) HEDMA_CIER_CIE1	interrupt 1 enable
	(RW) HEDMA_CIER_CIE2	interrupt 2 enable
	(RW) HEDMA_CIER_CIE3	interrupt 3 enable
	(RW) HEDMA_CIER_CIE4	interrupt 4 enable
	(RW) HEDMA_CIER_CIE5	interrupt 5 enable
	(RW) HEDMA_CIER_CIE6	interrupt 6 enable
	(RW) HEDMA_CIER_CIE7	interrupt 7 enable
	(RW) HEDMA_CIER_CIE8	interrupt 8 enable
	(RW) HEDMA_CIER_CIE9	interrupt 9 enable
	(RW) HEDMA_CIER_CIE10	interrupt 10 enable
	(RW) HEDMA_CIER_CIE11	interrupt 11 enable
	(RW) HEDMA_CIER_CIE12	interrupt 12 enable
	(RW) HEDMA_CIER_CIE13	interrupt 13 enable
	(RW) HEDMA_CIER_CIE14	interrupt 14 enable
	(RW) HEDMA_CIER_CIE15	interrupt 15 enable
* only supported on devices with EDMA		

5.5.10 HEDMA_CCER *Channel chain enable register*

(RW) HEDMA_CCER*		
Fields	(RW) HEDMA_CCER_CCE8	channel chain enable 8
	(RW) HEDMA_CCER_CCE9	channel chain enable 9
	(RW) HEDMA_CCER_CCE10	channel chain enable 10
	(RW) HEDMA_CCER_CCE11	channel chain enable 11
* only supported on devices with EDMA		

5.5.11 HEDMA_ER *EDMA event flag register*

(R) HEDMA_ER*		
Fields	(R) HEDMA_ER_EVT0	event 0 flag
	(R) HEDMA_ER_EVT1	event 1 flag
	(R) HEDMA_ER_EVT2	event 2 flag
	(R) HEDMA_ER_EVT3	event 3 flag
	(R) HEDMA_ER_EVT4	event 4 flag
	(R) HEDMA_ER_EVT5	event 5 flag
	(R) HEDMA_ER_EVT6	event 6 flag
	(R) HEDMA_ER_EVT7	event 7 flag
	(R) HEDMA_ER_EVT8	event 8 flag
	(R) HEDMA_ER_EVT9	event 9 flag
	(R) HEDMA_ER_EVT10	event 10 flag
	(R) HEDMA_ER_EVT11	event 11 flag
	(R) HEDMA_ER_EVT12	event 12 flag
	(R) HEDMA_ER_EVT13	event 13 flag
	(R) HEDMA_ER_EVT14	event 14 flag
	(R) HEDMA_ER_EVT15	event 15 flag
* only supported on devices with EDMA		

5.5.12 HEDMA_EER

EDMA event enable register

(RW) HEDMA_EER*		
Fields	(RW) HEDMA_EER_EE0	event 0 enable
	(RW) HEDMA_EER_EE1	event 1 enable
	(RW) HEDMA_EER_EE2	event 2 enable
	(RW) HEDMA_EER_EE3	event 3 enable
	(RW) HEDMA_EER_EE4	event 4 enable
	(RW) HEDMA_EER_EE5	event 5 enable
	(RW) HEDMA_EER_EE6	event 6 enable
	(RW) HEDMA_EER_EE7	event 7 enable
	(RW) HEDMA_EER_EE8	event 8 enable
	(RW) HEDMA_EER_EE9	event 9 enable
	(RW) HEDMA_EER_EE10	event 10 enable
	(RW) HEDMA_EER_EE11	event 11 enable
	(RW) HEDMA_EER_EE12	event 12 enable
	(RW) HEDMA_EER_EE13	event 13 enable
	(RW) HEDMA_EER_EE14	event 14 enable
	(RW) HEDMA_EER_EE15	event 15 enable
* only supported on devices with EDMA		

5.5.13 HEDMA_ECR — *EDMA event clear register*

(RW) HEDMA_ECR*		
Fields	(RW) HEDMA_ECR_EC0	event 0 clear
	(RW) HEDMA_ECR_EC1	event 1 clear
	(RW) HEDMA_ECR_EC2	event 2 clear
	(RW) HEDMA_ECR_EC3	event 3 clear
	(RW) HEDMA_ECR_EC4	event 4 clear
	(RW) HEDMA_ECR_EC5	event 5 clear
	(RW) HEDMA_ECR_EC6	event 6 clear
	(RW) HEDMA_ECR_EC7	event 7 clear
	(RW) HEDMA_ECR_EC8	event 8 clear
	(RW) HEDMA_ECR_EC9	event 9 clear
	(RW) HEDMA_ECR_EC10	event 10 clear
	(RW) HEDMA_ECR_EC11	event 11 clear
	(RW) HEDMA_ECR_EC12	event 12 clear
	(RW) HEDMA_ECR_EC13	event 13 clear
	(RW) HEDMA_ECR_EC14	event 14 clear
	(RW) HEDMA_ECR_EC15	event 15 clear
* only supported on devices with EDMA		

5.5.14 HEDMA_ESR

EDMA event set register

(RW) HEDMA_ESR*		
Fields	(RW) HEDMA_ESR_ES0	event 0 set
	(RW) HEDMA_ESR_ES1	event 1 set
	(RW) HEDMA_ESR_ES2	event 2 set
	(RW) HEDMA_ESR_ES3	event 3 set
	(RW) HEDMA_ESR_ES4	event 4 set
	(RW) HEDMA_ESR_ES5	event 5 set
	(RW) HEDMA_ESR_ES6	event 6 set
	(RW) HEDMA_ESR_ES7	event 7 set
	(RW) HEDMA_ESR_ES8	event 8 set
	(RW) HEDMA_ESR_ES9	event 9 set
	(RW) HEDMA_ESR_ES10	event 10 set
	(RW) HEDMA_ESR_ES11	event 11 set
	(RW) HEDMA_ESR_ES12	event 12 set
	(RW) HEDMA_ESR_ES13	event 13 set
	(RW) HEDMA_ESR_ES14	event 14 set
	(RW) HEDMA_ESR_ES15	event 15 set
* only supported on devices with EDMA		

5.6 HEMIF

5.6.1 HEMIF_GBLCTL *global control register*

(RW) HEMIF_GBLCTL		
Fields	(R) HEMIF_GBLCTL_MAP	map mode
	(RW) HEMIF_GBLCTL_RBTR8	requester arbitration mode
	(RW) HEMIF_GBLCTL_SSCRT	SBSRAM clock rate select
	(RW) HEMIF_GBLCTL_CLK2EN	CLKOUT2 enable
	(RW) HEMIF_GBLCTL_CLK1EN	CLKOUT1 enable
	(RW) HEMIF_GBLCTL_SSCEN	SSCLK enable
	(RW) HEMIF_GBLCTL_SDCEN	SDCLK enable
	(RW) HEMIF_GBLCTL_NOHOLD	external hold disable
	(R) HEMIF_GBLCTL_HOLDA	HOLDA output control
	(R) HEMIF_GBLCTL_HOLD	HOLD input
	(R) HEMIF_GBLCTL_ARDY	ARDY input
	(R) HEMIF_GBLCTL_BUSREQ	BUSREQ output control

5.6.2 HEMIF_CECTL *CE space control register*

(RW) HEMIF_CECTL		
Fields	(RW) HEMIF_CECTL_RDHLD	read hold
	(RW) HEMIF_CECTL_WRHLDMSB	write hold MSB
	(RW) HEMIF_CECTL_MTYPE	memory type
	(RW) HEMIF_CECTL_RDSTRB	read strobe
	(RW) HEMIF_CECTL_TA	turn around time
	(RW) HEMIF_CECTL_RDSETUP	read setup
	(RW) HEMIF_CECTL_WRHLD	write hold
	(RW) HEMIF_CECTL_WRSTRB	write strobe
	(RW) HEMIF_CECTL_WRSETUP	write setup

5.6.3 HEMIF_SDCTL *SDRAM control register*

(RW) HEMIF_SDCTL		
Fields	(RW) HEMIF_SDCTL_TRC	Trc value
	(RW) HEMIF_SDCTL_TRP	Trp value
	(RW) HEMIF_SDCTL_TRCD	Trcd value
	(W) HEMIF_SDCTL_INIT	SDRAM initialization
	(RW) HEMIF_SDCTL_RFEN	refresh enable
	(RW) HEMIF_SDCTL_SDWID	SDRAM width select
	(RW) HEMIF_SDCTL_SDCSZ	SDRAM column size
	(RW) HEMIF_SDCTL_SDRSZ	SDRAM row size
	(RW) HEMIF_SDCTL_SDBSZ	SDRAM bank size

5.6.4 HEMIF_SDTIM *SDRAM timing register*

(RW) HEMIF_TIM		
Fields	(RW) HEMIF_SDTIM_PERIOD	refresh period
	(R) HEMIF_SDTIM_CNTR	current value of refresh counter
	(RW) HEMIF_SDTIM_XRFR	extra refreshes

5.6.5 HEMIF_SDEXT *SDRAM extension register*

(RW) HEMIF_SDEXT*		
Fields	(RW) HEMIF_SDEXT_TCL	CAS latency
	(RW) HEMIF_SDEXT_TRAS	Tras value
	(RW) HEMIF_SDEXT_TRRD	Trrd value
	(RW) HEMIF_SDEXT_TWR	Twr value
	(RW) HEMIF_SDEXT_THZP	Thzp value
	(RW) HEMIF_SDEXT_RD2RD	read to read cycles
	(RW) HEMIF_SDEXT_RD2DEAC	read to DEAC/DCAB cycles
	(RW) HEMIF_SDEXT_RD2WR	read to write cycles
	(RW) HEMIF_SDEXT_R2WDQM	BEx to write cycles
	(RW) HEMIF_SDEXT_WR2WR	write to write cycles
	(RW) HEMIF_SDEXT_WR2DEAC	write to DEAC/DCAB cycles
	(RW) HEMIF_SDEXT_WR2RD	write to read cycles
* only supported on C6X11 devices		

5.7 HHPI

5.7.1 HHPI_HPIC — *HPI control register*

(RW) HHPI_HPIC*		
Fields	(R) HHPI_HPIC_HWOB	halfword ordering bit
	(RW) HHPI_HPIC_DSPINT	host to CPU interrupt
	(RW) HHPI_HPIC_HINT	DSP to host interrupt
	(R) HHPI_HPIC_HRDY	ready signal to host
	(R) HHPI_HPIC_FETCH	host fetch request
* only supported on devices with an HPI		

5.8 HIRQ

5.8.1 HIRQ_MUXL

interrupt multiplexer low register

(RW) HIRQ_MUXL		
Fields	(RW) HIRQ_MUXL_INTSEL4	interrupt select 4
	(RW) HIRQ_MUXL_INTSEL5	interrupt select 5
	(RW) HIRQ_MUXL_INTSEL6	interrupt select 6
	(RW) HIRQ_MUXL_INTSEL7	interrupt select 7
	(RW) HIRQ_MUXL_INTSEL8	interrupt select 8
	(RW) HIRQ_MUXL_INTSEL9	interrupt select 9

5.8.2 HIRQ_MUXH

interrupt multiplexer high register

(RW) HIRQ_MUXH		
Fields	(RW) HIRQ_MUXH_INTSEL10	interrupt select 10
	(RW) HIRQ_MUXH_INTSEL11	interrupt select 11
	(RW) HIRQ_MUXH_INTSEL12	interrupt select 12
	(RW) HIRQ_MUXH_INTSEL13	interrupt select 13
	(RW) HIRQ_MUXH_INTSEL14	interrupt select 14
	(RW) HIRQ_MUXH_INTSEL15	interrupt select 15

5.8.3 HIRQ_EXTPOL

external interrupt polarity register

(RW) HIRQ_EXTPOL		
Fields	(RW) HIRQ_EXTPOL_XIP4	external interrupt 4 polarity
	(RW) HIRQ_EXTPOL_XIP5	external interrupt 5 polarity
	(RW) HIRQ_EXTPOL_XIP6	external interrupt 6 polarity
	(RW) HIRQ_EXTPOL_XIP7	external interrupt 7 polarity

5.9 HMCBSP

5.9.1 HMCBSP_DRR *data receive register*

(R) HMCBSP_DRR0 **(R) HMCBSP_DRR1** **(R) HMCBSP_DRR2***		
Fields	(R) HMCBSP_DRR_DRR	data
* only on devices with three or more serial ports		

5.9.2 HMCBSP_DXR *data transmit register*

(W) HMCBSP_DXR0 **(W) HMCBSP_DXR1** **(W) HMCBSP_DXR2***		
Fields	(W) HMCBSP_DXR_DXR	data
* only on devices with three or more serial ports		

5.9.3 HMCBSP_SPCR *serial port control register*

(RW) HMCBSP_SPCR0 **(RW) HMCBSP_SPCR1** **(RW) HMCBSP_SPCR2***		
Fields	(RW) HMCBSP_SPCR_RRST	receiver reset
	(R) HMCBSP_SPCR_RRDY	receiver ready
	(R) HMCBSP_SPCR_RFULL	receive shift register full
	(RW) HMCBSP_SPCR_RSYNCERR	receive synchronization error
	(RW) HMCBSP_SPCR_RINTM	receive interrupt mode
	(RW) HMCBSP_SPCR_DXENA	DX enabler
	(RW) HMCBSP_SPCR_CLKSTP	clock stop mode
	(RW) HMCBSP_SPCR_RJUST	receive data sign-ext and justification mode
	(RW) HMCBSP_SPCR_DLB	digital loopback mode
	(RW) HMCBSP_SPCR_XRST	transmitter reset
	(R) HMCBSP_SPCR_XRDY	transmitter ready
	(R) HMCBSP_SPCR_XEMPTY	transmitter shift register empty
	(RW) HMCBSP_SPCR_XSYNCERR	transmit synchronization error
	(RW) HMCBSP_SPCR_XINTM	transmit interrupt mode

Table Continued

	(RW) HMCBSP_SPCR_GRST	sample rate generator reset
	(RW) HMCBSP_SPCR_FRST	frame sync generator reset
* only on devices with three or more serial ports		

5.9.4 HMCBSP_RCR *receive control register*

(RW) HMCBSP_RCR0 **(RW) HMCBSP_RCR1** **(RW) HMCBSP_RCR2***		
Fields	(RW) HMCBSP_RCR_RWDREVRS	32-bit reversal feature
	(RW) HMCBSP_RCR_RWDLEN1	element length in phase 1
	(RW) HMCBSP_RCR_RFRLEN1	frame length in phase 1
	(RW) HMCBSP_RCR_RPHASE2	phase 2
	(RW) HMCBSP_RCR_RDATDLY	data delay
	(RW) HMCBSP_RCR_RFIG	frame ignore
	(RW) HMCBSP_RCR_RCOMPAND	companding mode
	(RW) HMCBSP_RCR_RWDLEN2	element length in phase 2
	(RW) HMCBSP_RCR_RFRLEN2	frame length in phase 2
	(RW) HMCBSP_RCR_RPHASE	phases
* only on devices with three or more serial ports		

5.9.5 HMCBSP_XCR *transmit control register*

(RW) HMCBSP_XCR0 (RW) HMCBSP_XCR1 (RW) HMCBSP_XCR2*		
Fields	(RW) HMCBSP_XCR_XWDREVRS	32-bit reversal feature
	(RW) HMCBSP_XCR_XWDLEN1	element length in phase 1
	(RW) HMCBSP_XCR_XFRLEN1	frame length in phase 1
	(RW) HMCBSP_XCR_XPHASE2	phase 2
	(RW) HMCBSP_XCR_XDATDLY	data delay
	(RW) HMCBSP_XCR_XFIG	frame ignore
	(RW) HMCBSP_XCR_XCOMPAND	companding mode
	(RW) HMCBSP_XCR_XWDLEN2	element length in phase 2
	(RW) HMCBSP_XCR_XFRLEN2	frame length in phase 2
	(RW) HMCBSP_XCR_XPHASE	phases
* only on devices with three or more serial ports		

5.9.6 HMCBSP_SRGR *sample rate generator register*

(RW) HMCBSP_SRGR0 (RW) HMCBSP_SRGR1 (RW) HMCBSP_SRGR2*		
Fields	(RW) HMCBSP_SRGR_CLKGDV	clock divider
	(RW) HMCBSP_SRGR_FWID	frame width
	(RW) HMCBSP_SRGR_FPER	frame period
	(RW) HMCBSP_SRGR_FSGM	transmit frame sync mode
	(RW) HMCBSP_SRGR_CLKSM	clock mode
	(RW) HMCBSP_SRGR_CLKSP	CLKS polarity clock edge select
	(RW) HMCBSP_SRGR_GSYNC	clock sync
* only on devices with three or more serial ports		

5.9.7 HMCBSP_MCR *multichannel control register*

(RW) HMCBSP_MCR0 (RW) HMCBSP_MCR1 (RW) HMCBSP_MCR2*		
Fields	(RW) HMCBSP_MCR_RMCM	receive multichannel selection enable
	(R) HMCBSP_MCR_RCBLK	receive current subframe
	(RW) HMCBSP_MCR_RPABLK	receive partition A subframe
	(RW) HMCBSP_MCR_RPBBLK	receive partition B subframe
	(RW) HMCBSP_MCR_XMCM	transmit multichannel selection enable
	(R) HMCBSP_MCR_XCBLK	transmit current subframe
	(RW) HMCBSP_MCR_XPABLK	transmit partition A subframe
	(RW) HMCBSP_MCR_XPBBLK	transmit partition B subframe
* only on devices with three or more serial ports		

5.9.8 HMCBSP_RCER *receive channel enable register*

(RW) HMCBSP_RCER0 (RW) HMCBSP_RCER1 (RW) HMCBSP_RCER2*		
Fields	(RW) HMCBSP_RCER_RCEA0	enable element 0 in partition A
	(RW) HMCBSP_RCER_RCEA1	enable element 1 in partition A
	(RW) HMCBSP_RCER_RCEA2	enable element 2 in partition A
	(RW) HMCBSP_RCER_RCEA3	enable element 3 in partition A
	(RW) HMCBSP_RCER_RCEA4	enable element 4 in partition A
	(RW) HMCBSP_RCER_RCEA5	enable element 5 in partition A
	(RW) HMCBSP_RCER_RCEA6	enable element 6 in partition A
	(RW) HMCBSP_RCER_RCEA7	enable element 7 in partition A
	(RW) HMCBSP_RCER_RCEA8	enable element 8 in partition A
	(RW) HMCBSP_RCER_RCEA9	enable element 9 in partition A
	(RW) HMCBSP_RCER_RCEA10	enable element 10 in partition A
	(RW) HMCBSP_RCER_RCEA11	enable element 11 in partition A
	(RW) HMCBSP_RCER_RCEA12	enable element 12 in partition A
	(RW) HMCBSP_RCER_RCEA13	enable element 13 in partition A
	(RW) HMCBSP_RCER_RCEA14	enable element 14 in partition A
	(RW) HMCBSP_RCER_RCEA15	enable element 15 in partition A
	(RW) HMCBSP_RCER_RCEB0	enable element 0 in partition B
	(RW) HMCBSP_RCER_RCEB1	enable element 1 in partition B

Table Continued

	(RW) HMCBSP_RCER_RCEB2	enable element 2 in partition B
	(RW) HMCBSP_RCER_RCEB3	enable element 3 in partition B
	(RW) HMCBSP_RCER_RCEB4	enable element 4 in partition B
	(RW) HMCBSP_RCER_RCEB5	enable element 5 in partition B
	(RW) HMCBSP_RCER_RCEB6	enable element 6 in partition B
	(RW) HMCBSP_RCER_RCEB7	enable element 7 in partition B
	(RW) HMCBSP_RCER_RCEB8	enable element 8 in partition B
	(RW) HMCBSP_RCER_RCEB9	enable element 9 in partition B
	(RW) HMCBSP_RCER_RCEB10	enable element 10 in partition B
	(RW) HMCBSP_RCER_RCEB11	enable element 11 in partition B
	(RW) HMCBSP_RCER_RCEB12	enable element 12 in partition B
	(RW) HMCBSP_RCER_RCEB13	enable element 13 in partition B
	(RW) HMCBSP_RCER_RCEB14	enable element 14 in partition B
	(RW) HMCBSP_RCER_RCEB15	enable element 15 in partition B
* only on devices with three or more serial ports		

5.9.9 HMCBSP_XCER *transmit channel enable register*

(RW) HMCBSP_XCER0 **(RW) HMCBSP_XCER1** **(RW) HMCBSP_XCER2***		
Fields	(RW) HMCBSP_XCER_XCEA0	enable element 0 in partition A
	(RW) HMCBSP_XCER_XCEA1	enable element 1 in partition A
	(RW) HMCBSP_XCER_XCEA2	enable element 2 in partition A
	(RW) HMCBSP_XCER_XCEA3	enable element 3 in partition A
	(RW) HMCBSP_XCER_XCEA4	enable element 4 in partition A
	(RW) HMCBSP_XCER_XCEA5	enable element 5 in partition A
	(RW) HMCBSP_XCER_XCEA6	enable element 6 in partition A
	(RW) HMCBSP_XCER_XCEA7	enable element 7 in partition A
	(RW) HMCBSP_XCER_XCEA8	enable element 8 in partition A
	(RW) HMCBSP_XCER_XCEA9	enable element 9 in partition A
	(RW) HMCBSP_XCER_XCEA10	enable element 10 in partition A
	(RW) HMCBSP_XCER_XCEA11	enable element 11 in partition A
	(RW) HMCBSP_XCER_XCEA12	enable element 12 in partition A
	(RW) HMCBSP_XCER_XCEA13	enable element 13 in partition A
	(RW) HMCBSP_XCER_XCEA14	enable element 14 in partition A

Table Continued

	(RW) HMCBSP_XCER_XCEA15	enable element 15 in partition A
	(RW) HMCBSP_XCER_XCEB0	enable element 0 in partition B
	(RW) HMCBSP_XCER_XCEB1	enable element 1 in partition B
	(RW) HMCBSP_XCER_XCEB2	enable element 2 in partition B
	(RW) HMCBSP_XCER_XCEB3	enable element 3 in partition B
	(RW) HMCBSP_XCER_XCEB4	enable element 4 in partition B
	(RW) HMCBSP_XCER_XCEB5	enable element 5 in partition B
	(RW) HMCBSP_XCER_XCEB6	enable element 6 in partition B
	(RW) HMCBSP_XCER_XCEB7	enable element 7 in partition B
	(RW) HMCBSP_XCER_XCEB8	enable element 8 in partition B
	(RW) HMCBSP_XCER_XCEB9	enable element 9 in partition B
	(RW) HMCBSP_XCER_XCEB10	enable element 10 in partition B
	(RW) HMCBSP_XCER_XCEB11	enable element 11 in partition B
	(RW) HMCBSP_XCER_XCEB12	enable element 12 in partition B
	(RW) HMCBSP_XCER_XCEB13	enable element 13 in partition B
	(RW) HMCBSP_XCER_XCEB14	enable element 14 in partition B
	(RW) HMCBSP_XCER_XCEB15	enable element 15 in partition B
* only on devices with three or more serial ports		

5.9.10 HMCBSP_PCR

pin control register

(RW) HMCBSP_PCR0 **(RW) HMCBSP_PCR1** **(RW) HMCBSP_PCR2***		
Fields	(RW) HMCBSP_PCR_CLKRP	receive clock polarity
	(RW) HMCBSP_PCR_CLKXP	transmit clock polarity
	(RW) HMCBSP_PCR_FSRP	receive frame sync polarity
	(RW) HMCBSP_PCR_FSXP	transmit frame sync polarity
	(R) HMCBSP_PCR_DRSTAT	DR pin status
	(RW) HMCBSP_PCR_DXSTAT	DX pin status
	(RW) HMCBSP_PCR_CLKSSTAT	CLKS pin status
	(RW) HMCBSP_PCR_CLKRM	receiver clock mode
	(RW) HMCBSP_PCR_CLKXM	transmitter clock mode
	(RW) HMCBSP_PCR_FSRM	receive frame sync mode
	(RW) HMCBSP_PCR_FSXM	transmit frame sync mode

Table Continued

	(RW) HMCBSP_PCR_RIOEN	receiver general purpose IO mode
	(RW) HMCBSP_PCR_XIOEN	transmitter general purpose IO mode
* only on devices with three or more serial ports		

5.10 HPWR

5.10.1 HPWR_PDCTL *peripheral power-down control register*

(RW) HPWR_PDCTL*		
Fields	(RW) HPWR_PDCTL_DMA	DMA clock enable
	(RW) HPWR_PDCTL_EMIF	EMIF clock enable
	(RW) HPWR_PDCTL_MCBSP0	MCBSP0 clock enable
	(RW) HPWR_PDCTL_MCBSP1	MCBSP1 clock enable
	(RW) HPWR_PDCTL_MCBSP2	MCBSP2 clock enable
* only on 6202 and 6203 devices		

5.11 HTIMER

5.11.1 HTIMER_CTL *timer control register*

(RW) HTIMER_CTL0 **(RW) HTIMER_CTL1**		
Fields	(RW) HTIMER_CTL_FUNC	function of TOUT pin
	(RW) HTIMER_CTL_INVOUT	TOUT inverter control
	(RW) HTIMER_CTL_DATOUT	data output
	(RW) HTIMER_CTL_DATIN	data in
	(RW) HTIMER_CTL_PWID	pulse width
	(RW) HTIMER_CTL_GO	GO bit
	(RW) HTIMER_CTL_HLD	hold
	(RW) HTIMER_CTL_CP	clock/pulse mode
	(RW) HTIMER_CTL_CLKSRC	timer input clock source
	(RW) HTIMER_CTL_INVINP	TINP inverter control
	(R) HTIMER_CTL_TSTAT	timer status

5.11.2 HTIMER_PRD *timer period register*

(RW) HTIMER_PRD0 **(RW) HTIMER_PRD1**		
Fields	(RW) HTIMER_PRD_PRD	period

5.11.3 HTIMER_CNT *timer count register*

(RW) HTIMER_CNT0 **(RW) HTIMER_CNT1**		
Fields	(RW) HTIMER_CNT_CNT	count

Appendix A

Glossary

A

address: The location of program code or data stored; an individually accessible memory location.

A-law companding: See *compress and expand (compand)*.

API: See *application programming interface.*

API module: A set of API functions designed for a specific purpose.

application programming interface (API): Used for proprietary application programs to interact with communications software or to conform to protocols from another vendor's product.

assembler: A software program that creates a machine language program from a source file that contains assembly language instructions, directives, and macros. The assembler substitutes absolute operation codes for symbolic operation codes and absolute or relocatable addresses for symbolic addresses.

assert: To make a digital logic device pin active. If the pin is active low, then a low voltage on the pin asserts it. If the pin is active high, then a high voltage asserts it.

B

bit: A binary digit, either a 0 or 1.

big endian: An addressing protocol in which bytes are numbered from left to right within a word. More significant bytes in a word have lower numbered addresses. Endian ordering is specific to hardware and is determined at reset. See also *little endian*.

block: The three least significant bits of the program address. These correspond to the address within a fetch packet of the first instruction being addressed.

board support library (BSL): The BSL is a set of application programming interfaces (APIs) consisting of target side DSP code used to configure and control board level peripherals.

boot: The process of loading a program into program memory.

boot mode: The method of loading a program into program memory. The ’C6x DSP supports booting from external ROM or the host port interface (HPI).

BSL: *See board support library.*

byte: A sequence of eight adjacent bits operated upon as a unit.

C

cache: A fast storage buffer in the central processing unit of a computer.

cache module: CACHE is an API module containing a set of functions for managing data and program cache.

cache controller: System component that coordinates program accesses between CPU program fetch mechanism, cache, and external memory.

CCS: Code Composer Studio.

central processing unit (CPU): The portion of the processor involved in arithmetic, shifting, and Boolean logic operations, as well as the generation of data- and program-memory addresses. The CPU includes the central arithmetic logic unit (CALU), the multiplier, and the auxiliary register arithmetic unit (ARAU).

CHIP: *See CHIP module.*

CHIP module: The CHIP module is an API module where chip-specific and device-related code resides. CHIP has some API functions for obtaining device endianess, memory map mode if applicable, CPU and REV IDs, and clock speed.

chip support library (CSL): The CSL is a set of application programming interfaces (APIs) consisting of target side DSP code used to configure and control all on-chip peripherals.

clock cycle: A periodic or sequence of events based on the input from the external clock.

clock modes: Options used by the clock generator to change the internal CPU clock frequency to a fraction or multiple of the frequency of the input clock signal.

code: A set of instructions written to perform a task; a computer program or part of a program.

coder-decoder or compression/decompression (codec): A device that codes in one direction of transmission and decodes in another direction of transmission.

compiler: A computer program that translates programs in a high-level language into their assembly-language equivalents.

compress and expand (compand): A quantization scheme for audio signals in which the input signal is compressed and then, after processing, is reconstructed at the output by expansion. There are two distinct companding schemes: A-law (used in Europe) and μ-law (used in the United States).

control register: A register that contains bit fields that define the way a device operates.

control register file: A set of control registers.

CSL: See *chip support library*.

CSL module: The CSL module is the top-level CSL API module.It interfaces to all other modules and its main purpose is to initialize the CSL library.

D

DAT: *Data; see DAT module.*

DAT module: The DAT is an API module that is used to move data around by means of DMA/EDMA hardware. This module serves as a level of abstraction that works the same for devices that have the DMA or EDMA peripheral.

device ID: Configuration register that identifies each peripheral component interconnect (PCI).

digital signal processor (DSP): A semiconductor that turns analog signals—such as sound or light—into digital signals, which are discrete or discontinuous electrical impulses, so that they can be manipulated.

direct memory access (DMA): A mechanism whereby a device other than the host processor contends for and receives mastery of the memory bus so that data transfers can take place independent of the host.

DMA : See *direct memory access*.

DMA module: DMA is an API module that currently has two architectures used on ’C6x devices: DMA and EDMA (enhanced DMA). Devices such as the ’6201 have the DMA peripheral, whereas the ’6211 has the EDMA peripheral.

DMA source: The module where the DMA data originates. DMA data is read from the DMA source.

DMA transfer: The process of transferring data from one part of memory to another. Each DMA transfer consists of a read bus cycle (source to DMA holding register) and a write bus cycle (DMA holding register to destination).

E

EDMA: *Enhanced direct memory access; see EDMA module.*

EDMA module: EDMA is an API module that currently has two architectures used on ’C6x devices: DMA and EDMA (enhanced DMA). Devices such as the ’6201 have the DMA peripheral, whereas the ’6211 has the EDMA peripheral.

EMIF: *See external memory interface; see also EMIF module.*

EMIF module: EMIF is an API module that is used for configuring the EMIF registers.

evaluation module (EVM): Board and software tools that allow the user to evaluate a specific device.

external interrupt: A hardware interrupt triggered by a specific value on a pin.

external memory interface (EMIF): Microprocessor hardware that is used to read to and write from off-chip memory.

F

fetch packet: A contiguous 8-word series of instructions fetched by the CPU and aligned on an 8-word boundary.

flag: A binary status indicator whose state indicates whether a particular condition has occurred or is in effect.

frame: An 8-word space in the cache RAMs. Each fetch packet in the cache resides in only one frame. A cache update loads a frame with the requested fetch packet. The cache contains 512 frames.

G

global interrupt enable bit (GIE): A bit in the control status register (CSR) that is used to enable or disable maskable interrupts.

H

HAL: *Hardware abstraction layer* of the CSL. The HAL underlies the service layer and provides it a set of macros and constants for manipulating the peripheral registers at the lowest level. It is a low-level symbolic interface into the hardware providing symbols that describe peripheral registers/ bitfields and macros for manipulating them.

host: A device to which other devices (peripherals) are connected and that generally controls those devices.

host port interface (HPI): A parallel interface that the CPU uses to communicate with a host processor.

HPI: See *host port interface*; see also *HPI module*.

HPI module: HPI is an API module used for configuring the HPI registers. Functions are provided for reading HPI status bits and setting interrupt events.

I

index: A relative offset in the program address that specifies which of the 512 frames in the cache into which the current access is mapped.

indirect addressing: An addressing mode in which an address points to another pointer rather than to the actual data; this mode is prohibited in RISC architecture.

instruction fetch packet: A group of up to eight instructions held in memory for execution by the CPU.

internal interrupt: A hardware interrupt caused by an on-chip peripheral.

interrupt: A signal sent by hardware or software to a processor requesting attention. An interrupt tells the processor to suspend its current operation, save the current task status, and perform a particular set of instructions. Interrupts communicate with the operating system and prioritize tasks to be performed.

interrupt service fetch packet (ISFP): A fetch packet used to service interrupts. If eight instructions are insufficient, the user must branch out of this block for additional interrupt service. If the delay slots of the branch do not reside within the ISFP, execution continues from execute packets in the next fetch packet (the next ISFP).

interrupt service routine (ISR): A module of code that is executed in response to a hardware or software interrupt.

interrupt service table (IST) A table containing a corresponding entry for each of the 16 physical interrupts. Each entry is a single-fetch packet and has a label associated with it.

Internal peripherals: Devices connected to and controlled by a host device. The ’C6x internal peripherals include the direct memory access (DMA) controller, multichannel buffered serial ports (McBSPs), host port interface (HPI), external memory-interface (EMIF), and runtime support timers.

IRQ: *Interrupt request; see IRQ module.*

IRQ module: IRQ is an API module that manages CPU interrupts.

IST: *See interrupt service table.*

L

least significant bit (LSB): The lowest-order bit in a word.

linker: A software tool that combines object files to form an object module, which can be loaded into memory and executed.

little endian: An addressing protocol in which bytes are numbered from right to left within a word. More significant bytes in a word have higher-numbered addresses. Endian ordering is specific to hardware and is determined at reset. See also *big endian*.

M

μ-law companding: See *compress and expand (compand)*.

maskable interrupt: A hardware interrupt that can be enabled or disabled through software.

MCBSP: *See multichannel buffered serial port; see also MCBSP module*.

MCBSP module: MCBSP is an API module that contains a set of functions for configuring the McBSP registers.

memory map: A graphical representation of a computer system's memory, showing the locations of program space, data space, reserved space, and other memory-resident elements.

memory-mapped register: An on-chip register mapped to an address in memory. Some memory-mapped registers are mapped to data memory, and some are mapped to input/output memory.

most significant bit (MSB): The highest order bit in a word.

multichannel buffered serial port (McBSP): An on-chip full-duplex circuit that provides direct serial communication through several channels to external serial devices.

multiplexer: A device for selecting one of several available signals.

N

nonmaskable interrupt (NMI): An interrupt that can be neither masked nor disabled.

O

object file: A file that has been assembled or linked and contains machine language object code.

off chip: A state of being external to a device.

on chip: A state of being internal to a device.

P

peripheral: A device connected to and usually controlled by a host device.

program cache: A fast memory cache for storing program instructions allowing for quick execution.

program memory: Memory accessed through the ‘C6x’s program fetch interface.

PWR: *Power; see PWR module*.

PWR module: PWR is an API module that is used to configure the power-down control registers, if applicable, and to invoke various power-down modes.

R

random-access memory (RAM): A type of memory device in which the individual locations can be accessed in any order.

register: A small area of high speed memory located within a processor or electronic device that is used for temporarily storing data or instructions. Each register is given a name, contains a few bytes of information, and is referenced by programs.

reduced-instruction-set computer (RISC): A computer whose instruction set and related decode mechanism are much simpler than those of microprogrammed complex instruction set computers. The result is a higher instruction throughput and a faster real-time interrupt service response from a smaller, cost-effective chip.

reset: A means of bringing the CPU to a known state by setting the registers and control bits to predetermined values and signaling execution to start at a specified address.

RTOS *Real-time operating system.*

S

service layer: The top layer of the 2-layer chip support library architecture providing high-level APIs into the CSL and BSL. The service layer is where the actual APIs are defined and is the layer the user interfaces to.

STDINC: *Standard include; see STDINC module*

STDINC module: STDINC is an API module that defines some identifiers which are globally useful to everyone and are used throughout the CSL source code.

synchronous-burst static random-access memory (SBSRAM): RAM whose contents does not have to be refreshed periodically. Transfer of data is at a fixed rate relative to the clock speed of the device, but the speed is increased.

synchronous dynamic random-access memory (SDRAM): RAM whose contents is refreshed periodically so the data is not lost. Transfer of data is at a fixed rate relative to the clock speed of the device.

syntax: The grammatical and structural rules of a language. All higher-level programming languages possess a formal syntax.

system software: The blanketing term used to denote collectively the chip support libraries and board support libraries.

T

tag: The 18 most significant bits of the program address. This value corresponds to the physical address of the fetch packet that is in that frame.

timer: A programmable peripheral used to generate pulses or to time events.

TIMER module: TIMER is an API module used for configuring the timer registers.

W

word: A multiple of eight bits that is operated upon as a unit. For the ‘C6x, a word is 32 bits in length.

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

W

Y