TMS320C6000 Chip Support Library API Reference Guide

Literature Number SPRU401G
June 2003

Printed on Recycled Paper

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

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

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third–party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Following are URLs where you can obtain information on other Texas Instruments products & application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments
Post Office Box 655303 Dallas, Texas 75265

Preface

Read This First

About This Manual

The TMS320C6000™ Chip Support Library (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 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, and standardization. A version of the CSL is available for all TMS320C6000™ devices.

This document is organized as follows:

- Introduction – a high level overview of the CSL
- 22 CSL API module chapters
- HAL macro chapter
- Using CSL APIs Without DSP/BIOS
- Registers
- How to Use the CSL
- Cache register comparison
- Glossary

How to Use This Manual

The information in this document describes the contents of the TMS320C6000™ chip support library (CSL) as follows:

- Chapter 1 provides an overview of the CSL, includes a table showing CSL API module support for various C6000 devices, and lists the API modules.

- Each additional chapter discusses an individual CSL API module and provides:
  - A description of the API module
  - A table showing the APIs within the module and a page reference for more specific information
  - A table showing the macros within the module and a page reference for more specific information
  - A module API Reference section in alphabetical order listing the CSL API functions, enumerations, type definitions, structures, constants, and global variables. Examples are given to show how these elements are used.
- Chapter 24 describes the hardware abstraction layer (HAL) and provides a HAL macro reference section.
- Appendix A provides an example of using CSL independently of DSP/BIOS.
- Appendix B provides a list of the registers associated with current TMS320C6000 DSP devices.
- Appendix C provides basic examples of how to use CSL functions and shows how to define build options in the Code Composer Studio environment for a variety of peripherals.
- Appendix D provides a comparison of the old and new CACHE register names, as they have recently been changed.
- Appendix E provides a glossary.

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 TMS320C6000™ CPU architecture, instruction set, pipeline, and interrupts for these digital signal processors.

TMS320C6x C Source Debugger User’s Guide (literature number SPRU188) tells you how to invoke the TMS320C6x™ simulator and emulator versions of the C source debugger interface. This book discusses various aspects of the debugger, including command entry, code execution, data management, breakpoints, profiling, and analysis.

TMS320C6000 Peripherals Reference Guide (literature number SPRU190) describes the peripherals available on the C6000 platform of devices. This includes internal data and program memories, external memory interface (EMIF), host port interface (HPI), multichannel buffered serial ports (McBSPs), direct memory access (DMA), enhanced direct access controller (EDMA), expansion bus (XBUS), peripheral component interconnect (PCI), clocking and phase-locked loop (PLL), timers, and 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 TMS320C6000™ generation of devices.

TMS320C6000 Optimizing Compiler User’s Guide (literature number SPRU187) describes the TMS320C6000™ C compiler and the assembly optimizer. This C compiler accepts ANSI standard C source code and produces assembly language source code for the TMS320C6000 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.

TMS320C64x Technical Overview (SPRU395) The TMS320C64x technical overview gives an introduction to the TMS320C64x™ digital signal processor, and discusses the application areas that are enhanced by the TMS320C64x VelociTI™.

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.

Trademarks

Contents

Figures

Tables

Examples

Chapter 1

CSL Overview

This chapter provides an overview of the chip support library (CSL), shows which TMS320C6000™ devices support the various application programming interfaces (APIs), and lists each of the API modules.

Topic **Page**

1.1 CSL Introduction

The chip support library (CSL) provides a C-language interface for configuring and controlling on-chip peripherals. It consists of discrete modules that are built and archived into a library file. Each module relates to a single peripheral with the exception of several modules that provide general programming support, such as the interrupt request (IRQ) module which contains APIs for interrupt management, and the CHIP module which allows the global setting of the chip.

1.1.1 Benefits of the CSL

The benefits of the CSL include peripheral ease of use, shortened development time, portability, hardware abstraction, and a level of standardization and compatibility among devices. Specifically, the CSL offers:

- Standard Protocol-to-Program Peripherals

  The CSL provides a standard protocol for programming the on-chip peripherals. This includes data types and macros to define a peripheral's configuration, and functions to implement the various operations of each peripheral.

- Basic Resource Management

  Basic resource management is provided through the use of Open and Close functions for many of the peripherals. This is especially helpful for peripherals that support multiple channels.

- Symbolic Peripheral Descriptions

  As a side benefit to the creation of the CSL, a complete symbolic description of all peripheral registers and register fields has been created. You will find it advantageous to use the higher-level protocols described in the first two benefits, because these are less device-specific, thus making it easier to migrate your code to newer versions of TI DSPs.

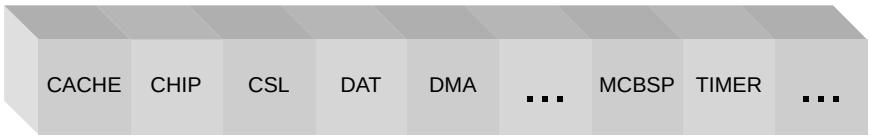
In general, the CSL makes it easier for you to get your system up and running in the shortest length of time.

1.1.2 CSL Architecture

The CSL granularity is designed such that each peripheral is covered by a single API module. Hence, there is a direct memory access (DMA) API module for the DMA peripheral, a multichannel buffered serial port (McBSP) API module for the McBSP peripheral, and so on.

Figure 1–1 illustrates some of the individual API modules (see section 1.8 for a complete list). This architecture allows for future expansion of the CSL because new API modules can be added as new peripheral devices emerge.

Figure 1–1. API Module Architecture



It is important to note that **not all** devices support **all** API modules. This depends on if the device actually has the peripheral to which an API relates. For example, the enhanced direct memory access (EDMA) API module is not supported on a C6201 because this device does not have an EDMA peripheral. Other modules such as the interrupt request (IRQ) module, however, are supported on all devices.

Table 1–1 lists general and peripheral modules with their associated include file and the module support symbol. These components must be included in your application.

Table 1–1. CSL Modules and Include Files

Peripheral Module (PER)	Description	Include File	Module Support Symbol†
CACHE	Cache module	csl_cache.h	CACHE_SUPPORT
CHIP	Chip-specific module	csl_chip.h	CHIP_SUPPORT
CSL	Top-level module	csl.h	NA
DAT	Device independent data copy/fill module	csl_dat.h	DAT_SUPPORT
DMA	Direct memory access module	csl_dma.h	DMA_SUPPORT
EDMA	Enhanced direct memory access module	csl_edma.h	EDMA_SUPPORT
EMIF	External memory interface module	csl_emif.h	EMIF_SUPPORT
EMIFA	External memory interface A module	csl_emifa.h	EMIFA_SUPPORT
EMIFB	External memory interface B module	csl_emifb.h	EMIFB_SUPPORT
GPIO	General-Purpose input/output module	csl_gpio.h	GPIO_SUPPORT
HPI	Host port interface module	csl_hpi.h	HPI_SUPPORT
I2C	Inter–Integrated circuit module	csl_i2c.h	I2C_SUPPORT
IRQ	Interrupt controller module	csl_irq.h	IRQ_SUPPORT
McASP	Multichannel audio serial port module	csl_mcasp.h	MCASP_SUPPORT
McBSP	Multichannel buffered serial port module	csl_mcbsp.h	MCBSP_SUPPORT
PCI	Peripheral component interconnect interface module	csl_pci.h	PCI_SUPPORT
PWR	Power-down module	csl_pwr.h	PWR_SUPPORT
TCP	Turbo decoder coprocessor module	csl_tcp.h	TCP_SUPPORT
TIMER	Timer module	csl_timer.h	TIMER_SUPPORT
UTOP	Utopia interface module	csl_utop.h	UTOP_SUPPORT
VCP	Viterbi decoder coprocessor module	csl_vcp.h	VCP_SUPPORT
XBUS	Expansion bus module	csl_xbus.h	XBUS_SUPPORT

† See definition in the related CSL module.

1.1.3 Interdependencies

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

1.2 CSL Naming Conventions

Table 1–2 shows the conventions used when naming CSL functions, macros, and data types.

Table 1–2. CSL Naming Conventions

Object Type	Naming Convention
Function	PER_funcName()†
Variable	PER_varName†
Macro	PER_MACRO_NAME†
Typedef	PER_Typename†
Function Argument	funcArg
Structure Member	memberName

† PER is the placeholder for the module name.

- ❑ All functions, variables, macros, and data types start with PER_ (where PER is the module/peripheral name) in caps (uppercase letters)
- ❑ Function names follow the peripheral name and use all small (lower-case) letters. Capital letters are used only if the function name consists of two separate words, such as PER_getConfig()
- ❑ Macro names follow the peripheral name and use all caps, for example, DMA_PRICTL_RMK
- ❑ Data types start with uppercase letters followed by lowercase letters, such as DMA_Handle

Note: CSL Macro and Function Names

The CSL macro and constant names are defined for each register and each field in CSL include files. Therefore, you will need to be careful not to redefine macros using similar names.

Because many CSL functions are predefined in CSL libraries, you will need to name your own functions carefully.

1.3 CSL Data Types

The CSL provides its own set of data types. Table 1–3 lists the CSL data types as defined in the stdinc.h file.

Table 1–3. CSL Data Types

Data Type	Description
Uint8	unsigned char
Uint16	unsigned short
Uint32	unsigned int
Uint40	unsigned long
Int8	char
Int16	short
Int32	int
Int40	long

These data types are available to all CSL modules. Additional data types are defined within each module and are described by each module's chapter.

1.4 CSL Functions

Table 1–4 provides a generic description of the most common CSL functions where *PER* indicates a peripheral as listed in Table 1–1. Because not all of the functions are available for all the modules, specific descriptions and functions are listed in each module chapter.

The following conventions are used and are shown in Table 1–4.

- Italics indicate variable names.
- Brackets [...] indicate optional parameters.
  - *[handle]* is required only for the handle-based peripherals: DAT, DMA, EDMA, GPIO, McBSP, and TIMER. See section 1.7.1.
  - *[priority]* is required only for the DAT peripheral module.

Table 1–4. Generic CSL Functions

Function	Description
*handle = PER*_**open**(*channelNumber,* *[priority]* *flags*)	Opens a peripheral channel and then performs the operation indicated by *flags*; must be called before using a channel. The return value is a unique device handle to use in subsequent API calls. The *priority* parameter applies only to the DAT module.
*PER*_**config**(*[handle,]* **configStructure*)	Writes the values of the configuration structure to the peripheral registers. You can initialize the configuration structure with: ❑ Integer constants ❑ Integer variables ❑ CSL symbolic constants, *PER_REG*_DEFAULT (See Section 1.6 CSL *Symbolic Constant Values*) ❑ Merged field values created with the *PER_REG*_RMK macro
*PER*_**configArgs**(*[handle,]* *regval_1,* . . . *regval_n*)	Writes the individual values (*regval_n*) to the peripheral registers. These values can be any of the following: ❑ Integer constants ❑ Integer variables ❑ CSL symbolic constants, *PER_REG*_DEFAULT ❑ Merged field values created with the *PER_REG*_RMK macro
*PER*_**reset**(*[handle]* *)*	Resets the peripheral to its power-on default values.
*PER*_**close**(*handle*)	Closes a peripheral channel previously opened with *PER*_open(). The registers for the channel are set to their power-on defaults, and any pending interrupt is cleared.

1.4.1 Peripheral Initialization via Registers

The CSL provides two types of functions for initializing the registers of a peripheral: PER_config() and PER_configArgs() (where *PER* is the peripheral as listed in Table 1–1).

- PER_config() initializes the control registers of the PER peripheral, where PER is one of the CSL modules. This function requires an address as its one parameter. The address specifies the location of a structure that represents the peripherals register values. The configuration structure data type is defined for each peripheral module that contains the PER_config() function. Example 1–1 shows an example of this method.

Example 1–1. Using PER_config() with the configuration structure PER_Config

```
PER_Config MyConfig = {
  reg0,
  reg1,
  …
};
…
PER_config(&MyConfig);
```

- PER_configArgs() allows you to pass the individual register values as arguments to the function, which then writes those individual values to the register. Example 1–2 shows an example of this method.

You can use these two initialization functions interchangeably but you still need to generate the register values. To simplify the process of defining the values to write to the peripheral registers, the CSL provides the PER_REG_RMK (make) macros, which form merged values from a list of field arguments. Macros are discussed in Section 1.5, *CSL Macros*.

Example 1–2. Using PER_configArgs

```
PER_configArgs(reg0, reg1, …);
```

1.5 CSL Macros

Table 1–5 provides a generic description of the most common CSL macros, where:

- *PER* indicates a peripheral. (e.g., DMA)
- *REG* indicates, if applicable, a register name (e.g., PRICTL0, AUXCTL)
- *FIELD* indicates a field in a register (e.g., ESIZE)
- *regval* indicates an integer constant, an integer variable, a symbolic constant (*PER_REG*_DEFAULT), or a merged field value created with the peripheral field make macro, *PER*_FMK().
- *fieldval* indicates an integer constant, integer variable, or symbolic constant (*PER_REG_FIELD_SYMVAL*) as explained in section 1.6); all field values are right justified
- *x* indicates an integer constant, integer variable.
- *sym* indicates a symbolic constant
- CSL also offers equivalent macros to those listed in Table 1–5, but instead of using REG to identify which channel the register belongs to, it uses the handle value. The handle value is returned by the PER_open() function (see section 1.7). These macros are shown in Table 1–6.

Each API chapter provides specific descriptions of the macros within that module. Page references to the macros in the hardware abstraction layer (Chapter 24, *Using the HAL Macros*), are provided for additional information.

Table 1–5. Generic CSL Macros

Macro	Description
*PER_REG_***RMK**(*fieldval_n,* . . . *fieldval_0*)	Creates a value to store in the peripheral register; _RMK macros make it easier to construct register values based on field values. The following rules apply to the _RMK macros: ❏ Include only fields that are writable. ❏ Specify field arguments as most-significant bit first. ❏ Whether or not they are used, all writable field values must be included. ❏ If you pass a field value exceeding the number of bits allowed for that particular field, the _RMK macro truncates that field value.
*PER*_**RGET**(*REG*)	Returns the value in the peripheral register.
*PER*_**RSET**(*REG*, *regval*)	Writes the value to the peripheral register.
*PER*_**FMK** *(REG,* FIELD*,* *fieldval* *)*	Creates a shifted version of *fieldval* that you could OR with the result of other _FMK macros to initialize register REG. This allows the user to initialize few fields in REG as an alternative to the _RMK macro, which requires that ALL register fields be initialized.
*PER*_**FGET**(*REG*, *FIELD*)	Returns the value of the specified *FIELD* in the peripheral register.
*PER*_**FSET**(*REG,* *FIELD,* *fieldval*)	Writes *fieldval* to the specified *FIELD* in the peripheral register.
*PER_REG_***ADDR**(*REG*)	If applicable, gets the memory address (or subaddress) of the peripheral register REG.
*PER*_**FSETS** *(REG,* FIELD*,* *sym* *)*	Writes the symbol value to the specified field in the peripheral.
*PER*_**FMKS** *(REG,* FIELD*,* *sym* *)*	Creates a shifted version of the symbol value that you can OR with the result of other _FMK/_FMKS macros to initialize register REG. (See also PER_FMK() macro.)

Table 1–6. Generic CSL Handle-Based Macros

Macro	Description
*PER*_**ADDRH** (h, *REG*)	Returns the address of a memory-mapped register for a given handle.
*PER*_**RGETH** (h**,** *REG*)	Returns the value of a register for a given handle.
*PER*_**RSETH** (h, *REG,* x)	Sets the register value to x for a given handle.
*PER*_**FGETH** (h**,** *REG, FIELD*)	Returns the value of the field for a given handle.
*PER*_F**SETH** (h**,** *REG, FIELD,* x)	Sets the field value to x for a given handle.
*PER*_**FSETSH** (h, *REG, FIELD, SYM*)	Sets the field value to the symbol value for a given handle.

Handle-based CSL macros are applicable to the following peripherals:

- DMA
- EDMA
- GPIO
- McBSP
- TIMER
- I2C
- McASP

1.6 CSL Symbolic Constant Values

To facilitate initialization of values in your application code, the CSL provides symbolic constants for registers and writable field values as described in Table 1–7, where:

- *PER* indicates a peripheral
- *REG* indicates a peripheral register
- *FIELD* indicates a field in the register
- *SYMVAL* indicates the symbolic value of a register field

Each API chapter provides specific descriptions of the symbolic constants within that module. Page references to the constants in the hardware abstraction layer (Chapter 24, *Using the HAL Macros),* are provided for additional information.

Table 1–7. Generic CSL Symbolic Constants

(a) Constant Values for Registers

Constant	Description
*PER_REG_***DEFAULT**	Default value for a register; corresponds to the register value after a reset or to 0 if a reset has no effect.

(b) Constant Values for Fields

Constant	Description
PER_REG_FIELD_SYMVAL	Symbolic constant to specify values for individual fields in the specified peripheral register. See the CSL Registers in Appendix B for the symbolic values.
*PER_REG_FIELD_***DEFAULT**	Default value for a field; corresponds to the field value after a reset or to 0 if a reset has no effect.

1.7 Resource Management

CSL provides a limited set of functions that enable resource management for applications which may support multiple algorithms, such as two McBSP or two TIMERs, and may reuse the same type of peripheral device.

Resource management in CSL is achieved through API calls to the PER_open( ) and PER_close( ) functions. The PER_open( ) function normally takes a device number and a reset flag as the primary arguments and returns a pointer to a handle structure that contains information about which channel (DMA) or port (McBSP) was opened, then set “Allocate” flag defined in the handle structure to 1, meaning the channel or port is in use. When given a specific device number, the open function checks a global “allocate” flag to determine its availability. If the device/channel is available, then it returns a pointer to a predefined handle structure for this device. If the device has already been opened by another process, then an invalid handle is returned whose value is equal to the CSL symbolic constant, INV. Note that CSL does nothing other than return an invalid handle from PER_open( ). You must use this to insure that no resource-usage conflicts occur. It is left to the user to check the value returned from the PER_open( ) function to guarantee that the resource has been allocated.

A device/channel may be freed for use by other processes by a call to PER_close( ). PER_close( ) clears the allocate flag defined under the handle structure object and resets the device/channel.

All CSL modules that support multiple devices or channels, such as McBSP and DMA, require a device handle as a primary argument to most API functions. For these APIs, the definition of a PER_Handle object is required.

1.7.1 Using CSL Handles

Handles are required only for peripherals that have multiple channels or ports, such as DMA, EDMA, GPIO, McBSP, TIMER, I2C, and McASP.

You obtain a handle by calling the PER_open( ) function. When you no longer need a particular channel, free those resources by calling the PER_close( ) function. The PER_open( ) and PER_close( ) functions ensure that you do not initialize the same channel more than once.

CSL handle objects are used to uniquely identify an open peripheral channel/port or device. Handle objects must be declared in the C source, and initialized by a call to a PER_open( ) function before calling any other API functions that require a handle object as an argument. PER_open( ) returns a value of “INV” if the resource is already allocated.

```
DMA_Handle myDma;
/* Defines a DMA_Handle object, myDma */
```

Once defined, the CSL handle object is initialized by a call to PER_open.

```
•
•
myDma = DMA_open (DMA_CHA0, DMA_OPEN_RESET);
/* Open DMA channel 0 */
```

The call to DMA_open initializes the handle, myDma. This handle can then be used in calls to other API functions.

```
if(myDma != INV) {
DMA_start (myDma);                /* Begin transfer */
•
•
DMA_close (myDma); }              /* Free DMA channel */
```

1.7.2 Initializing Registers

The CSL provides two types of functions for initializing the registers of a peripheral: PER_config() and PER_configArgs() (where *PER* is the peripheral as listed in Table 1–1).

- PER_config() initializes the control registers of the PER peripheral, where PER is one of the CSL modules. This function requires an address as its one parameter. The address specifies the location of a structure that represents the peripherals register values. The configuration structure data type is defined for each peripheral module that contains the PER_config() function. Example 1–1 shows an example of this method.

Example 1–3. Using PER_config() with the configuration structure PER_Config

```
PER_Config MyConfig = {
  reg0,
  reg1,
  …
};
…
PER_config(&MyConfig);
```

- PER_configArgs() allows you to pass the individual register values as arguments to the function, which then writes those individual values to the register. Example 1–2 shows an example of this method.

You can use these two initialization functions interchangeably but you still need to generate the register values. To simplify the process of defining the values to write to the peripheral registers, the CSL provides the PER_REG_RMK (make) macros, which form merged values from a list of field arguments. Macros are discussed in Section 1.5, *CSL Macros*.

Example 1–4. Using PER_configArgs

```
PER_configArgs(reg0, reg1, …);
```

Handles are required only for peripherals that have multiple channels or ports, such as DMA, McBSP, TIMER, and EDMA.

You obtain a handle by calling the PER_open( ) function. When you no longer need a particular channel, free those resources by calling the PER_close( ) function. The PER_open( ) and PER_close( ) functions ensure that you do not initialize the same channel more than once.

CSL handle objects are used to uniquely identify an open peripheral channel/ port or device. Handle objects must be declared in the C source, and initialized by a call to a PER_open( ) function before calling any other API functions that require a handle object as an argument. PER_open( ) returns a value of “INV” if the resource is already allocated.

1.8 CSL API Module Support

Not all CSL API modules are supported on all devices. For example, the EDMA API module is not supported on the C6201 because the C6201 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.

Note:

To build the program with the right library, the device support symbol must be set in the compiler option window. For example, if using C6201, the compiler option set in the preprocessor tab would be –dCHIP_6201.

Table 1–8 and Table 1–9 show which devices support the API modules.

Table 1–8. CSL API Module Support for TMS320C6000 Devices

Module	6201	6202	6203	6204	6205	6211	6701	6711	6712	6713	DA610
CACHE	X	X	X	X	X	X	X	X	X	X	X
CHIP	X	X	X	X	X	X	X	X	X	X	X
DAT	X	X	X	X	X	X	X	X	X	X	X
DMA	X	X	X	X	X		X				
EDMA						X		X	X	X	X
EMIF	X	X	X	X	X	X	X	X	X	X	X
GPIO										X	X
HPI	X					X	X	X		X	X
I2C										X	X
IRQ	X	X	X	X	X	X	X	X	X	X	X
McASP										X	X
McBSP	X	X	X	X	X	X	X	X	X	X	X
PCI					X						
PLL										X	X
PWR	X	X	X	X	X	X	X	X	X	X	X
TIMER	X	X	X	X	X	X	X	X	X	X	X
XBUS		X	X	X							

Table 1–9. CSL API Module Support for TMS320C641x Devices

Module	6414	6415	6416
CACHE	X	X	X
CHIP	X	X	X
DAT	X	X	X
DMA			
EDMA	X	X	X
EMIFA	X	X	X
EMIFB	X	X	X
GPIO	X	X	X
HPI	X	X	X
IRQ	X	X	X
McBSP	X	X	X
PCI		X	X
PWR	X	X	X
TCP			X
TIMER	X	X	X
UTOP		X	X
VCP			X
XBUS			

1.8.1 CSL Endianess/Device Support Library

Table 1–10. CSL Device Support Library Name and Symbol Conventions

Device	Little Endian Library	Big Endian Library	Device Support Symbol
C6201	csl6201.lib	csl6201e.lib	CHIP_6201
C6202	csl6202.lib	csl6202e.lib	CHIP_6202
C6203	csl6203.lib	csl6203e.lib	CHIP_6203
C6204	csl6204.lib	csl6204e.lib	CHIP_6204
C6205	csl6205.lib	csl6205e.lib	CHIP_6205
C6211	csl6211.lib	csl6211e.lib	CHIP_6211
C6701	csl6701.lib	csl6701e.lib	CHIP_6701
C6711	csl6711.lib	csl6711e.lib	CHIP_6711
C6712	csl6712.lib	csl6712e.lib	CHIP_6712
C6713	csl6713.lib	csl6713e.lib	CHIP_6713
C6414	csl6414.lib	csl6414e.lib	CHIP_6414
C6415	csl6415.lib	csl6415e.lib	CHIP_6415
C6416	csl6416.lib	csl6416e.lib	CHIP_6416
DA610	cslDA610.lib	cslDA610e.lib	CHIP_DA610

Chapter 2

CACHE Module

This chapter describes the CACHE module, gives a description of the two CACHE architectures, lists the functions and macros within the module, and provides a CACHE API reference section.

Topic **Page**

2.1 Overview

The CACHE module functions are used for managing data and program cache.

Currently, TMS320C6x devices use three cache architectures. The first type, as seen on the C620x device, provides program cache by disabling on-chip program RAM and turning it into cache. The second and third types, seen on C621x/C671x and C64x devices respectively, are the two–level (L2) cache architectures. For the differences between C621x/C671x and C64x cache architectures, refer to *SPRU610 TMS320C64x DSP Two Level Internal Memory Reference Guide*.

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.

Table 2–1 shows the API functions within the CACHE module.

Table 2–1. CACHE APIs

Syntax	Type	Description	See page ...
CACHE_clean†	F	Cleans a specific cache region	2-7
CACHE_enableCaching	F	Enables caching for a specified block of address space	2-8
CACHE_flush†	F	Flushes a region of cache	2-10
CACHE_getL2SramSize	F	Returns current L2 size configured as SRAM	2-11
CACHE_invalidate†	F	Invalidates a region of cache	2-11
CACHE_invAllL1p	F	L1P invalidate all	2-12
CACHE_invL1d	F	L1D block invalidate (C64x only)	2-12
CACHE_invL1p	F	L1P block invalidate	2-13
CACHE_invL2	F	L2 block invalidate (C64x only)	2-14
CACHE_L1D_LINESIZE	C	A compile time constant whose value is the L1D line size.	2-15
CACHE_L1P_LINESIZE	C	A compile time constant whose value is the L1P line size.	2-15

Note: F = Function; C = Constant; M = Macro
† This API function is provided for backward compatibility. Users should use the new APIs.

Table 2–1. CACHE APIs (Continued)

Syntax	Type	Description	See page ...
CACHE_L2_LINESIZE	C	A compile time constant whose value is the L2 line size.	2-16
CACHE_reset	F	Resets cache to power-on default	2-16
CACHE_resetEMIFA	F	Resets the MAR registers dedicated to the EMIFA	2-16
CACHE_resetEMIFB	F	Resets the MAR registers dedicated to the EMIFB	2-17
CACHE_resetL2Queue	F	Resets the queue length of a given queue to default value	2-17
CACHE_ROUND_TO_LINESIZE (CACHE,ELCNT,ELSIZE)	M	Rounds to cache line size	2-17
CACHE_setL2Mode	F	Sets L2 cache mode	2-18
CACHE_setL2Queue	F	Sets the queue length of a given L2 queue	2-21
CACHE_setPriL2Req	F	Sets the L2 requestor priority level	2-21
CACHE_setPccMode	F	Sets program cache mode	2-22
CACHE_SUPPORT	C	A compile time constant whose value is 1 if the device supports the CACHE module	2-22
CACHE_wait	F	Waits for completion of the last cache operation	2-22
CACHE_wbAllL2	F	L2 writeback all	2-23
CACHE_wbInvL1d	F	L1D block writeback and invalidate	2-24
CACHE_wbInvAllL2	F	L2 writeback and invalidate all	2-25
CACHE_wbInvL2	F	L2 block writeback and invalidate	2-26
CACHE_wbL2	F	L2 block writeback	2-27

Note: F = Function; C = Constant; M = Macro
† This API function is provided for backward compatibility. Users should use the new APIs.

2.2 Macros

There are two types of CACHE macros: those that access registers and fields, and those that construct register and field values.

Table 2–2 lists the CACHE macros that access registers and fields, and Table 2–3 lists the CACHE macros that construct register and field values. The macros themselves are found in Chapter 24, *Using the HAL Macros*.

CACHE macros are not handle-based.

Table 2–2. CACHE Macros that Access Registers and Fields

Macro	Description/Purpose	See page...
CACHE_ADDR(<REG>)	Register address	24-12
CACHE_RGET(<REG>)	Returns the value in the peripheral register	24-18
CACHE_RSET(<REG>,x)	Register set	24-20
CACHE_FGET(<REG>,<FIELD>)	Returns the value of the specified field in the peripheral register	24-13
CACHE_FSET(<REG>,<FIELD>,fieldval)	Writes *fieldval* to the specified field in the peripheral register	24-15
CACHE_FSETS(<REG>,<FIELD>,<SYM>)	Writes the symbol value to the specified field in the peripheral	24-17
CACHE_RGETA(addr,<REG>)	Gets register for a given address	24-19
CACHE_RSETA(addr,<REG>,x)	Sets register for a given address	24-20
CACHE_FGETA(addr,<REG>,<FIELD>)	Gets field for a given address	24-13
CACHE_FSETA(addr,<REG>,<FIELD>, fieldval)	Sets field for a given address	24-16
CACHE_FSETSA(addr,<REG>,<FIELD>, <SYM>)	Sets field symbolically for a given address	24-17

Table 2–3. CACHE Macros that Construct Register and Field Values

Macro	Description/Purpose	See page ...
CACHE_<REG>_DEFAULT	Register default value	24-21
CACHE_<REG>_RMK()	Register make	24-23
CACHE_<REG>_OF()	Register value of ...	24-22
CACHE_<REG>_<FIELD>_DEFAULT	Field default value	24-24
CACHE_FMK()	Field make	24-14
CACHE_FMKS()	Field make symbolically	24-15
CACHE_<REG>_<FIELD>_OF()	Field value of ...	24-24
CACHE_<REG>_<FIELD>_<SYM>	Field symbolic value	24-24

2.3 Functions

CACHE_clean *Cleans a range of L2 cache*

> **Note:**
>
> This function is provided for backward compatibility only. The user is strongly advised to use the new functions as shown in Appendix D.

Function

```
void CACHE_clean(
    CACHE_Region region,
    void *addr,
    Uint32 wordCnt
);
```

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

wordCnt — Number of 32-bit words to clean. IMPORTANT: Maximum allowed wordCnt is 65535.

Return Value

none

Description

Cleans a range of L2 cache. All lines within the range defined by `addr` and `wordCnt` are cleaned out of L2. If `CACHE_L2ALL` is specified, then all of L2 is cleaned, `addr` and `wordCnt` are ignored. A clean operation involves writing back all dirty cache lines and then invalidating 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,(void*)0x80000000,0x00000400);
```

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

```
CACHE_clean(CACHE_L2ALL,(void*)0x00000000,0x00000000);
```

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 be one of the following:

For devices other than C64x–

- ❏ 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)

For C64x, EMIFB, CE0, CE1, CE2, and CE3–

- ❏ CACHE_EMIFB_CE00 –(60000000h to 60FFFFFFh)
- ❏ CACHE_EMIFB_CE01 –(61000000h to 61FFFFFFh)
- ❏ CACHE_EMIFB_CE02 –(62000000h to 62FFFFFFh)
- ❏ CACHE_EMIFB_CE03 –(63000000h to 63FFFFFFh)
- ❏ CACHE_EMIFB_CE010 –(64000000h to 64FFFFFFh)
- ❏ CACHE_EMIFB_CE011 –(65000000h to 65FFFFFFh)
- ❏ CACHE_EMIFB_CE012 –(66000000h to 66FFFFFFh)
- ❏ CACHE_EMIFB_CE013 –(67000000h to 67FFFFFFh)
- ❏ CACHE_EMIFB_CE020 –(68000000h to 68FFFFFFh)
- ❏ CACHE_EMIFB_CE021 –(69000000h to 69FFFFFFh)
- ❏ CACHE_EMIFB_CE022 –(6A000000h to 6AFFFFFFh)
- ❏ CACHE_EMIFB_CE023 –(6B000000h to 6BFFFFFFh)
- ❏ CACHE_EMIFB_CE030 –(6C000000h to 6CFFFFFFh)
- ❏ CACHE_EMIFB_CE031 –(6D000000h to 6DFFFFFFh)
- ❏ CACHE_EMIFB_CE032 –(6E000000h to 6EFFFFFFh)
- ❏ CACHE_EMIFB_CE033 –(6F000000h to 6FFFFFFFh)

For EMIFA CE0–

- CACHE_EMIFA_CE00 –(80000000h to 80FFFFFFh)
- CACHE_EMIFA_CE01 –(81000000h to 81FFFFFFh)
- CACHE_EMIFA_CE02 –(82000000h to 82FFFFFFh)
- CACHE_EMIFA_CE03 –(83000000h to 83FFFFFFh)
- CACHE_EMIFA_CE04 –(84000000h to 84FFFFFFh)
- CACHE_EMIFA_CE05 –(85000000h to 85FFFFFFh)
- CACHE_EMIFA_CE06 –(86000000h to 86FFFFFFh)
- CACHE_EMIFA_CE07 –(87000000h to 87FFFFFFh)
- CACHE_EMIFA_CE08 –(88000000h to 88FFFFFFh)
- CACHE_EMIFA_CE09 –(89000000h to 89FFFFFFh)
- CACHE_EMIFA_CE010 –(8A000000h to 8AFFFFFFh)
- CACHE_EMIFA_CE011 –(8B000000h to 8BFFFFFFh)
- CACHE_EMIFA_CE012 –(8C000000h to 8CFFFFFFh)
- CACHE_EMIFA_CE013 –(8D000000h to 8DFFFFFFh)
- CACHE_EMIFA_CE014 –(8E000000h to 8EFFFFFFh)
- CACHE_EMIFA_CE015 –(8F000000h to 8FFFFFFFh)

For CACHE_EMIFA_CE1, CACHE_EMIFA_CE2, and CACHE_EMIFA_CE3 the symbols are the same as CACHE_EMIFA_CE0, with start addresses 90000000h, A0000000h, and B0000000h, respectively.

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);
```

CACHE_flush *Flushes region of cache (obsolete)*

> **Note:**
>
> This function is provided for backward compatibility only. The user is strongly advised to use the new functions as shown in Appendix D.

Function

```
void CACHE_flush(
    CACHE_Region region,
    void *addr,
    Uint32 wordCnt
);
```

Arguments

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

- ❑ CACHE_L2
- ❑ CACHE_L2ALL
- ❑ CACHE_L1D

addr — Beginning address of range to flush; word aligned

wordCnt — Number of 32-bit words to flush. IMPORTANT: Maximum allowed wordCnt is 65535.

Return Value

none

Description

Flushes a range of L2 cache. All lines within the range defined by `addr` and `wordCnt` are flushed out of L2. If `CACHE_L2ALL` is specified, then all of L2 is flushed; `addr` and `wordCnt` 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,(void*)0x80000000,0x00000400);
```

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

```
CACHE_flush(CACHE_L2ALL,(void*)0x00000000,0x00000000);
```

CACHE_getL2SramSize *Returns current size of L2 that is 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();
```

CACHE_invalidate *Invalidates a region of cache (obsolete)*

Note:

This function is provided for backward compatibility only. The user is strongly advised to use the new functions as shown in Appendix D.

Function

```
void CACHE_invalidate(
    CACHE_Region region,
    void *addr,
    Uint32 wordCnt
);
```

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

wordCnt Number of 32-bit words to invalidate. IMPORTANT: Maximum allowed wordCnt is 65535.

Return Value none

Description Invalidates a range from cache. All lines within the range defined by `addr` and `wordCnt` are invalidated from `region`. If `CACHE_L1PALL` is specified, then all of L1P is invalidated; `addr` and `wordCnt` are ignored. Likewise, if `CACHE_L1DALL` is specified, then all of L1D is invalidated; `addr` and `wordCnt` 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_invalidate(CACHE_L1P,(void*)0x80000000,0x00000400);
```

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

```
CACHE_invalidate(CACHE_L1DALL,(void*)0x00000000,0x00000000);
```

CACHE_invAllL1p *L1P invalidates all*

Function

```
void CACHE_invAllL1p();
```

Arguments none

Return Value none

Description This function issues an L1P invalidate all command to the cache controller. Please see the *TMS320C621x/C671x DSP Two Level Internal Memory Reference Guide* (literature number SPRU609) and the *TMS320C64x DSP Two Level Internal Memory Reference Guide* (literature number SPRU610)for details of this operation.

Example

```
CACHE_invAllL1p();
```

CACHE_invL1d *L1D block invalidate (C64x only)*

Function

```
void CACHE_invL1d(
    void    *blockPtr,
    Uint32  byteCnt,
    int     wait
);
```

Arguments

blockPtr — Pointer to the beginning of the block

byteCnt — Number of bytes in the block. This value must be a multiple of four. The largest size this can be in 65535*4.

wait — Wait flag:

- ❑ CACHE_NOWAIT – return immediately
- ❑ CACHE_WAIT – wait until the operation completes

Return Value none

Description

This function issues an L1D block invalidate command to the cache controller. Please see the *TMS320C64x DSP Two Level Internal Memory Reference Guide* (literature number SPRU610)for details of this operation. If a previous cache operation is still active, the function waits for its completion before initiating the new operation, in order to prevent lockout of interrupts.

If the user specifies CACHE_NOWAIT, then the function returns immediately, regardless of whether the operation has completed. The user can call `CACHE_wait()` afterwards to wait for the operation to complete.

Although the block size can be specified in number of bytes, the cache controller operates on whole cache lines only.

This function is only supported on C64x devices.

Example

```
char buffer[1024];
/* call with wait flag set */
CACHE_invL1d(buffer, 1024, CACHE_WAIT);
...
/* call without the wait flag set */
CACHE_invL1d(buffer, 1024, CACHE_NOWAIT);
...
...
CACHE_wait();
```

CACHE_invL1p *L1P block invalidate*

Function

```
void CACHE_invL1p(
    void    *blockPtr,
    Uint32  byteCnt,
    int     wait
);
```

Arguments

blockPtr — Pointer to the beginning of the block

byteCnt — Number of bytes in the block. This value must be a multiple of four. The largest size this can be in 65535*4.

wait — Wait flag:

- ❑ CACHE_NOWAIT – return immediately
- ❑ CACHE_WAIT – wait until the operation completes

Return Value

none

Description

This function issues an L1P block invalidate command to the cache controller. Please see the *TMS320C621x/C671x DSP Two Level Internal Memory Reference Guide* (literature number SPRU609) and the *TMS320C64x DSP Two Level Internal Memory Reference Guide* (literature number SPRU610)for details of this operation. If a previous cache operation is still active, the function waits for its completion before initiating the new operation, in order to prevent lockout of interrupts.

If the user specifies CACHE_NOWAIT, then the function returns immediately, regardless of whether the operation has completed. The user can call `CACHE_wait()` afterwards to wait for the operation to complete.

Although the block size can be specified in number of bytes, the cache controller operates on whole cache lines only.

Example

```
char buffer[1024];
/* call with wait flag set */
CACHE_invL1p(buffer, 1024, CACHE_WAIT);
...
/* call without the wait flag set */
CACHE_invL1p(buffer, 1024, CACHE_NOWAIT);
...
...
CACHE_wait();
```

CACHE_invL2 *L2 block invalidate (C64x devices only)*

Function

```
void CACHE_invL2(
    void    *blockPtr,
    Uint32  byteCnt,
    int     wait
);
```

Arguments

blockPtr — Pointer to the beginning of the block

byteCnt — Number of bytes in the block. This value must be a multiple of four. The largest size this can be in 65535*4.

wait — Wait flag:

- ❑ CACHE_NOWAIT – return immediately
- ❑ CACHE_WAIT – wait until the operation completes

Return Value

none

Description

This function issues an L2 block invalidate command to the cache controller. Please see the *TMS320C64x DSP Two Level Internal Memory Reference Guide* (literature number SPRU610)for details of this operation. If a previous cache operation is still active, the function waits for its completion before initiating the new operation, in order to prevent lockout of interrupts.

If the user specifies CACHE_NOWAIT, then the function returns immediately, regardless of whether the operation has completed. The user can call `CACHE_wait()` afterwards to wait for the operation to complete.

Although the block size can be specified in number of bytes, the cache controller operates on whole cache lines only. To prevent unintended behavior, blockPtr and byteCnt should be multiples of the cache line size.

This function is supported on C64x devices only.

Example

```
char buffer[1024];
/* call with wait flag set */
CACHE_invL2(buffer, 1024, CACHE_WAIT);
...
/* call without the wait flag set */
CACHE_invL2(buffer, 1024, CACHE_NOWAIT);
...
...
CACHE_wait();
```

CACHE_L1D_LINESIZE

L1D line size

Constant

CACHE_L1D_LINESIZE

Description

Compile-time constant that is set equal to the L1D cache line size of the device.

Example

```
#pragma DATA_ALIGN(array, CACHE_L1D_LINESIZE)
```

CACHE_L1P_LINESIZE

L1P line size

Constant

CACHE_L1P_LINESIZE

Description

Compile-time constant that is set equal to the L1P cache line size of the device.

Example

```
#pragma DATA_ALIGN(array, CACHE_L1P_LINESIZE)
```

CACHE_L2_LINE-SIZE

L2 line size

Constant CACHE_L2_LINESIZE

Description Compile-time constant that is set equal to the L2 cache line size of the device.

Example

```
#pragma DATA_ALIGN(array, CACHE_L2_LINESIZE)
```

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: All MAR bits are cleared

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();
```

CACHE_resetEMIFA

Resets the MAR registers dedicated to the EMIFA CE spaces

Function void CACHE_resetEMIFA();

Arguments none

Return Value none

Description This function resets the MAR registers dedicated to the EMIFA CE spaces.

Example

```
CACHE_enableCaching(CACHE_EMIFA_CE00);
CACHE_enableCaching(CACHE_EMIFA_CE13);
CACHE_resetEMIFA();
```

CACHE_resetEMIFB *Resets the MAR registers dedicated to the EMIFB CE spaces*

Function void CACHE_resetEMIFB();

Arguments none

Return Value none

Description This function resets all the MAR registers dedicated to the EMIFB CE spaces.

Example

```
CACHE_enableCaching(CACHE_EMIFB_CE00);
CACHE_enableCaching(CACHE_EMIFB_CE13);
CACHE_resetEMIFB();
```

CACHE_resetL2Queue *Resets the queue length of the L2 queue to its default value*

Function

```
void CACHE_resetL2Queue(
    Uint32 queueNum
);
```

Arguments queueNum Queue number to be reset to the default length: The following constants may be used for L2 queue number:

- ❏ CACHE_L2Q0
- ❏ CACHE_L2Q1
- ❏ CACHE_L2Q2
- ❏ CACHE_L2Q3

Return Value none

Description This functions allows the user to reset the queue length of the given L2 queue to its default value. See the CACHE_setL2Queue() function.

Example

```
EDMA_setL2Queue(CACHE_L2Q2,4);
EDMA_resetL2Queue(CACHE_L2Q2);
```

CACHE_ROUND_TO_LINESIZE *Rounds to cache line size*

Macro

```
CACHE_ROUND_TO_LINESIZE(
    CACHE,
    ELCNT,
    ELSIZE
);
```

Arguments

CACHE Cache type: L1D, L1P, or L2

ELCNT Element count

ELSIZE Element size

Return Value Rounded up element count

Description This macro rounds an element up to make an array size a multiple number of cache lines.

Arrays located in external memory that require user-controlled coherence maintenance must be aligned at a cache line boundary and be a multiple of cache lines large to prevent incoherency problems. Please see the *TMS320C6000 DSP Cache User's Guide* (literature number SPRU656) for details.

Example

```
/* assume an L2 line size of 128 bytes */
/* align arrays y and x at the cache line border */
#pragma DATA_ALIGN(y, CACHE_L2_LINESIZE)
#pragma DATA_ALIGN(x, CACHE_L2_LINESIZE)
/* array y spans 7 full lines and 104 bytes of the next line*/
short y[500];
/* the array element count is increased such that the array x
   spans a multiple number of cache lines, i.e. 8 lines */
short x[CACHE_ROUNT_TO_LINESIZE(L2, 500, sizeof(short))]
```

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:

(For C6711/C6211)

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

(For C6713 and DA610)

- CACHE_256KSRAM

- ❑ CACHE_0KCACHE
- ❑ CACHE_240KSRAM
- ❑ CACHE_16KCACHE
- ❑ CACHE_224KSRAM
- ❑ CACHE_32KCACHE
- ❑ CACHE_208KSRAM
- ❑ CACHE_48KCACHE
- ❑ CACHE_192KSRAM
- ❑ CACHE_64KCACHE

(For C6414/C6415/C6416)

- ❑ CACHE_1024KSRAM
- ❑ CACHE_0KCACHE
- ❑ CACHE_992KSRAM
- ❑ CACHE_32KCACHE
- ❑ CACHE_960KSRAM
- ❑ CACHE_64KCACHE
- ❑ CACHE_896KSRAM
- ❑ CACHE_128KCACHE
- ❑ CACHE_768KSRAM
- ❑ CACHE_256KCACHE

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

(For C6711/C6211)

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

(For C6713 and DA610)

- ❑ CACHE_256KSRAM
- ❑ CACHE_0KCACHE
- ❑ CACHE_240KSRAM
- ❑ CACHE_16KCACHE
- ❑ CACHE_224KSRAM
- ❑ CACHE_32KCACHE
- ❑ CACHE_208KSRAM

- ❏ CACHE_48KCACHE
- ❏ CACHE_192KSRAM
- ❏ CACHE_64KCACHE

(For C6414/C6415/C6416)

- ❏ CACHE_1024KSRAM
- ❏ CACHE_0KCACHE
- ❏ CACHE_992KSRAM
- ❏ CACHE_32KCACHE
- ❏ CACHE_960KSRAM
- ❏ CACHE_64KCACHE
- ❏ CACHE_896KSRAM
- ❏ CACHE_128KCACHE
- ❏ CACHE_768KSRAM
- ❏ CACHE_256KCACHE

Description

This function 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 writeback-invalidated, then the mode is changed. If the cache size increases, the part of SRAM that is about to be turned into cache is writeback-invalidated from L1D and all of L2 is writeback-invalidated; then the mode is changed. Nothing happens when there is no change.

Increasing cache size means that some of the SRAM is lost. If there is data in the SRAM that should not be lost, it must be preserved before changing cache modes. Some of the cache modes are identical. For example, on the C6211, there are 64KBytes 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);
```

CACHE_setL2Queue *Sets the queue length of the L2 queue*

Function

```
void CACHE_setL2Queue(
    Uint32 queueNum;
    Uint32 length
);
```

Arguments

queueNum — Queue number to be set. The following constants may be used for L2 queue number:

- CACHE_L2Q0
- CACHE_L2Q1
- CACHE_L2Q2
- CACHE_L2Q3

length — Queue length to be set

Return Value

none

Description

This function allows the user to set the queue length of a specified L2 also CACHE_resetL2Queue() function.

Example

```
CACHE_setL2Queue(CACHE_L2Q1,5);
```

CACHE_setPriL2Req *Sets the L2 priority level "P" of the CCFG register*

Function

```
void CACHE_setPriL2Req(
    Uint32 priority
);
```

Arguments

priority — Priority request level to be set. The following constants may be used:

- CACHE_L2PRIURG (0)
- CACHE_L2PRIHIGH (1)
- CACHE_L2PRIMED (2)
- CACHE_L2PRILOW (3)

Return Value

none

Description

This function allows the user to set the L2 priority level "P" of the CCFG register.

Example

```
CACHE_setPriL2Req(CACHE_L2PRIHIGH);
```

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_PCC_MAPPED
- ❑ CACHE_PCC_ENABLE

Return Value

OldMode Returns the old program cache mode; will be one of the following:
- ❑ CACHE_PCC_MAPPED
- ❑ CACHE_PCC_ENABLE

Description

This function sets the program cache mode for devices that do not have an L2 cache. For devices that do have an L2 cache such as the C6211, this function does nothing. See the *TMS320C6000 Peripherals Reference Guide* (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_PCC_ENABLE);
```

CACHE_SUPPORT *Compile time constant*

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
```

CACHE_wait *Waits for completion of the last cache operation*

Function

int CACHE_wait();

Arguments

none

Return Value

none

Description This function waits for the completion of the last cache operation. This function ONLY works in conjunction with the following operations:

- CACHE_wbL2()
- CACHE_invL2()
- CACHE_wbInvL2()
- CACHE_wbAllL2()
- CACHE_wbInvAllL2()
- CACHE_invL1d()
- CACHE_wbInvL1d()
- CACHE_invL1p()

Example

```
CACHE_wbInvAllL2(CACHE_NOWAIT);
...
...
CACHE_wait();
```

CACHE_wbAllL2 *L2 writeback all*

Function

```
void CACHE_wbAllL2(
    int         wait
);
```

Arguments

wait Wait flag:
- CACHE_NOWAIT – return immediately
- CACHE_WAIT – wait until the operation completes

Return Value none

Description This function issues an L2 writeback all command to the cache controller. Please see the *TMS320C621x/C671x DSP Two Level Internal Memory Reference Guide* (literature number SPRU609) and the *TMS320C64x DSP Two Level Internal Memory Reference Guide* (literature number SPRU610)for details of this operation.

If the user specifies CACHE_NOWAIT, then the function returns immediately, regardless of whether the operation has completed. The user can call `CACHE_wait()` afterwards to wait for the operation to complete.

Example

```
/* call with wait flag set */
CACHE_wbAllL2(CACHE_WAIT);
...
/* call without the wait flag set */
CACHE_wbAllL2(CACHE_NOWAIT);
...
CACHE_wait();
```

CACHE_wbInvL1d *L1D block writeback and invalidate*

Function

```
void CACHE_wbInvL1d(
    void    *blockPtr,
    Uint32  byteCnt,
    int     wait
);
```

Arguments

blockPtr — Pointer to the beginning of the block

byteCnt — Number of bytes in the block. This value must be a multiple of four. The largest size this can be in 65535*4.

wait — Wait flag:

- CACHE_NOWAIT – return immediately
- CACHE_WAIT – wait until the operation completes

Return Value

none

Description

This function issues an L1D block writeback and invalidate command to the cache controller. Please see the *TMS320C621x/C671x DSP Two Level Internal Memory Reference Guide* (literature number SPRU609) and the *TMS320C64x DSP Two Level Internal Memory Reference Guide* (literature number SPRU610)for details of this operation. If a previous cache operation is still active, the function waits for its completion before initiating the new operation, in order to prevent lockout of interrupts.

If the user specifies CACHE_NOWAIT, then the function returns immediately, regardless of whether the operation has completed. The user can call `CACHE_wait()` afterwards to wait for the operation to complete.

Although the block size can be specified in number of bytes, the cache controller operates on whole cache lines only.

Example

```
char buffer[1024];
/* call with wait flag set */
CACHE_wbInvL1d(buffer, 1024, CACHE_WAIT);
...
/* call without the wait flag set */
CACHE_wbInvL1d(buffer, 1024, CACHE_NOWAIT);
...
...
CACHE_wait();
```

CACHE_wbInvAllL2 *L2 writeback and invalidate all*

Function

```
void CACHE_wbInvAllL2(
    int         wait
);
```

Arguments

wait — Wait flag:

- ❑ CACHE_NOWAIT – return immediately
- ❑ CACHE_WAIT – wait until the operation completes

Return Value

none

Description

This function issues an L2 writeback and invalidate all command to the cache controller. Please see the *TMS320C621x/C671x DSP Two Level Internal Memory Reference Guide* (literature number SPRU609) and the *TMS320C64x DSP Two Level Internal Memory Reference Guide* (literature number SPRU610)for details of this operation.

If the user specifies CACHE_NOWAIT, then the function returns immediately, regardless of whether the operation has completed. The user can call `CACHE_wait()` afterwards to wait for the operation to complete.

Example

```
/* call with wait flag set */
CACHE_wbInvAllL2(CACHE_WAIT);
...
/* call without the wait flag set */
CACHE_wbInvAllL2(CACHE_NOWAIT);
...
...
CACHE_wait();
```

CACHE_wbInvL2 *L2 block writeback and invalidate*

Function

```
void CACHE_wbInvL2(
    void    *blockPtr,
    Uint32  byteCnt,
    int     wait
);
```

Arguments

blockPtr — Pointer to the beginning of the block

byteCnt — Number of bytes in the block. This value must be a multiple of four. The largest size this can be in 65535*4.

wait — Wait flag:

- ❑ CACHE_NOWAIT – return immediately
- ❑ CACHE_WAIT – wait until the operation completes

Return Value

none

Description

This function issues an L2 block writeback and invalidate command to the cache controller. Please see the *TMS320C621x/C671x DSP Two Level Internal Memory Reference Guide* (literature number SPRU609) and the *TMS320C64x DSP Two Level Internal Memory Reference Guide* (literature number SPRU610)for details of this operation. If a previous cache operation is still active, the function waits for its completion before initiating the new operation, in order to prevent lockout of interrupts.

If the user specifies CACHE_NOWAIT, then the function returns immediately, regardless of whether the operation has completed. The user can call `CACHE_wait()` afterwards to wait for the operation to complete.

Although the block size can be specified in number of bytes, the cache controller operates on whole cache lines only. To prevent unintended behavior, blockPtr and byteCnt should be multiples of the cache line size.

Example

```
char buffer[1024];
/* call with wait flag set */
CACHE_wbInvL2(buffer, 1024, CACHE_WAIT);
...
/* call without the wait flag set */
CACHE_wbInvL2(buffer, 1024, CACHE_NOWAIT);
...
...
CACHE_wait();
```

CACHE_wbL2 *L2 block writeback*

Function

```
void CACHE_wbL2(
    void    *blockPtr,
    Uint32  byteCnt,
    int     wait
);
```

Arguments

blockPtr — Pointer to the beginning of the block

byteCnt — Number of bytes in the block. This value must be a multiple of four. The largest size this can be in 65535*4.

wait — Wait flag:

- ❑ CACHE_NOWAIT – return immediately
- ❑ CACHE_WAIT – wait until the operation completes

Return Value

none

Description

This function issues an L2 block writeback command to the cache controller. Please see the *TMS320C621x/C671x DSP Two Level Internal Memory Reference Guide* (literature number SPRU609) and the *TMS320C64x DSP Two Level Internal Memory Reference Guide* (literature number SPRU610) for details of this operation for details of this operation. If a previous cache operation is still active, the function waits for its completion before initiating the new operation, in order to prevent lockout of interrupts.

If the user specifies CACHE_NOWAIT, then the function returns immediately, regardless of whether the operation has completed. The user can call `CACHE_wait()` afterwards to wait for the operation to complete.

Although the block size can be specified in number of bytes, the cache controller operates on whole cache lines only. To prevent unintended behavior, blockPtr and byteCnt should be multiples of the cache line size.

Example

```
char buffer[1024];
/* call with wait flag set */
CACHE_wbL2(buffer, 1024, CACHE_WAIT);
...
/* call without the wait flag set */
CACHE_wbL2(buffer, 1024, CACHE_NOWAIT);
...
...
CACHE_wait();
```

Chapter 3

CHIP Module

This chapter describes the CHIP module, lists the API functions and macros within the module, and provides a CHIP API reference section.

Topic **Page**

3.1 Overview

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.

Table 3–1 shows the API functions within the CHIP module.

Table 3–1. CHIP APIs

Syntax	Type	Description	See page ...
CHIP_6XXX	C	Current device identification symbols	3-4
CHIP_getCpuId	F	Returns the CPU ID field of the CSR register	3-5
CHIP_getEndian	F	Returns the current endian mode of the device	3-5
CHIP_getMapMode	F	Returns the current map mode of the device	3-6
CHIP_getRevId	F	Returns the CPU revision ID	3-6
CHIP_SUPPORT	C	A compile time constant whose value is 1 if the device supports the CHIP module	3-6

Note: F = Function; C = Constant

3.2 Macros

There are two types of CHIP macros: those that access registers and fields, and those that construct register and field values.

Table 3–2 lists the CHIP macros that access registers and fields, and Table 3–3 lists the CHIP macros that construct register and field values. The macros themselves are found in Chapter 24, *Using the HAL Macros*.

CHIP macros are not handle-based.

Table 3–2. CHIP Macros that Access Registers and Fields

Macro	Description/Purpose	See page...
CHIP_CRGET(<REG>)	Gets the value of CPU register	24-12
CHIP_CRSET(<REG>,x)	Sets the value of CPU register	24-13
CHIP_RGET(<REG>)	Returns the value in the memory-mapped register	24-18
CHIP_RSET(<REG>,x)	Writes the value to the memory-mapped register	24-20
CHIP_FGET(<REG>,<FIELD>)	Returns the value of the specified field in the register	24-13
CHIP_FSET(<REG>,<FIELD>,fieldval)	Writes *fieldval* to the specified field of the register	24-15
CHIP_FSETS(<REG>,<FIELD>,<SYM>)	Writes the symbol value to the specified field in the peripheral	24-17

Table 3–3. CHIP Macros that Construct Register and Field Values

Macro	Description/Purpose	See page...
CHIP_<REG>_DEFAULT	Register default value	24-21
CHIP_<REG>_RMK()	Register make	24-23
CHIP_<REG>_OF()	Register value of ...	24-22
CHIP_<REG>_<FIELD>_DEFAULT	Field default value	24-24
CHIP_FMK()	Field make	24-14
CHIP_FMKS()	Field make symbolically	24-15
CHIP_<REG>_<FIELD>_OF()	Field value of ...	24-24
CHIP_<REG>_<FIELD>_<SYM>	Field symbolic value	24-24

3.3 Functions

CHIP_6XXX

Current chip identification symbols

Constant

CHIP_6201
CHIP_6202
CHIP_6203
CHIP_6204
CHIP_6205
CHIP_6211
CHIP_6414
CHIP_6415
CHIP_6416
CHIP_6701
CHIP_6711
CHIP_6712
CHIP_6713
CHIP_DA610

Description

These are the current chip identification symbols. They are used throughout the CSL code to make compile-time decisions. When using the CSL, you have to select the right chip type under Global Setting module. The chip type will generate the associated macro CHIP_6XXX.

You may also use these symbols to perform conditional compilation; for example:

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

CHIP_getCpuId *Returns CPU ID field of 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();
```

CHIP_getEndian *Returns current endian mode of 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;
0
Endian = CHIP_getEndian();
if (Endian == CHIP_ENDIAN_BIG) {
  /* user big endian configuration /
} else {
  / user little endian configuration */
}
```

CHIP_getMapMode — *Returns current map mode of 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;
0
MapMode = CHIP_getMapMode();
if (MapMode == CHIP_MAP_0) {
  /* user map 0 configuration /
} else {
  / user map 1 configuration */
}
```

CHIP_getRevId — *Returns 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();
```

CHIP_SUPPORT — *Compile-time constant*

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
```

Chapter 4

CSL Module

This chapter describes the CSL module, shows the single API function within the module, and provides a CSL API reference section.

Topic **Page**

4.1 Overview

The CSL module is the top-level API module whose primary purpose is to initialize the library.

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

Table 4–1 shows the only function exported by the CSL module.

Table 4–1. CSL API

Syntax	Type	Description	See page ...
CSL_init	F	Initializes the CSL library	4-3

Note: F = Function

4.2 Functions

CSL_init *Calls 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();
```

Chapter 5

DAT Module

This chapter describes the DAT module, lists the API functions within the DAT module, discusses how the DAT module manages the DMA/EDMA peripheral, and provides a DAT API reference section.

Topic **Page**

5.1 Overview

The data module (DAT) 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 as for devices that have the EDMA peripheral. Therefore, application code that uses the DAT module is compatible across all current devices regardless of which type of DMA controller it has.

Table 5–1 shows the API functions within the DAT module.

Table 5–1. DAT APIs

Syntax	Type	Description	See page ...
DAT_busy	F	Checks to see if a previous transfer has completed	5-4
DAT_close	F	Closes the DAT module	5-4
DAT_copy	F	Copies a linear block of data from Src to Dst using DMA or EDMA hardware	5-5
DAT_copy2d	F	Performs a 2-dimensional data copy using DMA or EDMA hardware.	5-6
DAT_fill	F	Fills a linear block of memory with the specified fill value using DMA or EDMA hardware	5-7
DAT_open	F	Opens the DAT module	5-9
DAT_setPriority	F	Sets the priority CPU vs DMA/EDMA	5-11
DAT_SUPPORT	C	A compile time constant whose value is 1 if the device supports the DAT module	5-11
DAT_wait	F	Waits for a previous transfer to complete	5-12

Note: F = Function; C = Constant

5.1.1 DAT Routines

The DAT module has been intentionally kept simple. There are routines to copy data from one location to another and routines to 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 can block on completion of the operation before moving on.

5.1.2 DAT Macros

There are no register and field access macros dedicated to the DAT module. The only macros used by DAT are equivalent to the DMA or EDMA macros.

5.1.3 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 that this does not 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 with `DAT_close()`.

Note:

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

5.1.4 Devices With DMA

On devices that have the DMA peripheral, such as the 6201, only one request may be active at once since only one DMA channel is used. If you submit two requests one after the other, 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.

5.1.5 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()` function returns a unique transfer ID number. This ID number is then used by the user so that the transfer can be completed. 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).

5.2 Functions

DAT_busy — *Checks to see if a previous transfer has completed*

Function

Uint32 DAT_busy(
    Uint32 id);

Arguments

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

Return Value

busy Returns non-zero if transfer is still busy, zero otherwise.

Description

Checks to see if a previous transfer has completed or not, identified by the transfer ID.

Example

```
DAT_open(DAT_CHAANY, DAT_PRI_LOW, 0);
...
transferId = DAT_copy(src,dst,len);
...
while (DAT_busy(transferId));
```

DAT_close — *Closes 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 used by the DAT module are closed.

Example

```
DAT_close();
```

DAT_copy

Copies 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

xfrId	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 in devices other than C64x devices, you should ensure that the source and destination addresses are aligned on a 4-byte boundary and the transfer length is a multiple of four. A maximum of 65,535 bytes may be copied. A `byteCnt` of zero has unpredictable results.

For C64x devices, the EDMA uses a 64-bit bus (8 bytes) to L2 SRAM. For best efficiency, the source and destination addresses should be aligned on an 8-byte boundary, with the transfer rate a multiple of eight.

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);
…
```

DAT_copy2d *Performs 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

xfrId	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 on devices other than C64x devices, you should ensure that the source address and destination address are aligned on a 4-byte boundary and that the `lineLen` and `linePitch` are multiples of 4-bytes.

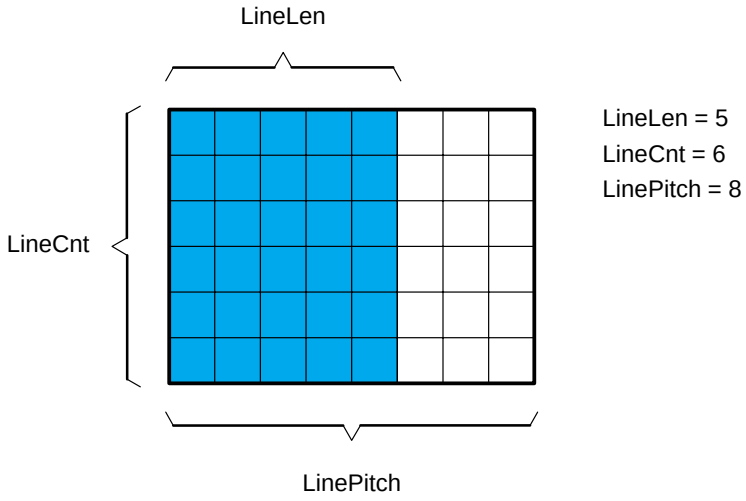
For C64x devices, the EDMA uses a 64-bit bus (8 bytes) to L2 SRAM. For best efficiency, the source and destination addresses should be aligned on an 8-byte boundary with the transfer rate a multiple of eight.

If the channel is busy with previous requests, this function will block (spin) and wait until 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 three 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` × `lineCnt`. The 1D part of the transfer is just a linear block of data. The 2D part is illustrated in Figure 5–1.

Figure 5–1. 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);
```

DAT_fill

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

Function

```
Uint32 DAT_fill(
    void *dst,
    Uint16 byteCnt,
    Uint32 *fillValue
);
```

Arguments

dst	Pointer to destination location
byteCnt	Number of bytes to fill
fillValue	Pointer to fill value

DAT_fill

Return Value xfrId 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.

For devices other than C64x devices, 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 to eliminate any endian issues.

For C64x devices, the fill count must be a multiple of 8 bytes. The EDMA uses a 64-bit bus to store data in L2 SRAM. A pointer of 64-bit value must be passed to the “fillvalue” parameter (a set of 8 consecutive bytes, aligned). The EDMA transfer element size is set to 64-bits. If you want to fill the memory region with a value of 0x1234, the pointer should point to two consecutive 32-bit words set to 0x12341234 value .

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);
```

For 64x devices:

```
Uint32  BUFF_SIZE 256; /* 8 * 8bytes */
Uint32 buff[BUFF_SIZE/sizeof(Uint32)];
Uint32  fillValue[2] = {0x12341234,0x12341234};
Uint32 *fillValuePtr = fillValue;
…
DAT_open(DAT_CHAANY,DAT_PRI_LOW,0);
DAT_fill(buff,BUFF_SIZE,&fillValue);
```

DAT_open — *Opens DAT module*

Function

```
Uint32 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

success — Returns zero on failure and non-zero if successful. Reasons for failure are:

- ❏ The DAT module is already open.
- ❏ Required resources could not be allocated.

DAT_open

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 does not have a channel associated with it.

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()` to free 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.

Note:

For devices with EDMA, the DAT module uses the EDMA registers to submit transfer requests. Also used is the channel interrupt pending register (CIPR). Interrupts are not enabled but the completion flags in CIPR are used. The DAT module uses interrupt completion codes 1 through 4 which amounts to a mask of 0x00000001E in the CIPR register. If you use the DAT module on devices with EDMA, you must avoid using transfer completion codes 1 through 4.

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);
```

DAT_setPriority — *Sets the priority of DMA or EDMA channel*

Function

```
void DAT_setPriority(
int priority
);
```

Arguments

priority Priority bit value

Return Value

none

Description

Sets the priority bit value PRI of PRICTL register for devices supporting DMA, and the PRI of OPT register for devices supporting EDMA. See also `DAT_open()` function. The priority value can be set by using the following predefined constants:

- ❏ DAT_PRI_LOW
- ❏ DAT_PRI_HIGH

Example

```
/* Open DAT channel with priority Low */
DAT_open(DMA_CHAANY,DAT_PRI_LOW,0)
/* Set transfer with priority high */
DAT_setPriority(DAT_PRI_HI);
```

DAT_SUPPORT — *Compile-time constant*

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.

Note: The DAT module is supported by all devices that have an EDMA or DMA peripheral.

Example

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

DAT_wait

Waits for previous transfer to complete identification by transfer ID

Function

```
void DAT_wait(
    Uint32 id
);
```

Arguments

id — Transfer identifier, returned by one of the DAT copy or DAT fill routines. Two predefined transfer IDs may be used:

- DAT_XFRID_WAITALL
- DAT_XFRID_WAITNONE

Using `DAT_XFRID_WAITALL` means wait until all transfers have completed. Using `DAT_XFRID_WAITNONE` means do not wait for any transfers to complete. This can be useful as the first operation in a pipelined copy sequence.

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 not 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);
```

Chapter 6

DMA Module

This chapter describes the DMA module, lists the API functions and macros within the module, discusses how to use a DMA channel, and provides a DMA API reference section.

Topic **Page**

6.1 Overview

Currently, the are two DMA architectures used on TMS320C6x™ devices are: DMA and EDMA (enhanced DMA). Devices such as the C6201™ have the DMA peripheral, whereas the C6211™ has the EDMA peripheral. The two architectures are different enough to warrant a separate API module for each.

Table 6–1 lists the configuration structures for use with the DMA functions. Table 6–2 lists the functions and constants available in the CSL DMA module.

Table 6–1. DMA Configuration Structures

Structure	Purpose	See page ...
DMA_Config	DMA structure that contains all local registers required to set up a specific DMA channel	6-7
DMA_GlobalConfig	Global DMA structure that contains all global registers that you may need to initialize a DMA channel	6-8

Table 6–2. DMA APIs

(a) *DMA Primary Functions*

Syntax	Type	Description	See page ...
DMA_close	F	Closes a DMA channel opened via `DMA_open()`	6-10
DMA_config	F	Sets up the DMA channel using the configuration structure	6-10
DMA_configArgs	F	Sets up the DMA channel using the register values passed in	6-11
DMA_open	F	Opens a DMA channel for use	6-12
DMA_pause	F	Pauses the DMA channel by setting the START bits in the primary control register appropriately	6-12
DMA_reset	F	Resets the DMA channel by setting its registers to power-on defaults	6-13
DMA_start	F	Starts a DMA channel running without auto–initialization	6-13
DMA_stop	F	Stops a DMA channel by setting the START bits in the primary control register appropriately	6-14

Note: F = Function; C = Constant; M = Macro

Table 6–2. DMA APIs (Continued)

(b) *DMA Global Register Functions*

Syntax	Type	Description	See page ...
DMA_allocGlobalReg	F	Provides resource management for the DMA global registers	6-14
DMA_freeGlobalReg	F	Frees a global DMA register previously allocated by calling `DMA_AllocGlobalReg()`	6-16
DMA_getGlobalReg	F	Reads a global DMA register that was previously allocated by calling `DMA_AllocGlobalReg()`	6-16
DMA_getGlobalRegAddr	F	Gets DMA global register address	6-17
DMA_globalAlloc	F	Allocates DMA global registers	6-18
DMA_globalConfig	F	Configures entry for DMA configuration structure	6-18
DMA_globalConfigArgs	F	Configures entry for DMA registers	6-19
DMA_globalFree	F	Frees Allocated DMA global register	6-20
DMA_globalGetConfig	F	Returns the entry for the DMA configuration structure	6-21
DMA_setGlobalReg	F	Sets value of a global DMA register previously allocated by calling `DMA_AllocGlobalReg()`	6-21

(c) *DMA Auxiliary Functions, Constants, and Macros*

Syntax	Type	Description	See page ...
DMA_autoStart	F	Starts a DMA channel with auto–initialization	6-22
DMA_CHA_CNT	C	Number of DMA channels for the current device	6-22
DMA_CLEAR_CONDITION	M	Clears condition flag	6-22
DMA_GBLADDRA	C	DMA global address register A mask	6-23
DMA_GBLADDRB	C	DMA global address register B mask	6-23
DMA_GBLADDRC	C	DMA global address register C mask	6-23
DMA_GBLADDRD	C	DMA global address register D mask	6-23
DMA_GBLCNTA	C	DMA global count reload register A mask	6-24
DMA_GBLCNTB	C	DMA global count reload register B mask	6-24
DMA_GBLIDXA	C	DMA global index register A mask	6-24

Note: F = Function; C = Constant; M = Macro

Table 6–2. DMA APIs (Continued)

DMA_GBLIDXB	C	DMA global index register B mask	6-24
DMA_GET_CONDITION	M	Gets condition flag	6-25
DMA_getConfig	F	Reads the current DMA configuration structure	6-25
DMA_getEventId	F	Returns the IRQ event ID for the DMA completion interrupt	6-26
DMA_getStatus	F	Reads the status bits of the DMA channel	6-26
DMA_restoreStatus	F	Restores the status from DMA_getStatus() by setting the START bit of the PRICTL primary control register	6-26
DMA_setAuxCtl	F	Sets the DMA AUXCTL register	6-27
DMA_SUPPORT	C	A compile time constant whose value is 1 if the device supports the DMA module	6-27
DMA_wait	F	Enters a spin loop that polls the DMA status bits until the DMA completes	6-28

Note: F = Function; C = Constant; M = Macro

6.1.1 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_config()` or by passing register values to the `DMA_configArgs()` function. To assist in creating register values, there are *DMA_RMK* (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 managing shared global DMA registers, `DMA_allocGlobalReg()`, `DMA_freeGlobalReg()`, `DMA_setGlobalReg()`, and `DMA_getGlobalReg()`.

6.2 Macros

There are two types of DMA macros: those that access registers and fields, and those that construct register and field values.

Table 6–3 lists the DMA macros that access registers and fields, and Table 6–4 lists the DMA macros that construct register and field values. The macros themselves are found in Chapter 24, *Using the HAL Macros*.

The DMA module includes handle-based macros.

Table 6–3. DMA Macros that Access Registers and Fields

Macro	Description/Purpose	See page ...
DMA_ADDR(<REG>)	Register address	24-12
DMA_RGET(<REG>)	Returns the value in the peripheral register	24-18
DMA_RSET(<REG>,x)	Register set	24-20
DMA_FGET(<REG>,<FIELD>)	Returns the value of the specified field in the peripheral register	24-13
DMA_FSET(<REG>,<FIELD>,fieldval)	Writes *fieldval* to the specified field in the peripheral register	24-15
DMA_FSETS(<REG>,<FIELD>,<SYM>)	Writes the symbol value to the specified field in the peripheral	24-17
DMA_RGETA(addr,<REG>)	Gets register for a given address	24-19
DMA_RSETA(addr,<REG>,x)	Sets register for a given address	24-20
DMA_FGETA(addr,<REG>,<FIELD>)	Gets field for a given address	24-13
DMA_FSETA(addr,<REG>,<FIELD>,fieldval)	Sets field for a given address	24-16
DMA_FSETSA(addr,<REG>,<FIELD>,<SYM>)	Sets field symbolically for a given address	24-17
DMA_ADDRH(h,<REG>)	Returns the address of a memory-mapped register for a given handle	24-12
DMA_RGETH(h,<REG>)	Returns the value of a register for a given handle	24-19
DMA_RSETH(h,<REG>,x)	Sets the register value to x for a given handle	24-21
DMA_FGETH(h,<REG>,<FIELD>)	Returns the value of the field for a given handle	24-14
DMA_FSETH(h,<REG>,<FIELD>,fieldval)	Sets the field value to x for a given handle	24-16

Table 6–4. DMA Macros that Construct Register and Field Values

Macro	Description/Purpose	See page ...
DMA_<REG>_DEFAULT	Register default value	24-21
DMA_<REG>_RMK()	Register make	24-23
DMA_<REG>_OF()	Register value of ...	24-22
DMA_<REG>_<FIELD>_DEFAULT	Field default value	24-24
DMA_FMK()	Field make	24-14
DMA_FMKS()	Field make symbolically	24-15
DMA_<REG>_<FIELD>_OF()	Field value of ...	24-24
DMA_<REG>_<FIELD>_<SYM>	Field symbolic value	24-24

6.3 Configuration Structures

Because the DMA has both local and global registers for each channel, the CSL DMA module has two configuration structures:

- **DMA_Config** (channel configuration structure) contains all the local registers required to set up a specific DMA channel.
- **DMA_GlobalConfig** (global configuration structure) contains all the global registers needed to initialize a DMA channel. These global registers are resources shared across the different DMA channels, and include element/frame indexes and reload registers, as well as src/dst page registers.

You can use literal values or the _RMK macros to create the structure member values.

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

Example

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

DMA_GlobalConfig *DMA global register configuration structure*

Structure

```
typedef struct {
    Uint32 addrA;
    Uint32 addrB;
    Uint32 addrC;
    Uint32 addrD;
    Uint32 idxA;
    Uint32 idxB;
    Uint32 cntA;
    Uint32 cntB;
} DMA_GlobalConfig;
```

Members

addrA	Global address register A value.
addrB	Global address register B value.
addrC	Global address register C value.
addrD	Global address register D value.
idxA	Global index register A value.
idxB	Global index register B value.
cntA	Global count reload register A value.
cntB	Global count reload register B value.

Description

This is the DMA global register configuration structure used to set up a DMA global register configuration. You create and initialize this structure, then pass its address to the DMA_globalConfig() function.

Example

```
Uint32 dmaGblRegMsk;
  Uint32 dmaGblRegId = DMA_GBLADDRB | DMA_GBLADDRC;
  DMA_GlobalConfig dmaGblCfg = {
    0x00000000, /*  Global Address Register A  */
    0x80001000, /*  Global Address Register B  */
    0x80002000, /*  Global Address Register C  */
    0x00000000, /*  Global Address Register D  */
    0x00000000, /*  Global Index Register A   */
    0x00000000, /*  Global Index Register B  */
    0x00000000, /*  Global Count Reload Register A  */
    0x00000000  /*  Global Count Reload Register B  */
  };
  dmaGblRegMsk = DMA_globalAlloc(dmaGblRegId);
  DMA_globalConfig(dmaGblRegMsk, &dmaGblCfg);
```

6.4 Functions

6.4.1 Primary Functions

DMA_close *Closes 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);
```

DMA_config *Sets up DMA channel using configuration structure*

Function

```
void DMA_config(
    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_configArgs()` and `DMA_Config`.

Example

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

DMA_configArgs *Sets up DMA channel using register values passed in*

Function

```
void DMA_configArgs(
    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_config()`.

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

Example

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

DMA_open

Opens 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 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);
```

DMA_pause

Pauses DMA channel by setting START bits in primary control register

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);
```

DMA_reset

Resets 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);
```

DMA_start

Starts DMA channel running without auto–initialization

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 auto–initialization by setting the START bits in the primary control register accordingly. See also `DMA_pause()`, `DMA_stop()`, and `DMA_autoStart()`.

Example

```
DMA_start(hDma);
```

DMA_stop *Stops DMA channel by setting START bits in primary control register*

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);
```

6.4.2 DMA Global Register Functions

DMA_allocGlobalReg *Allocates 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 reallocated 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 directly be used with the `DMA_PRICTL_RMK` 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);
```

DMA_freeGlobalReg *Frees global DMA register that was previously allocated*

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 reallocation.

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);
```

DMA_getGlobalReg *Reads global DMA register that was previously allocated*

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);
```

DMA_getGlobalRegAddr

Gets DMA global register address

Function

```
Uint32 DMA_getGlobalRegAddr(
    Uint32 regId
);
```

Arguments

regId		DMA global registers ID

Return Value

Uint32	DMA global register address corresponding to regId

Description

Get DMA global register address and return the address value.

Example

```
Uint32 regId = DMA_GBL_ADDRRLDB;
  Uint32 regAddr;
  regAddr = DMA_getGlobalRegAddr(regId);
```

DMA_globalAlloc *Allocates DMA global registers*

Function

```
Uint32 DMA_globalAlloc(
    Uint32 regs
);
```

Arguments

regs	DMA global registers ID

Return Value

Uint32	Allocated DMA global registers mask

Description

Allocates DMA global registers and returns a mask of allocated DMA global registers. Mask depends on DMA global register ID and the availability of the register.

Example

```
Uint32 dmaGblRegMsk;
  Uint32 regs = DMA_GBLADDRB | DMA_GBLADDRC;
  DmaGblRegMsk = DMA_globalAlloc(regs);
```

DMA_globalConfig *Sets up the DMA global registers using the configuration structure*

Function

```
void DMA_globalConfig(
    Uint32 regs,
    DMA_GlobalConfig *cfg
);
```

Arguments

regs	DMA global register mask
cfg	Pointer to an initialized configuration structure.

Return Value

none

Description

Sets up the DMA global registers using the configuration structure. The values of the structure that are written to the DMA global registers depend on the DMA global register mask.

Example

```
Uint32 dmaGblRegMsk;
  Uint32 dmaGblRegId = DMA_GBLADDRB | DMA_GBLADDRC;
  DMA_GlobalConfig dmaGblCfg = {
    0x00000000, /*  Global Address Register A  */
    0x80001000, /*  Global Address Register B  */
    0x80002000, /*  Global Address Register C  */
    0x00000000, /*  Global Address Register D  */
    0x00000000, /*  Global Index Register A    */
    0x00000000, /*  Global Index Register B    */
    0x00000000, /*  Global Count Reload Register A  */
    0x00000000  /*  Global Count Reload Register B  */
  };
  dmaGblRegMsk = DMA_globalAlloc(dmaGblRegId);
  DMA_globalConfig(dmaGblRegMsk, &dmaGblCfg);
```

DMA_globalConfigArgs — *Establishes DMA global register value*

Function

```
void DMA_globalConfigArgs(
    Uint32 regs,
    Uint32 addrA,
    Uint32 addrB,
    Uint32 addrC,
    Uint32 addrD,
    Uint32 idxA,
    Uint32 idxB,
    Uint32 cntA,
    Uint32 cntB
);
```

Arguments

regs	DMA global register mask value.
addrA	Global address register A value.
addrB	Global address register B value.
addrC	Global address register C value.
addrD	Global address register D value.
idxA	Global index register A value.
idxB	Global index register B value.

	cntA	Global count reload register A value.
	cntB	Global count reload register B value.

Return Value none

Description Sets up the DMA global registers using the register values passed in. The register values that are written to the DMA global registers depend on the DMA global register mask.

Example

```
Uint32 dmaGblRegMsk;
  Uint32 dmaGblRegId = DMA_GBLADDRB | DMA_GBLADDRC;
  Uint32 addrA = 0x00000000;
  Uint32 addrB = 0x80001000;
  Uint32 addrC = 0x80002000;
  Uint32 addrD = 0x00000000;
  Uint32 idxA = 0x00000000;
  Uint32 idxB = 0x00000000;
  Uint32 cntA = 0x00000000;
  Uint32 cntB = 0x00000000;
  dmaGblRegMsk = DMA_globalAlloc(dmaGblRegId);
  DMA_globalConfigArgs(
    dmaGblRegMsk,
    addrA,
    addrB,
    addrC,
    addrD,
    idxA,
    idxB,
    cntA,
    cntB
  );
```

DMA_globalFree *Frees allocated DMA global registers*

Function

```
Void DMA_globalFree(
    Uint32 regs
);
```

Arguments	regs	DMA global registers ID

Return Value none

Description Frees previously allocated DMA global registers; depends on regs.

Example
```
Uint32 dmaGblRegId = DMA_GBLADDRB | DMA_GBLADDRC;
  DMA_globalFree(dmaGblRegId);
```

DMA_globalGetConfig *Gets current DMA global register configuration value*

Function
```
void DMA_globalGetConfig(
    Uint32 regs,
    DMA_GlobalConfig *cfg
);
```

Arguments

regs DMA global register ID

cfg Pointer to an initialized configuration structure.

Return Value none

Description Gets DMA global registers current configuration value depending on DMA global register ID.

Example
```
Uint32 dmaGblRegId = DMA_GBLADDRB | DMA_GBLADDRC;
DMA_GlobalConfig dmaGblCfg;
DMA_globalGetConfig(dmaGblRegId, &dmaGblCfg);
```

DMA_setGlobalReg *Sets value of global DMA register that was previously allocated*

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_INDEX,0x00200040);
…
 DMA_setGlobalReg(RegId,0x12345678);
```

6.4.3 DMA Auxiliary Functions, Constants, and Macros

DMA_autoStart *Starts 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);
```

DMA_CHA_CNT *Number of DMA channels for current device*

Constant

DMA_CHA_CNT

Description

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

DMA_CLEAR_CONDITION *Clears one of the condition flags in 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

Return Value none

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

Example

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

DMA_GBLADDRA *DMA global address register A mask*

Constant DMA_GBLADDRA

Description This constant allows selection of the global address register A. See `DMA_globalAlloc()`, `DMA_globalConfig()`, and `DMA_globalConfigArgs()` functions.

DMA_GBLADDRB *DMA global address register B mask*

Constant DMA_GBLADDRB

Description This constant allows selection of the global address register B. See `DMA_globalAlloc()`, `DMA_globalConfig()`, and `DMA_globalConfigArgs()` functions.

DMA_GBLADDRC *DMA global address register C mask*

Constant DMA_GBLADDRC

Description This constant allows selection of the global address register C. See `DMA_globalAlloc()`, `DMA_globalConfig()`, and `DMA_globalConfigArgs()` functions.

DMA_GBLADDRD *DMA global address register D mask*

Constant DMA_GBLADDRD

Description This constant allows selection of the global address register D. See `DMA_globalAlloc()`, `DMA_globalConfig()`, and `DMS_globalConfigArgs()` functions.

DMA_GBLCNTA *DMA global count reload register A mask*

Constant DMA_GBLCNTA

Description This constant allows selection of the global count reload register A. See `DMA_globalAlloc()`, `DMA_globalConfig()`, and `DMA_globalConfigArgs()` functions.

DMA_GBLCNTB *DMA global count reload register B mask*

Constant DMA_GBLCNTB

Description This constant allows selection of the global count reload register B. See `DMA_globalAlloc()`, `DMA_globalConfig()`, and `DMA_globalConfigArgs()` functions.

DMA_GBLIDXA *DMA global index register A mask*

Constant DMA_GBLIDXA

Description This constant allows selection of the global index register A. See `DMA_globalAlloc()`, `DMA_globalConfigArgs()`, and `DMA_globalConfig()` functions.

DMA_GBLIDXB *DMA global index register B mask*

Constant DMA_GBLIDXB

Description This constant allows selection of the global index register B. See `DMA_globalAlloc()`, `DMA_globalConfig()`, and `DMA_globalConfigArgs()` functions.

DMA_GET_CONDITION — *Gets one of the condition flags in 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

Return Value

Condition — 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* (SPRU190) for a description of the condition flags.

Example

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

DMA_getConfig — *Reads the current DMA configuration values*

Function

```
void DMA_getConfig(
    DMA_Handle hDma,
    DMA_Config *config
);
```

Arguments

hDma — DMA handle. See `DMA_open()`

config — Pointer to a configuration structure

Return Value

none

Description

Get DMA current configuration value

Example

```
DMA_config dmaCfg;
DMA_getConfig(hDma, &dmaCfg);
```

DMA_getEventId — *Returns IRQ event ID for 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);
```

DMA_getStatus — *Reads status bits of 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_getStatus(hDma)==DMA_STATUS_RUNNING);
```

DMA_restoreStatus — *Restores the status from DMA_getStatus()*

Function

```
void DMA_restoreStatus(
    Uint32 hDma,
    Uint32 status
);
```

Arguments

hDma	Handle to DMA channel. See `DMA_open()`
status	Status from `DMA_getStatus()` function

Return Value none

Description Restores the status from `DMA_getStatus()` by setting the START bit of the PRICTL primary control register.

Example

```
status = DMA_getStatus(hDma);
...
DMA_restoreStatus(hDma, status);
```

DMA_setAuxCtl *Sets 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_AUXCTL_RMK` macro to construct the register value based on field values. The default value for this register is `DMA_AUXCTL_DEFAULT`.

Example

```
DMA_setAuxCtl(0x00000000);
```

DMA_SUPPORT *Compile time constant*

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
```

DMA_wait

Enters spin loop that polls DMA status bits until 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_getStatus(hDma)&DMA_STATUS_RUNNING);
```

Example

```
DMA_wait(hDma);
```

Chapter 7

EDMA Module

This chapter describes the EDMA module, lists the API functions and macros within the module, discusses how to use an EDMA channel, and provides an EDMA API reference section.

Topic **Page**

7.1 Overview

Currently, there are two DMA architectures used on C6x devices: DMA and EDMA (Enhanced DMA). Devices such as the C6201™ have the DMA peripheral whereas C6211™ devices have the EDMA peripheral. The two architectures are different enough to warrant a separate API module for each.

Table 7–1 lists the configuration structures for use with the EDMA functions. Table 7–2 lists the functions and constants available in the CSL EDMA module.

Table 7–1. EDMA Configuration Structure

Structure	Purpose	See page ...
EDMA_Config	The EDMA configuration structure used to set up an EDMA channel	7-7

Table 7–2. EDMA APIs

(a) EDMA Primary Functions

Syntax	Type	Description	See page ...
EDMA_close	F	Closes a previously opened EDMA channel	7-8
EDMA_config	F	Sets up the EDMA channel using the configuration structure	7-8
EDMA_configArgs	F	Sets up the EDMA channel using the EDMA parameter arguments	7-9
EDMA_open	F	Opens an EDMA channel	7-10
EDMA_reset	F	Resets the given EDMA channel	7-12

(b) EDMA Auxiliary Functions and Constants

Syntax	Type	Description	See page ...
EDMA_allocTable	F	Allocates a parameter RAM table from PRAM	7-12
EDMA_allocTableEx	F	Allocates set of parameter RAM tables from PRAM	7-13
EDMA_CHA_CNT	C	Number of EDMA channels	7-14
EDMA_chain	F	Sets the TCC,TCINT fields of the parent EDMA handle	7-14
EDMA_clearChannel	F	Clears the EDMA event flag in the EDMA channel event register	7-15
EDMA_clearPram	F	Clears the EDMA parameter RAM (PRAM)	7-16
EDMA_disableChaining	F	Disables EDMA chaining	7-16

Note: F = Function; C = Constant

Table 7–2. EDMA APIs (Continued)

Syntax	Type	Description	See page ...
EDMA_enableChaining	F	Enables EDMA chaining	7-16
EDMA_disableChannel	F	Disables an EDMA channel	7-17
EDMA_enableChannel	F	Enables an EDMA channel	7-17
EDMA_freeTable	F	Frees up a PRAM table previously allocated	7-18
EDMA_freeTableEx	F	Frees a previously allocated set of parameter RAM tables	7-18
EDMA_getChannel	F	Returns the current state of the channel event	7-19
EDMA_getConfig	F	Reads the current EDMA configuration values	7-19
EDMA_getPriQStatus	F	Returns the value of the priority queue status register (PQSR)	7-20
EDMA_getScratchAddr	F	Returns the starting address of the EDMA PRAM used as non-cacheable on-chip SRAM (scratch area)	7-20
EDMA_getScratchSize	F	Returns the size (in bytes) of the EDMA PRAM used as non-cacheable on-chip SRAM (scratch area)	7-20
EDMA_getTableAddress	F	Returns the 32-bit absolute address of the table	7-21
EDMA_intAlloc	F	Allocates a transfer complete code	7-21
EDMA_intClear	F	Clears EDMA transfer completion interrupt pending flag	7-21
EDMA_intDefaultHandler	F	Default function called by `EDMA_intDispatcher()`	7-22
EDMA_intDisable	F	Disables EDMA transfer completion interrupt	7-22
EDMA_intDispatcher	F	`Calls an ISR when CIER[x] and CIPR[x] are both set`	7-22
EDMA_intEnable	F	Enables EDMA transfer completion interrupt	7-23
EDMA_intFree	F	Frees a transfer complete code previously allocated	7-23
EDMA_intHook	F	Hooks to an ISR channel which is called by `EDMA_int-Dispatcher()`	7-24
EDMA_intTest	F	Tests EDMA transfer completion interrupt pending flag	7-24
EDMA_link	F	Links two EDMA transfers together	7-25
EDMA_qdmaConfig	F	Sets up QDMA registers using configuration structure	7-25
EDMA_qdmaConfigArgs	F	Sets up QDMA registers using arguments	7-26
EDMA_resetAll	F	Resets all EDMA channels supported by the chip device	7-27
EDMA_resetPriQLength	F	Resets the Priority queue length to the default value	7-27

Note: F = Function; C = Constant

Table 7–2. EDMA APIs (Continued)

Syntax	Type	Description	See page ...
EDMA_setChannel	F	Triggers an EDMA channel by writing to the appropriate bit in the event set register (ESR)	7-28
EDMA_setEvtPolarity	F	Sets the polarity of the event associated with the EDMA handle.	7-28
EDMA_setPriQLength	F	Sets the length of a given priority queue allocation register	7-29
EDMA_SUPPORT	C	A compile-time constant whose value is 1 if the device supports the EDMA module	7-29
EDMA_TABLE_CNT	C	A compile-time constant that holds the total number of parameter table entries in the EDMA PRAM	7-29

Note: F = Function; C = Constant

7.1.1 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_config()` or by passing register values to the `EDMA_configArgs()` function. To assist in creating register values, the `_RMK` (make) macros construct register values based on field values. In addition, the symbol constants may be used for the field values.

Two functions manage the parameter RAM (PRAM) tables: `EDMA_allocTable()` and `EDMA_freeTable()`.

7.2 Macros

There are two types of EDMA macros: those that access registers and fields, and those that construct register and field values.

Table 7–3 lists the EDMA macros that access registers and fields, and Table 7–4 lists the EDMA macros that construct register and field values. The macros themselves are found in Chapter 24, *Using the HAL Macros*.

The EDMA module includes handle-based macros.

Table 7–3. EDMA Macros That Access Registers and Fields

Macro	Description/Purpose	See page...
EDMA_ADDR(<REG>)	Register address	24-12
EDMA_RGET(<REG>)	Returns the current value of a register	24-18
EDMA_RSET(<REG>,x)	Register set	24-20
EDMA_FGET(<REG>,<FIELD>)	Returns the value of the specified field in a register	24-13
EDMA_FSET(<REG>,<FIELD>,x)	Writes *fieldval* to the specified field in a register	24-15
EDMA_FSETS(<REG>,<FIELD>,<SYM>)	Writes the symbol value to the specified field in the peripheral	24-17
EDMA_RGETA(addr,<REG>)	Gets register value for a given address	24-19
EDMA_RSETA(addr,<REG>,x)	Sets register for a given address	24-20
EDMA_FGETA(addr,<REG>,<FIELD>)	Gets field for a given address	24-13
EDMA_FSETA(addr,<REG>,<FIELD>,x)	Sets field for a given address	24-16
EDMA_FSETSA(addr,<REG>,<FIELD>,<SYM>)	Sets field symbolically for a given address	24-17
EDMA_ADDRH(h,<REG>)	Returns the address of a memory-mapped register for a given handle	24-12
EDMA_RGETH(h,<REG>)	Returns the value of a register for a given handle	24-19
EDMA_RSETH(h,<REG>,x)	Sets the register value to x for a given handle	24-21
EDMA_FGETH(h,<REG>,<FIELD>)	Returns the value of the field for a given handle	24-14
EDMA_FSETH(h,<REG>,<FIELD>,x)	Sets the field value to x for a given handle	24-16
EDMA_FSETSH(h,<REG>,<FIELD>,<SYM>)	Sets the field symbolically for a given handle	24-18

Table 7–4. EDMA Macros that Construct Register and Field Values

Macro	Description/Purpose	See page...
EDMA_<REG>_DEFAULT	Register default value	24-21
EDMA_<REG>_RMK()	Register make	24-23
EDMA_<REG>_OF()	Register value of ...	24-22
EDMA_<REG>_<FIELD>_DEFAULT	Field default value	24-24
EDMA_FMK()	Field make	24-14
EDMA_FMKS()	Field make symbolically	24-15
EDMA_<REG>_<FIELD>_OF()	Field value of ...	24-24
EDMA_<REG>_<FIELD>_<SYM>	Field symbolic value	24-24

7.3 Configuration Structure

EDMA_Config *EDMA configuration structure used to set up 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_config()` function.

Example

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

7.4 Functions

7.4.1 EDMA Primary Functions

EDMA_close *Closes 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 device handle:

From `EDMA_open()`

Example

```
EDMA_close(hEdma);
```

EDMA_config *Sets up EDMA channel using configuration structure*

Function

```
void EDMA_config(
    EDMA_Handle hEdma,
    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_configArgs()` and `EDMA_Config`.

This function accepts the following device handles:

- From `EDMA_open()`
- From `EDMA_allocTable()`

Example

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

EDMA_configArgs *Sets up EDMA channel using EDMA parameter arguments*

Function

```
void EDMA_configArgs(
    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_config()`.

This function accepts the following device handles:

- From `EDMA_open()`
- From `EDMA_allocTable()`

Example

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

EDMA_open *Opens 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

The following are for C64x only:

- EDMA_CHA_XEVT2
- EDMA_CHA_REVT2
- EDMA_CHA_TINT2

- ❏ EDMA_CHA_SDINTA
- ❏ EDMA_CHA_SDINTB
- ❏ EDMA_CHA_PCI
- ❏ EDMA_CHA_GPINT0
- ❏ EDMA_CHA_GPINT1
- ❏ EDMA_CHA_GPINT2
- ❏ EDMA_CHA_GPINT3
- ❏ EDMA_CHA_GPINT4
- ❏ EDMA_CHA_GPINT5
- ❏ EDMA_CHA_GPINT6
- ❏ EDMA_CHA_GPINT7
- ❏ EDMA_CHA_GPINT8
- ❏ EDMA_CHA_GPINT9
- ❏ EDMA_CHA_GPINT10
- ❏ EDMA_CHA_GPINT11
- ❏ EDMA_CHA_GPINT12
- ❏ EDMA_CHA_GPINT13
- ❏ EDMA_CHA_GPINT14
- ❏ EDMA_CHA_GPINT15
- ❏ EDMA_CHA_UREVT
- ❏ EDMA_CHA_UXEVT
- ❏ EDMA_CHA_VCPREVT
- ❏ EDMA_CHA_VCPXEVT
- ❏ EDMA_CHA_TCPREVT
- ❏ EDMA_CHA_TCPXEVT

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, `INV` is returned.

If the `EDMA_OPEN_RESET` is specified, the EDMA channel is reset and 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, INV is returned.

Note: If the DAT module is open [see DAT_open()], then EDMA transfer completion interrupts 1 through 4 are reserved.

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

Example

```
EDMA_Handle hEdma;
...
hEdma = EDMA_open(EDMA_CHA_TINT0,EDMA_OPEN_RESET);
...
```

EDMA_reset — *Resets 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.

The following steps are taken:

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

This function accepts the following device handle:

From EDMA_open()

Example

```
EDMA_reset(hEdma);
```

7.4.2 EDMA Auxiliary Functions and Constants

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 EDMA_TABLE_CNT–1; –1 for any.

Return Value Device Handle Returns a device handle

Description This function allocates the PRAM tables dedicated to the Reload/Link parameters. You use the Reload/Link 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 `INV` is returned.

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

For TMS320C621x/C671x, the first two tables located at 0x01A00180 and 0x01A00198, respectively, are reserved. The first parameter table is initialized to zero, and the second table is reserved for CSL code. The first available table for the user starts at address 0x01A001B0. There are 67 available tables, with table numbers from 0 to 66.

For TMS320C64xx, the first two tables located at 0x01A00600 and 0x01A00618 are reserved. The first parameter table is initialized to zero, and the second table is reserved for CSL code. The first available table for the user starts at address 0x01A00630. There are 19 available tables, with table numbers from 0 to 18.

```
hEdmaTable=EDMA_allocTable(0);
```

will allocate the Reload/Link parameter table located at:

- 0x01A001B0 for C621x/C671x devices
- 0x01A00630 for C64xx devices

Example

```
EDMA_Handle hEdmaTable;
...
hEdmaTable = EDMA_allocTable(–1);
```

EDMA_allocTableEx *Allocates set of parameter RAM tables from PRAM*

Function

```
int EDMA_allocTableEx(
    int cnt,
    EDMA_Handle *array
);
```

Arguments

cnt	Number of tables to allocate
array	An array to hold the table handles for each table allocated

Return Value

numAllocated Returns the actual number of tables allocated. It will either be cnt or 0.

Description

This function allocates a set of parameter RAM tables from PRAM. The tables are not guaranteed to be contiguous in memory. You use PRAM tables for linking transfers together. The array passed in is filled with table handles and each one may be used for APIs that require a device handle.

If you finish with the tables and wish to free them up again, call `EDMA_freeTableEx()`.

Example

```
EDMA_Handle hEdmaTableArray[16];
...
if (EDMA_allocTableEx(16,hEdmaTableArray)) {
  ...
}
```

EDMA_CHA_CNT *Number of EDMA channels*

Constant

EDMA_CHA_CNT

Description

Compile time constant that holds the number of EDMA channels.

EDMA_chain *Sets the TCC, TCINT fields of the parent EDMA handle*

Function

```
void EDMA_chain(
    EDMA_Handle parent,
    EDMA_Handle nextChannel,
    int flag_tcc,
    int flag_atcc
);
```

Arguments

parent — EDMA handle following the chainable transfer.

nextChannel — EDMA handle associated with the channel to be chained.

flag_tcc — Flag for TCC,TCINT setting (and TCCM for C64x devices). The following constants must be used:

- 0
- EDMA_TCC_SET or 1

flag_atcc — Flag for ATCC, ATCINT setting (C64x devices only). The following constants must be used:

- 0
- EDMA_ATCC_SET or 1

Return Value none

Description Sets the TCC,TCINT fields (and TCCM field for C64x devices) of the “parent” EDMA handle based on the “nextChannel” EDMA handle.

For C621x/C671x, only channels from 8 to 11 are chainable.

Example

```
EDMA_Handle hEdmaChain,hEdmaPar;
Unit32 Tcc;

/*Open and Configure parent Channel*/
hEdmaPar=EDMA_open(EDMA_CHA_TINT1,EDMA_OPEN_RESET);
EDMA_config(hEdmaPar,&myConfig);

/*Allocate a transfer complete code*/
Tcc=intAlloc(–1);

/*Open the Channel for the next transfer with TCC value*/
hEdmaChain=EDMA_open(Tcc,EDMA_OPEN_RESET);

/*Update the TCC, TCINT, (TCCM) fields of the parent channel
configuration*/
EDMA_chain(hEdmaPar,hEdmaChain,EDMA_TCC_SET,0)
/*Enable chaining: CCER (CCERL/CCERH) setting*/
);
EDMA_enableChaining(hEdmaChain);
```

EDMA_clearChannel *Clears EDMA event flag in 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 device handle:

From `EDMA_open()`

Example

```
EDMA_clearChannel(hEdma);
```

EDMA_clearPram *Clears the EDMA parameter RAM (PRAM)*

Function

```
void EDMA_clearPram(
    Uint32 val
);
```

Arguments val Value to clear the PRAM with

Return Value none

Description This function clears all of the EDMA parameter RAM with the value specified. This function should not be called if EDMA channels are active.

Example

```
EDMA_clearPram(0);
```

EDMA_disableChaining *Disables EDMA chaining*

Function

```
void EDMA_disableChaining(
    EDMA_Handle hEdma
);
```

Arguments hEdma EDMA handle to be chained

Return Value none

Description Disables the CCE bit in the Channel Chaining Enable Register associated with the EDMA handle. See also `EDMA_enableChaining()`.

For C621x/C671x, only channels from 8 to 11 are chainable.

Example

```
EDMA_enableChaining(hEdmaCha8);
EDMA_disableChaining(hEdmaCha8);
```

EDMA_enableChaining *Enables EDMA chaining*

Function

```
void EDMA_enableChaining(
    EDMA_Handle hEdma
);
```

Arguments hEdma EDMA handle to be chained

Return Value none

Description Enables the CCE bit in the Channel Chaining Enable Register associated with the EDMA handle.

For C621x/C671x, only channels from 8 to 11 are chainable.

Example

```
EDMA _Handle hEdmaCha8
Uint32 Tcc;

/*Allocate the transfer complete code*/
Tcc=EDMA_intAlloc(8);
/*Open the channel related to the TCC*/
hEdmaCha8=EDMA_open(Tcc,EDMA_OPEN_RESET);
/*Enable chaining*/
EDMA_enableChaining(hEdmaCha8);
```

EDMA_disableChannel *Disables 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 device handle:

From `EDMA_open()`

Example

```
EDMA_disableChannel(hEdma);
```

EDMA_enableChannel *Enables 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 enable it explicitly.

This function accepts the following device handle:

From `EDMA_open()`

Example

```
EDMA_enableChannel(hEdma);
```

EDMA_freeTable — *Frees up 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 handle:

From `EDMA_allocTable()`

Example

```
EDMA_freeTable(hEdmaTable);
```

EDMA_freeTableEx — *Frees a previously allocated set of parameter RAM tables*

Function

```
void EDMA_freeTableEx(
    int cnt,
    EDMA_Handle *array
);
```

Arguments

cnt Number of table handles in array to free

array An array containing table handles for each table to be freed

Return Value

none

Description

This function frees a set of parameter RAM tables that were previously allocated. You use PRAM tables for linking transfers together. The array that is passed in must contain the table handles for each one to be freed.

Example

```
EDMA_Handle hEdmaTableArray[16];
...
if (EDMA_allocTableEx(16,hEdmaTableArray)) {
  ...
}
...
EDMA_freeTableEx(16,hEdmaTableArray);
```

EDMA_getChannel — *Returns current state of 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 (ER).

This function accepts the following device handle:

From `EDMA_open()`

Example

```
flag = EDMA_getChannel(hEdma);
```

EDMA_getConfig — *Reads the current EDMA configuration values*

Function

```
void EDMA_getConfig(
    EDMA_Handle hEdma,
    EDMA_Config *config
);
```

Arguments

hEdma	Device handle. See `EDMA_open()`.
config	Pointer to a configuration structure.

Return Value

none

Description

Get EDMA current configuration value

Example

```
EDMA_Config edmaCfg;
EDMA_getConfig(hEdma,&edmaCfg);
```

EDMA_getPriQStatus *Returns value of 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();
```

EDMA_getScratchAddr *Returns starting address of EDMA PRAM 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();
```

EDMA_getScratchSize *Returns size (in bytes) of EDMA PRAM 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();
```

EDMA_getTableAddress *Returns 32-bit absolute address of table*

Function

```
Uint32 EDMA_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 handle:

From `EDMA_allocTable()`

Example

```
addr = EDMA_getTableAddress(hEdmaTable);
```

EDMA_intAlloc *Allocates a transfer complete code*

Function

```
int EDMA_intAlloc(
int tcc
);
```

Arguments

tcc	Transfer-complete code number or –1

Return Value

tccReturn	Transfer-complete code number or –1

Description

This function allocates the transfer-complete code passed in and returns the same TCC number if successful, or –1 otherwise. If –1 is used as an argument, the first available TCC number is allocated.

Example

```
EDMA_intAlloc(5);
EDMA_intAlloc(43);
tcc=EDMA_intAlloc(–1);;
```

EDMA_intClear *Clears EDMA transfer-completion interrupt-pending flag*

Function

```
void EDMA_intClear(
    Uint32 intNum
);
```

Arguments

intNum	Transfer-completion interrupt number [0..31].

Return Value none

Description This function clears a transfer-completion interrupt flag by modifying the CIPR register appropriately.

Note: If the DAT module is open [see `DAT_open()`], then EDMA transfer-completion interrupts 1 through 4 are reserved.

Example

```
EDMA_intClear(12);
```

EDMA_intDefaultHandler

Default function called by EDMA_intDispatcher()

Function

```
void EDMA_intDefaultHandler(
    int  tccNum
);
```

Arguments tccNum Channel completion number

Return Value none

Description This is the default function that is called by `EDMA_intDispatcher()`. It does nothing, it just returns. See also `EDMA_intDispatcher()` and `EDMA_intHook()`.

EDMA_intDisable

Disables EDMA transfer-completion interrupt

Function

```
void EDMA_intDisable(
    Uint32   intNum
);
```

Arguments intNum Transfer-completion interrupt number [0..31].

Return Value none

Description This function disables a transfer-completion interrupt by modifying the CIER register appropriately.

Note: If the DAT module is open [see `DAT_open()`], then EDMA transfer-completion interrupts 1 through 4 are reserved.

Example

```
EDMA_intDisable(12);
```

EDMA_intDispatcher

Calls an ISR when CIER[x] and CIPR[x] are both set

Function

```
void EDMA_intDispatcher(
    void
);
```

Arguments none

Return Value none

Description This function checks for CIER and CIPR for all those bits which are set in both the registers and calls the corresponding ISR. For example, if CIER[14] = 1 and CIPR[14] =1 then it calls the ISR corresponding to channel 14. By default, this ISR is `EDMA_intHandler()`, however, this can be changed by `EDMA_intHook()`. See also `EDMA_intDefaultHandler()` and `EDMA_intHook()`.

Example EDMA_intDispatcher();

EDMA_intEnable *Enables the EDMA transfer-completion interrupt*

Function

```
void EDMA_intEnable(
    Uint32 intNum
);
```

Arguments intNum Transfer-completion interrupt number [0..31].

Return Value none

Description This function enables a transfer completion interrupt by modifying the CIER register appropriately.

Note: If the DAT module is open [see `DAT_open()`], then EDMA transfer-completion interrupts 1 through 4 are reserved.

Example

```
EDMA_intEnable(12);
```

EDMA_intFree *Frees the transfer-complete code number*

Function

```
void EDMA_intFree(
int tcc
);
```

Arguments tcc Transfer-complete code number to be free

Return Value none

Description This function frees a transfer-complete code number previously allocated.

Example

```
EDMA_intAlloc(17);
...
EDMA_intFree(17);
```

EDMA_intHook — *Hooks an isr to a channel, which is called by EDMA_intDsipatcher()*

Function

```
EDMA_IntHandler   EDMA_intHook(
    int                    tccNum
    EDMA_IntHandler     funcAddr
);
```

Arguments

tccNum	Channel to which the ISR is to be hooked
funcAddr	ISR name

Return Value

IntHandler Returns the old ISR address

Description

This function hooks an ISR to the specified channel.

When the tcint is '1' and tccNum is specified in the EDMA options, the EDMA controller sets the corresponding bit in the CIPR register. If the corresponding bit in the CIER register is also set, then calling `EDMA_intDispatcher()` would call the ISR corresponding the the tccNum, which by default is nothing. To change this default ISR to a different one, use `EDMA_intHook()`. Only when an ISR is hooked this way would it be called. See also `EDMA_intDefaultHandler()` and `EDMA_intDispatcher()`.

Example

```
void complete();
EDMA_intHook(13,  complete);  //Hooks  complete  function  to
channel 13
```

EDMA_intTest — *Tests EDMA transfer-completion interrupt-pending flag*

Function

```
Uint32 EDMA_intTest(
    Uint32 intNum
);
```

Arguments

intNum	Transfer-completion interrupt number [0..31].

Return Value

Uint32	Result: 0 = flag was clear 1 = flag was set

Description

This function tests a transfer-completion interrupt flag by reading the CIPR register appropriately.

Note: If the DAT module is open [see `DAT_open()`], then EDMA transfer-completion interrupts 1 through 4 are reserved.

Example

```
if (EDMA_intTest(12)) {
  ...
}
```

EDMA_link — *Links two EDMA transfers together*

Function

```
void EDMA_link(
    EDMA_Handle parent,
    EDMA_Handle child
);
```

Arguments

parent	Handle of the parent (link from parent)
child	Handle of the child (link to child)

Return Value

none

Description

This function links two EDMA transfers together by setting the LINK field of the parent's RLD parameter appropriately. Both parent and child handles may be from `EDMA_open()`, `EDMA_allocTable()`, or a combination of both.

parent–>child

Note: This function does not attempt to set the LINK field of the OPT parameter; this is still up to the user.

Example

```
EDMA_Handle hEdma;
EDMA_Handle hEdmaTable;
...
hEdma = EDMA_open(EDMA_CHA_TINT1,0);
hEdmaTable = EDMA_allocTable(–1);
EDMA_link(hEdma,hEdmaTable);
EDMA_link(hEdmaTable,hEdmaTable);
```

EDMA_qdmaConfig — *Sets up QDMA registers using configuration structure*

Function

```
void EDMA_qdmaConfig(
    EDMA_Config *config
);
```

Arguments

config	Pointer to an initialized configuration structure

Return Value

none

Description Sets up the QDMA registers using the configuration structure. The src, cnt, dst, and idx values are written to the normal QDMA registers, then the opt value is written to the pseudo-OPT register which initiates the transfer. The rld member of the structure is ignored, since the QDMA does not support reloads or linking. See also `EDMA_qdmaConfigArgs()` and `EDMA_Config`.

Example

```
EDMA_Config myConfig = {
  0x41200000, /* opt */
  0x80000000, /* src */
  0x00000040, /* cnt */
  0x80010000, /* dst */
  0x00000004,  /* idx */
  0x00000000  /* rld will be ignored */
};
...
EDMA_qdmaConfig(&myConfig);
```

EDMA_qdmaConfigArgs *Sets up QDMA registers using arguments*

Function

```
void EDMA_qdmaConfigArgs(
    Uint32 opt,
    Uint32 src,
    Uint32 cnt,
    Uint32 dst,
    Uint32 idx
);
```

Arguments

opt	Options
src	Source address
cnt	Transfer count
dst	Destination address
idx	Index

Return Value none

Description Sets up the QDMA registers using the arguments passed in. The src, cnt, dst, and idx values are written to the normal QDMA registers, then the opt value is written to the pseudo-OPT register which initiates the transfer. See also `EDMA_qdmaConfig()` and `EDMA_Config`.

Example

```
EDMA_qdmaConfigArgs(
  0x41200000, /* opt */
  0x80000000, /* src */
  0x00000040, /* cnt */
  0x80010000, /* dst */
  0x00000004  /* idx */
);
```

EDMA_resetAll — *Resets all EDMA channels supported by the chip device*

Function void EDMA_resetAll();

Arguments none

Return Value none

Description This function resets all EDMA channels supported by the device by disabling EDMA enable bits, disabling and clearing the channel interrupt registers, and clearing the PRAM tables associated to the EDMA events.

Example

```
EDMA_resetAll();
```

EDMA_resetPriQLength — *Resets the priority queue length (C64x devices only)*

Function

```
void EDMA_resetPriQLength(
    Uint32 priNum
)
```

Arguments priNum — Queue Number [0–3] associated to the following constants:

- EDMA_Q0
- EDMA_Q1
- EDMA_Q2
- EDMA_Q3

Return Value none

Description Resets the queue length of the associated priority queue allocation register to the default value. See also `EDMA_setPriQLength()` function

Example

```
/* Sets the queue length of the PQAR0 register */
EDMA_setPriQLength(EDMA_Q0,4);
/* Resets the queue length of the PQAR0 */
EDMA_resetPriQLength(EDMA_Q0);
```

EDMA_setChannel *Triggers EDMA channel by writing to appropriate bit in 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 handle:

From `EDMA_open()`

Example

```
EDMA_setChannel(hEdma);
```

EDMA_setEvtPolarity *Sets the event polarity associated with an EDMA channel*

Function

```
void EDMA_setEvtPolarity(
    EDMA_handle hEdma,
    int polarity
);
```

Arguments

hEdma Device handle associated with the EDMA channel obtained by EDMA_open()

polarity Event polarity (0 or 1)

- ❏ EDMA_EVT_LOWHIGH (0)
- ❏ EDMA_EVT_HIGHLOW (1)

Return Value

none

Description

Sets the polarity of the event associated with the EDMA channel.

Example

```
/* Sets the polarity of the event to falling-edge of
transition*/
hEdma=EDMA_open(EDMA_CHA_TINT1,0);
EDMA_setEvtPolarity(hEdma,EDMA_EVT_HIGHLOW);
```

EDMA_setPriQLength *Sets the priority queue length (C64x devices only)*

Function

```
EDMA_setPriQLength(
    Uint32 priNum,
    Uint32 length
);
```

Arguments

priNum — Queue Number [0–3] associated to the following constants:

- EDMA_Q0
- EDMA_Q1
- EDMA_Q2
- EDMA_Q3

length — length of the queue

Return Value

none

Description

Sets the queue length of the associated priority queue allocation register (See `EDMA_resetPriQLength()` function.)

Example

```
/* Sets the queue length of the PQAR1 register to 4 */
EDMA_setPriQLength(EDMA_Q1,4);
EDMA_resetPriQLength(EDMA_Q1);
```

EDMA_SUPPORT *Compile-time constant*

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
```

EDMA_TABLE_CNT *Compile-time constant*

Constant

EDMA_TABLE_CNT

Description

Compile-time constant that holds the total number of reload/link parameter-table entries in the EDMA PRAM.

Chapter 8

EMIF Module

This chapter describes the EMIF module, lists the API functions and macros within the module, and provides an EMIF API reference section.

Note: This module has not been updated for C64x™ devices.

Topic **Page**

8.1 Overview

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_config()` or by passing register values to the `EMIF_configArgs()` 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.

Table 8–1 lists the configuration structure for use with the EMIF functions. Table 8–2 lists the functions and constants available in the CSL EMIF module.

Table 8–1. EMIF Configuration Structure

Structure	Purpose	See page ...
EMIF_Config	Structure used to set up the EMIF peripheral	8-5

Table 8–2. EMIF APIs

Syntax	Type	Description	See page ...
EMIF_config	F	Sets up the EMIF using the configuration structure	8-6
EMIF_configArgs	F	Sets up the EMIF using the register value arguments	8-6
EMIF_getConfig	F	Reads the current EMIF configuration values	8-8
EMIF_SUPPORT	C	A compile time constant that has a value of 1 if the device supports the EMIF module	8-8

Note: F = Function; C = Constant

8.2 Macros

There are two types of EMIF macros: those that access registers and fields, and those that construct register and field values.

Table 8–3 lists the EMIF macros that access registers and fields, and Table 8–4 lists the EMIF macros that construct register and field values. The macros themselves are found in Chapter 24, *Using the HAL Macros*.

EMIF macros are not handle based.

Table 8–3. EMIF Macros that Access Registers and Fields

Macro	Description/Purpose	See page ...
EMIF_ADDR(<REG>)	Register address	24-12
EMIF_RGET(<REG>)	Returns the value in the peripheral register	24-18
EMIF_RSET(<REG>,x)	Register set	24-20
EMIF_FGET(<REG>,<FIELD>)	Returns the value of the specified field in the peripheral register	24-13
EMIF_FSET(<REG>,<FIELD>,fieldval)	Writes *fieldval* to the specified field in the peripheral register	24-15
EMIF_FSETS(<REG>,<FIELD>,<SYM>)	Writes the symbol value to the specified field in the peripheral	24-17
EMIF_RGETA(addr,<REG>)	Gets register for a given address	24-19
EMIF_RSETA(addr,<REG>,x)	Sets register for a given address	24-20
EMIF_FGETA(addr,<REG>,<FIELD>)	Gets field for a given address	24-13
EMIF_FSETA(addr,<REG>,<FIELD>,fieldval)	Sets field for a given address	24-16
EMIF_FSETSA(addr,<REG>,<FIELD>,<SYM>)	Sets field symbolically for a given address	24-17

Table 8–4. EMIF Macros that Construct Register and Field Values

Macro	Description/Purpose	See page ...
EMIF_<REG>_DEFAULT	Register default value	24-21
EMIF_<REG>_RMK()	Register make	24-23
EMIF_<REG>_OF()	Register value of ...	24-22
EMIF_<REG>_<FIELD>_DEFAULT	Field default value	24-24
EMIF_FMK()	Field make	24-14
EMIF_FMKS()	Field make symbolically	24-15
EMIF_<REG>_<FIELD>_OF()	Field value of ...	24-24
EMIF_<REG>_<FIELD>_<SYM>	Field symbolic value	24-24

8.3 Configuration Structure

EMIF_Config *Structure used to set up EMIF peripheral*

Structure EMIF_Config

Members

Uint32 gblctl	EMIF global control register value
Uint32 cectl0	CE0 space control register value
Uint32 cectl1	CE1 space control register value
Uint32 cectl2	CE2 space control register value
Uint32 sdctl3	CE3 space control register value
Uint32 cectl	SDRAM control register value
Uint32 sdtim	SDRAM timing register value
Uint32 sdext	SDRAM extension register value (for 6211/6711 only)

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

Example

```
EMIF_Config MyConfig = { /* example for 6211/6711 */
  0x00003060, /* gblctl */
  0x00000040, /* cectl0 */
  0x404F0323, /* cectl1 */
  0x00000030, /* cectl2 */
  0x00000030, /* cectl3 */
  0x72270000, /* sdctl  */
  0x00000410, /* sdtim  */
  0x00000000  /* sdext  */
};
…
EMIF_config(&MyConfig);
```

8.4 Functions

EMIF_config — *Sets up EMIF using configuration structure*

Function

```
void EMIF_config(
    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_configArgs()` and `EMIF_Config`.

Example

```
EMIF_Config MyConfig = { /* example for 6211/6711 */
  0x00003060, /* gblctl */
  0x00000040, /* cectl0 */
  0x404F0323, /* cectl1 */
  0x00000030, /* cectl2 */
  0x00000030, /* cectl3 */
  0x72270000, /* sdctl  */
  0x00000410, /* sdtim  */
  0x00000000  /* sdext  */
};
…
EMIF_config(&MyConfig);
```

EMIF_configArgs — *Sets up EMIF using register value arguments*

Function

```
/* for 6211/6711 only*/
void EMIF_configArgs(
    Uint32 gblctl,
    Uint32 cectl0,
    Uint32 cectl1,
    Uint32 cectl2,
    Uint32 cectl3,
    Uint32 sdctl,
    Uint32 sdtim,
    Uint32 sdext
```

```
);

/* for all other devices*/
void EMIF_configArgs(
    Uint32 gblctl,
    Uint32 cectl0,
    Uint32 cectl1,
    Uint32 cectl2,
    Uint32 cectl3,
    Uint32 sdctl,
    Uint32 sdtim
 );
```

Arguments

gblctl	EMIF global control register value
cectl0	CE0 space control register value
cectl1	CE1 space control register value
cectl2	CE2 space control register value
cectl3	CE3 space control register value
sdctl	SDRAM control register value
sdtim	SDRAM timing register value
sdext	SDRAM extension register value (optional – reserved for 6211/6711 only)

Return Value none

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

Example

```
EMIF_configArgs( /* devices other than 6211/6711 */
  0x00003060, /* gblctl */
  0x00000040, /* cectl0 */
  0x404F0323, /* cectl1 */
  0x00000030, /* cectl2 */
  0x00000030, /* cectl3 */
  0x72270000, /* sdctl  */
  0x00000410  /* sdtim  */
);
```

EMIF_getConfig — *Reads the current EMIF configuration values*

Function

```
void EMIF_getConfig(
    EMIF_Config *config
);
```

Arguments

config Pointer to a configuration structure.

Return Value

none

Description

Get EMIF current configuration value

Example

```
EMIF_config emifCfg;
EMIF_getConfig(&emifCfg);
```

EMIF_SUPPORT — *Compile time constant*

Constant

EMIF_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
```

Chapter 9

EMIFA/EMIFB Modules

This chapter describes the EMIFA and EMIFB modules, lists the API functions and macros within the modules, and provides an API reference section.

Topic **Page**

9.1 Overview

The EMIFA and EMIFB modules have simple APIs for configuring the EMIFA and EMIFB registers respectively.

The EMIFA and EMIFB may be configured by passing a configuration structure to `EMIFA_config()` and EMIFB_config() or by passing register values to the `EMIFA_configArgs() and EMIFB_configArgs()` functions. To assist in creating register values, the *EMIFA_<REG>_RMK*() and *EMIFB_<REG>_RMK()* (make) macros construct register values based on field values. In addition, the symbol constants may be used for the field values.

Table 9–1 lists the configuration structure for use with the EMIFA/EMIFB functions.
Table 9–2 lists the functions and constants available in the CSL EMIFA/EMIFB modules.

Table 9–1. EMIFA/EMIFB Configuration Structure

Syntax	Type	Description	See page ...
EMIFA_Config EMIFB_Config	S	Structure used to set up the EMIFA(B) peripheral	9-5

Table 9–2. EMIFA/EMIFB APIs

Syntax	Type	Description	See page ...
EMIFA_config EMIFB_config	F	Sets up the EMIFA(B) using the configuration structure	9-7
EMIFA_configArgs EMIFB_configArgs	F	Sets up the EMIFA(B) using the register value arguments	9-8
EMIFA_getConfig EMIFB_getConfig	F	Reads the current EMIFA(B) configuration values	9-11
EMIFA_SUPPORT EMIFB_SUPPORT	C	A compile time constant that has a value of 1 if the device supports the EMIFA and/or EMIFB modules	9-11

Note: F = Function; C = Constant

9.2 Macros

There are two types of macros: those that access registers and fields, and those that construct register and field values.

Table 9–3 lists the EMIFA and EMIFB macros that access registers and fields, and Table 9–4 lists the EMIFA and EMIFB macros that construct register and field values. The macros themselves are found in Chapter 24, *Using the HAL Macros*.

EMIFA and EMIFB macros are not handle-based.

Table 9–3. EMIFA/EMIFB Macros that Access Registers and Fields

Macro	Description/Purpose	See page ...
EMIFA_ADDR(<REG>) EMIFB_ADDR(<REG>)	Register address	24-12
EMIFA_RGET(<REG>) EMIFB_RGET(<REG>)	Return the value in the peripheral register	24-18
EMIFA_RSET(<REG>,x) EMIFB_RSET(<REG>,x)	Register set	24-20
EMIFA_FGET(<REG>,<FIELD>) EMIFB_FGET(<REG>,<FIELD>)	Return the value of the specified field in the peripheral register	24-13
EMIFA_FSET(<REG>,<FIELD>,fieldval) EMIFB_FSET(<REG>,<FIELD>,fieldval)	Write *fieldval* to the specified field in the peripheral register	24-13
EMIFA_FSETS(<REG>,<FIELD>,<SYM>) EMIFB_FSETS(<REG>,<FIELD>,<SYM>)	Write the symbol value to the specified field in the peripheral	24-17
EMIFA_RGETA(addr,<REG>) EMIFB_RGETA(addr,<REG>)	Get register for a given address	24-19
EMIFA_RSETA(addr,<REG>,x) EMIFB_RSETA(addr,<REG>,x)	Set register for a given address	24-20
EMIFA_FGETA(addr,<REG>,<FIELD>) EMIFB_FGETA(addr,<REG>,<FIELD>)	Get field for a given address	24-13
EMIFA_FSETA(addr,<REG>,<FIELD>,x) EMIFB_FSETA(addr,<REG>,<FIELD>,x)	Set field for a given address	24-16
EMIFA_FSETSA(addr,<REG>,<FIELD>,<SYM>) EMIFB_FSETSA(addr,<REG>,<FIELD>,<SYM>)	Set field symbolically for a given address	24-12

Table 9–4. EMIFA/EMIFB Macros that Construct Register and Field Values

Macro	Description/Purpose	See page ...
EMIFA_<REG>_DEFAULT EMIFB_<REG>_DEFAULT	Register default value	24-21
EMIFA_<REG>_RMK() EMIFB_<REG>_RMK()	Register make	24-23
EMIFA_<REG>_OF() EMIFB_<REG>_OF()	Register value of ...	24-22
EMIFA_<REG>_<FIELD>_DEFAULT EMIFB_<REG>_<FIELD>_DEFAULT	Field default value	24-24
EMIFA_FMK() EMIFB_FMK()	Field make	24-14
EMIFA_FMKS() EMIFB_FMKS()	Field make symbolically	24-15
EMIFA_<REG>_<FIELD>_OF() EMIFB_<REG>_<FIELD>_OF()	Field value of ...	24-24
EMIFA_<REG>_<FIELD>_<SYM> EMIFB_<REG>_<FIELD>_<SYM>	Field symbolic value	24-24

9.3 Configuration Structure

EMIFA_Config
EMIFB_Config

Structures used to set up EMIFA and EMIFB peripherals

Structure

EMIFA_Config
EMIFB_Config

Members

Uint32 gblctl	EMIFA(B)global control register value
Uint32 cectl0	CE0 space control register value
Uint32 cectl1	CE1 space control register value
Uint32 cectl2	CE2 space control register value
Uint32 cectl3	CE3 space control register value
Uint32 sdctl	SDRAM control register value
Uint32 sdtim	SDRAM timing register value
Uint32 sdext	SDRAM extension register value
Uint32 cesec0	CE0 space secondary control register value
Uint32 cesec1	CE1 space secondary control register value
Uint32 cesec2	CE2 space secondary control register value
Uint32 cesec3	CE3 space secondary control register value

Description

These are the EMIFA and EMIFB configuration structures used to set up the EMIFA and EMIFB peripherals, respectively. You create and initialize these structures and then pass their addresses to the `EMIFA_config()` and `EMIFB_config()` functions. You can use literal values or the *EMIFA_<REG>_RMK* and *EMIFB_<REG>_RMK* macros to create the structure member values.

Example

```
EMIFA_Config MyConfigA = {
  0x00003060, /* gblctl */
  0x00000040, /* cectl0 */
  0x404F0323, /* cectl1 */
  0x00000030, /* cectl2 */
  0x00000030, /* cectl3 */
  0x07117000, /* sdctl  */
  0x00000610, /* sdtim  */
  0x00000000, /* sdext  */
  0x00000000, /* cesec0 */
  0x00000000, /* cesec1 */
  0x00000000, /* cesec2 */
  0x00000000  /* cesec3 */
};

EMIFB_Config MyConfigB = {
  0x00003060, /* gblctl */
  0x00000040, /* cectl0 */
  0x404F0323, /* cectl1 */
  0x00000030, /* cectl2 */
  0x00000030, /* cectl3 */
  0x07117000, /* sdctl  */
  0x00000610, /* sdtim  */
  0x00000000, /* sdext  */
  0x00000000, /* cesec0 */
  0x00000000, /* cesec1 */
  0x00000000, /* cesec2 */
  0x00000000  /* cesec3 */
};
…
EMIFA_config(&MyConfigA);
EMIFB_config(&MyConfigB);
```

9.4 Functions

EMIFA_config EMIFB_config

Sets up EMIFA and EMIFB using configuration structures

Function

```
void EMIFA_config(
    EMIFA_Config *config
);

void EMIFB_config(
    EMIFB_Config *config
);
```

Arguments

config Pointer to an initialized configuration structure

Return Value

none

Description

Sets up the EMIFA and/or EMIFB using the configuration respective structures. The values of the structures are written to the EMIFA and EMIFB registers respectively.

Example

```
EMIFA_Config MyConfigA = {
  0x00003060, /* gblctl */
  0x00000040, /* cectl0 */
  0x404F0323, /* cectl1 */
  0x00000030, /* cectl2 */
  0x00000030, /* cectl3 */
  0x07117000, /* sdctl  */
  0x00000610, /* sdtim  */
  0x00000000, /* sdext  */
  0x00000000, /* cesec0 */
  0x00000000, /* cesec1 */
  0x00000000, /* cesec2 */
  0x00000000. /* cesec3 */
};

EMIFB_Config MyConfigB = {
  0x00003060, /* gblctl */
  0x00000040, /* cectl0 */
  0x404F0323, /* cectl1 */
  0x00000030, /* cectl2 */
  0x00000030, /* cectl3 */
  0x07117000, /* sdctl  */
  0x00000610, /* sdtim  */
  0x00000000, /* sdext  */
  0x00000000, /* cesec0 */
  0x00000000, /* cesec1 */
  0x00000000, /* cesec2 */
  0x00000000  /* cesec3 */
};
…
EMIFA_config(&MyConfigA);
EMIFB_config(&MyConfigB);
```

EMIFA_configArgs EMIFB_configArgs

Sets up EMIFA and EMIFB using register value arguments

Function

```
void EMIFA_configArgs(
    Uint32 gblctl,
    Uint32 cectl0,
    Uint32 cectl1,
    Uint32 cectl2,
    Uint32 cectl3,
```

```
    Uint32 sdctl,
    Uint32 sdtim,
    Uint32 sdext,
    Uint32 cesec0,
    Uint32 cesec1,
    Uint32 cesec2,
    Uint32 cesec3
);

void EMIFB_configArgs(
    Uint32 gblctl,
    Uint32 cectl0,
    Uint32 cectl1,
    Uint32 cectl2,
    Uint32 cectl3,
    Uint32 sdctl,
    Uint32 sdtim,
    Uint32 sdext,
    Uint32 cesec0,
    Uint32 cesec1,
    Uint32 cesec2,
    Uint32 cesec3
);
```

Arguments

gblctl	EMIFA(B) global control register value
cectl0	CE0 space control register value
cectl1	CE1 space control register value
cectl2	CE2 space control register value
cectl3	CE3 space control register value
sdctl	SDRAM control register value
sdtim	SDRAM timing register value
sdext	SDRAM extension register value
cesec0	CE0 space secondary register value
cesec1	CE1 space secondary register value

cesec2	CE2 space secondary register value
cesec3	CE3 space secondary register value

Return Value none

Description Set up the EMIFA and EMIFB using the register value arguments. The arguments are written to the EMIFA and EMIFB registers respectively. See also `EMIFA_config(),EMIFB_config() functions.`

Example

```
EMIFA_configArgs(
  0x00003060, /* gblctl */
  0x00000040, /* cectl0 */
  0x404F0323, /* cectl1 */
  0x00000030, /* cectl2 */
  0x00000030, /* cectl3 */
  0x07117000, /* sdctl  */
  0x00000610, /* sdtim  */
  0x00000000, /* sdext  */
  0x00000000, /* cesec0 */
  0x00000000, /* cesec1 */
  0x00000000, /* cesec2 */
  0x00000000  /* cesec3 */
);

EMIFB_configArgs(
  0x00003060, /* gblctl */
  0x00000040, /* cectl0 */
  0x404F0323, /* cectl1 */
  0x00000030, /* cectl2 */
  0x00000030, /* cectl3 */
  0x07117000, /* sdctl  */
  0x00000610, /* sdtim  */
  0x00000000, /* sdext  */
  0x00000000, /* cesec0 */
  0x00000000, /* cesec1 */
  0x00000000, /* cesec2 */
  0x00000000  /* cesec3 */
);
```

EMIFA_getConfig EMIFB_getConfig

Reads the current EMIFA and EMIFB configuration values

Function

```
void EMIFA_getConfig(
    EMIFA_Config *config
);

void EMIFB_getConfig(
    EMIFB_Config *config
);
```

Arguments

config Pointer to a configuration structure.

Return Value

none

Description

Get EMIFA and EMIFB current configuration values.

Example

```
EMIFA_config emifCfgA;
EMIFB_config emifCfgB;

EMIFA_getConfig(&emifCfgA);
EMIFB_getConfig(&emifCfgB);
```

EMIFA_SUPPORT EMIFB_SUPPORT

Compile-time constants

Constant

EMIFA_SUPPORT

EMIFB_SUPPORT

Description

Compile time constants that have a value of 1 if the device supports the EMIFA and EMIFB modules respectively, and 0 otherwise. You are not required to use this constant.

Currently, all devices support this module.

Example

```
#if (EMIFA_SUPPORT)
  /* user EMIFA configuration /
#endif
```

Chapter 10

GPIO Module

This chapter describes the GPIO module, lists the GPIO functions and macros within the module, and provides a GPIO API reference section.

Topic **Page**

10.1 Overview

For TMS320C64x™ devices, the GPIO peripheral provides 16 dedicated general-purpose pins that can be configured as either inputs or outputs. Each GPx pin configured as an input can directly trigger a CPU interrupt or a GPIO event. The properties and functionalities of the GPx pins are covered by a set of CSL APIs.

Table 10–1 lists the configuration structure for use with the GPIO functions. Table 10–2 lists the functions and constants available in the CSL GPIO module.

Table 10–1. GPIO Configuration Structure

Syntax	Type	Description	See page ...
GPIO_Config	S	The GPIO configuration structure used to set the GPIO Global control register	10-7

Table 10–2. GPIO APIs

(a) Primary GPIO Functions

Syntax	Type	Description	See page ...
GPIO_close	F	Closes a GPIO port previously opened via `GPIO_open()`	10-8
GPIO_config	F	Sets up the GPIO global control register using the configuration structure	10-8
GPIO_configArgs	F	Sets up the GPIO global control register using the register values passed in	10-9
GPIO_open	F	Opens a GPIO port for use	10-10
GPIO_reset	C	Resets the given GPIO channel	10-10

(b) Auxiliary GPIO Functions

Syntax	Type	Description	See page ...
GPIO_clear	F	Clears the GPIO Delta registers	10-11
GPIO_deltaLowClear	F	Clears bits of given input pins in Delta Low register	10-11
GPIO_deltaLowGet	F	Indicates if a given input pin has undergone a high-to-low transition. Returns 0 if the transition is not detected.	10-11
GPIO_deltaHighClear	F	Clears the bit of a given input pin in Delta High register	10-12

Syntax	Type	Description	See page ...
GPIO_deltaHighGet	F	Indicates if a given input pin has undergone a low-to-high transition. Returns 0 if the transition is not detected.	10-13
GPIO_getConfig	F	Reads the current GPIO configuration structure	10-13
GPIO_GPINTx	C	Constants dedicated to GPIO interrupt/event signals: GPIO_GPINT0, GPIO_GPINT4, GPIO_GPINT5, GPIO_GPINT6, GPIO_GPINT7	10-14
GPIO_intPolarity	F	Sets the polarity of the GPINTx interrupt/event signals when configured in Pass Through mode	10-14
GPIO_maskLowClear	F	Clears the bits of given input pins in Mask Low register	10-14
GPIO_maskLowSet	F	Enables given pins to cause a CPU interrupt or EDMA event based on corresponding GPxDL or inverted GPxVAL by setting the associated mask bit.	10-15
GPIO_maskHighClear	F	Clears the bits of input pins in Mask High register	10-15
GPIO_maskHighSet	F	Enables given pins to cause a CPU interrupt or EGPIO event based on corresponding GPxDH or GPxVAL by setting the associated mask bit.	10-16
GPIO_pinDisable	F	Disables given pins under the Global Enable register	10-17
GPIO_pinDirection	F	Sets the direction of the given pins. Applies only if the corresponding pins are enabled.	10-17
GPIO_pinEnable	F	Enables the given pins under the Global Enable register	10-18
GPIO_pinRead	F	Reads the detected values of pins configured as inputs and the values to be driven on given output pins.	10-18
GPIO_pinWrite	F	Writes the values to be driven on given output pins.	10-19
GPIO_PINx	C	Constants dedicated to GPIO pins: GPIO_PIN0 –GPIO_PIN15.	10-19
GPIO_read	C	Reads data from a set of pins.	10-20
GPIO_SUPPORT	C	A compile time constant whose value is 1 if the device supports the GPIO module.	10-20
GPIO_write	C	Writes the value to the specified set of GPIO pins.	10-20

Note: F = Function; C = Constant

10.1.1 Using GPIO

To use the GPIO pins, you must first allocate a device using `GPIO_open()`, and then configure the Global Control register to determine the peripheral mode by using the configuration structure to `GPIO_config()` or by passing register value to the `GPIO_configArgs()` function. To assist in creating register values, there are *GPIO_<REG>_RMK* (make) macros that construct register value based on field values. In addition, there are symbol constants that may be used for the field values.

Note that most functions apply to enabled pins only. In order to enable the pins, `GPIO_enablePins()` must be called before using any other functions on these pins.

Important note for C64x users: Migration CSL 2.1 to CSL 2.2

All GPIO APIs have changed with the addition of the handle passed as an input parameter. Although it is possible to include the header file <csl_legacy.h> to avoid any changes to the user's code, it is strongly recommended to update the APIs using the handle–based methodology described in section 1.7.1, Using CSL Handles.

10.2 Macros

There are two types of GPIO macros: those that access registers and fields, and those that construct register and field values.

Table 10–3 lists the GPIO macros that access registers and fields, and Table 10–4 lists the GPIO macros that construct register and field values. The macros themselves are found in Chapter 24, *Using the HAL Macros*.

The GPIO module includes handle-based macros.

Table 10–3. GPIO Macros that Access Registers and Fields

Macro	Description/Purpose	See page ...
GPIO_ADDR(<REG>)	Register address	24-12
GPIO_RGET(<REG>)	Returns the value in the peripheral register	24-18
GPIO_RSET(<REG>,x)	Register set	24-20
GPIO_FGET(<REG>,<FIELD>)	Returns the value of the specified field in the peripheral register	24-13
GPIO_FSET(<REG>,<FIELD>,fieldval)	Writes *fieldval* to the specified field in the peripheral register	24-15
GPIO_FSETS(<REG>,<FIELD>,<SYM>)	Writes the symbol value to the specified field in the peripheral	24-17
GPIO_RGETA(addr,<REG>)	Gets register for a given address	24-19
GPIO_RSETA(addr,<REG>,x)	Sets register for a given address	24-20
GPIO_FGETA(addr,<REG>,<FIELD>)	Gets field for a given address	24-13
GPIO_FSETA(addr,<REG>,<FIELD>, fieldval)	Sets field for a given address	24-16
GPIO_FSETSA(addr,<REG>,<FIELD>, <SYM>)	Sets field symbolically for a given address	24-17
GPIO_ADDRH(h,<REG>)	Returns the address of a memory-mapped register for a given handle	
GPIO_RGETH(h,<REG>)	Returns the value of a register for a given handle	
GPIO_RSETH(h,<REG>,x)	Sets the register value to x for a given handle	
GPIO_FGETH(h,<REG>,<FIELD>)	Returns the value of the field for a given handle	

Table 10–3. GPIO Macros that Access Registers and Fields (Continued)

Macro	Description/Purpose	See page ...
GPIO_FSETH(h,<REG>,<FIELD>, fieldval)	Sets the field value to x for a given handle	
GPIO_FSETSH(h,<REG>,<FIELD>, <SYM>)	Sets field for a given address	

Table 10–4. GPIO Macros that Construct Register and Field Values

Macro	Description/Purpose	See page ...
GPIO_<REG>_DEFAULT	Register default value	24-21
GPIO_<REG>_RMK()	Register make	24-23
GPIO_<REG>_OF()	Register value of ...	24-22
GPIO_<REG>_<FIELD>_DEFAULT	Field default value	24-24
GPIO_FMK()	Field make	24-14
GPIO_FMKS()	Field make symbolically	24-15
GPIO_<REG>_<FIELD>_OF()	Field value of ...	24-24
GPIO_<REG>_<FIELD>_<SYM>	Field symbolic value	24-24

10.3 Configuration Structure

GPIO_Config *GPIO configuration structure used to set up GPIO registers*

Structure GPIO_Config

Members

Uint32 gpgc	GPIO Global control register value
Uint32 gpen	GPIO Enable register value
Uint32 gpdir	GPIO Direction register value
Uint32 gpval	GPIO Value register value
Uint32 gphm	GPIO High Mask register value
Uint32 gplm	GPIO Low Mask register value
Uint32 gppol	GPIO Interrupt Polarity register value

Description This is the GPIO configuration structure used to set up the GPIO registers. You create and initialize this structure, then pass its address to the `GPIO_config()` function. You can use literal values or the *_RMK* macros to create the structure register value.

Example

```
GPIO_Config MyConfig = {
  0x00000031, /* gpgc */
  0x000000F9, /* gpen */
  0x00000070, /* gdir */
  0x00000082, /* gpval */
  0x00000000, /* gphm */
  0x00000000, /* gplm */
  0x00000030  /* gppol */
  };

...
GPIO_config(hGpio,&MyConfig);
```

10.4 Functions

10.4.1 Primary GPIO Functions

GPIO_close *Closes GPIO channel previously opened via GPIO_open()*

Function

```
void   GPIO_close(
    GPIO_Handle    hGpio
);
```

Arguments

hGpio — Handle to GPIO device, see `GPIO_open()`

Return Value

none

Description

This function closes a GPIO channel previously opened via `GPIO_open()`. This function accepts the following device handle: From `GPIO_open()`.

Example

```
GPIO_close(hGpio);
```

GPIO_config *Sets up GPIO modes using a configuration structure*

Function

```
void   GPIO_config(
    GPIO_Handle    hGpio,
    GPIO_Config    *config
);
```

Arguments

hGpio — Handle to GPIO device, see `GPIO_open()`

config — Pointer to an initialized configuration structure

Return Value

none

Description

Sets up the GPIO mode using the configuration structure. The values of the structure are written to the GPIO Global control register. See also `GPIO_configArgs()` and `GPIO_Config`.

Example

```
GPIO_Config MyConfig = {
  0x00000031, /* gpgc */
  GPIO_GPEN_RMK(0x000000F9), /* gpen */
  0x00000070, /* gdir */
  0x00000082, /* gpval */
  0x00000000, /* gphm */
  0x00000000, /* gplm */
  0x00000030  /* gppol */
};
...
GPIO_config(hGpio,&MyConfig);
```

GPIO_configArgs *Sets up GPIO mode using register value passed in*

Function

```
void   GPIO_configArgs(
    GPIO_Handle    hGpio,
    Uint32         gpgc,
    Uint32         gpen,
    Uint32         gpdir,
    Uint32         gpval,
    Uint32         gphm,
    Uint32         gplm,
    Uint32         gppol
 );
```

Arguments

hGpio	Handle to GPIO device, see `GPIO_open()`
gpgc	Global control register value
gpen	GPIO Enable register value
gpdir	GPIO Direction register value
gpval	GPIO Value register value
gphm	GPIO High Mask register value
gplm	GPIO Low Mask register value
gppol	GPIO Interrupt Polarity register value

Return Value

none

Description

Sets up the GPIO mode using the register value passed in. The register value is written to the GPIO Global Control register. See also `GPIO_config()`.

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

Example

```
GPIO_configArgs(hGpio,
  0x00000031, /* gpgc */
  0x000000F9, /* gpen */
  0x00000070, /* gdir */
  0x00000082, /* gpval */
  0x00000000, /* gphm */
  0x00000000, /* gplm */
  0x00000030  /* gppol */
);
```

GPIO_reset — *Resets a given GPIO channel*

Function

```
void   GPIO_reset(
    GPIO_Handle   hGpio
);
```

Arguments

hGpio Device handle obtained by `GPIO_open()`

Return Value

none

Description

Resets the given GPIO channel. The registers are set to their default values, with the exceptions of the Delta High and Delta Low registers, which may be cleared using the `GPIO_clear()` function.

Example

```
GPIO_reset(hGpio);
```

GPIO_open — *Opens GPIO device*

Function

```
GPIO_Handle   GPIO_open(
    int         chaNum,
    Uint32      flags
);
```

Arguments

hGpio Handle to GPIO device, see `GPIO_open()`

chaNum GPIO channel to open:
- ❑ GPIO_DEV0

flags Open flags; may be logical OR of any of the following:
- ❑ GPIO_OPEN_RESET

Return Value

Device Handle Returns a device handle to be used by other GPIO API function calls

Description

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

If the `GPIO_OPEN_RESET` is specified, the GPIO channel is reset, the channel interrupt is disabled and cleared. If the device cannot be opened, `INV` is returned.

Example

```
GPIO_Handle hGpio;
...
hGpio = GPIO_open(GPIO_DEV0,GPIO_OPEN_RESET);
...
```

10.4.2 Auxiliary GPIO Functions and Constants

GPIO_clear *Clears GPIO Delta registers*

Function

```
void   GPIO_clear(
    GPIO_Handle    hGpio
);
```

Arguments

hGpio — Handle to GPIO device, see `GPIO_open()`

Return Value

none

Description

This function clears the GPIO Delta Low (GPDL) and Delta High (GPDH) registers by writing 1 to every bit in these registers.

Example

```
GPIO_clear(hGpio);
```

GPIO_deltaLowClear *Clears bits of given input pins in Delta Low Register*

Function

```
void   GPIO_deltaLowClear(
    GPIO_Handle    hGpio,
    Uint32         pinId
);
```

Arguments

hGpio — Handle to GPIO device, see `GPIO_open()`

pinId — Pin ID to the associated pin to be cleared

Return Value

none

Description

This function clears the bits of given pins register in Delta Low Register.

Example

```
/* Clears one pin */
GPIO_deltaLowClear (hGpio,GPIO_PIN2);
/* Clears several pins */
Uint32 PinID= GPIO_PIN2 | GPIO_PIN3;
GPIO_deltaLowClear (hGpio,PinID);
```

GPIO_deltaLowGet *Returns high-to-low transition detection status for given input pins*

Function

```
Uint32   GPIO_deltaLowGet(
    GPIO_Handle    hGpio,
    Uint32         pinId
);
```

Arguments

hGpio — Handle to GPIO device, see `GPIO_open()`

pinId — Pin ID to the associated pin to be cleared

Return Value status Returns the transition detection status of pinID.

Description This function indicates if a given input pin has undergone a high-to-low transition. Returns the status of the transition detection for the pins associated with the pinId.

Example

```
/* Get transition Detection Status for pin2 */
Uint32 detectionHL;
detectionHL = GPIO_deltaLowGet (hGpio,GPIO_PIN2);
/* Get transition Detection Status for several pins */
Uint32 PinID= GPIO_PIN2 | GPIO_PIN3;
Uint32 detectionHL;
detectionHL = GPIO_deltaLowGet (hGpio,PinID);
/* detectionHL can take the following values :       */
/* 0x00000000 : No high-low transition detected     */
/* 0x00000004 : transition detected for GP2 */
/* 0x00000008 : transition detected for GP3 */
/* 0x0000000C : transitions detected for GP2 and GP3 */
```

GPIO_deltaHighClear *Clears bits of given input pins in Delta High Register*

Function

```
void   GPIO_deltaHighClear(
    GPIO_Handle    hGpio,
    Uint32         pinId
);
```

Arguments hGpio Handle to GPIO device, see `GPIO_open()`

pinId Pin ID to the associated pin to be cleared

Return Value none

Description This function clears the bits of given pin register in Delta High Register.

Example

```
/* Clears one pin */
GPIO_deltaHighClear (hGpio,GPIO_PIN2);
/* Clears several pins */
Uint32 PinID= GPIO_PIN2 | GPIO_PIN3;
GPIO_deltaHighClear (hGpio,PinID);
```

GPIO_deltaHighGet — *Returns low-to-high transition detection status for given input pins*

Function

```
Uint32  GPIO_deltaHighGet(
    GPIO_Handle   hGpio,
    Uint32        pinId
);
```

Arguments

hGpio	Handle to GPIO device, see `GPIO_open( )`
pinId	Pin ID to the associated pin to be cleared

Return Value

status	Returns the transition detection status of pinID.

Description

This function indicates if a given input pin has undergone a low-to-high transition. Returns the status of the transition detection for the pins associated to the pinId.

Example

```
/* Get transition Detection Status for pin2 */
Uint32 detectionLH;
detectionLH = GPIO_deltaHighGet (hGpio,GPIO_PIN2);
/* Get transition Detection Status for several pins */
Uint32 PinID= GPIO_PIN2 | GPIO_PIN3;
Uint32 detectionLH;
detectionLH = GPIO_deltaHighGet (hGpio,PinID);
/* detectionLH can take the following values :   */
/* 0x00000000 :  no high-low transitions detected*/
/* 0x00000004 : transition detected for GP2 */
/* 0x00000008 : transition detected for GP3 */
/* 0x0000000C : transitions detected for GP2 and GP3 */
```

GPIO_getConfig — *Reads the current GPIO Configuration Structure*

Function

```
void  GPIO_getConfig(
    GPIO_Handle   hGpio,
    GPIO_Config   *Config
);
```

Arguments

hGpio	Handle to GPIO device, see `GPIO_open( )`
Config	Pointer to a configuration structure.

Return Value

none

Description

Get GPIO current configuration value

Example

```
GPIO_config GPIOCfg;
GPIO_getConfig(hGpio,&GPIOCfg);
```

GPIO_GPINTx — *Compiler constant dedicated to identify GPIO interrupt/event pins*

Constant GPIO_GPINTx with x={0,4,5,6,7}

Description Set of constants that takes the value of the masks of the associated interrupt/event pins.

These constants are used by the GPIO functions that use signal as the input parameter. Bits of several pins can be set simultaneously by using the logic OR between the masks. See `GPIO_intPolarity()`.

Example

```
GPIO_intPolarity(GPIO_GPINT7,GPIO_RISING);
GPIO_intPolarity(GPIO_GPINT8,GPIO_FALLING);
```

GPIO_intPolarity — *Sets the polarity of the GPINTx interrupt/even signals*

Function

```
void   GPIO_intPolarity(
    GPIO_Handle    hGpio,
    Uint32         signal,
    Uint32         polarity
);
```

Arguments

hGpio — Handle to GPIO device, see `GPIO_open()`

signal — The interrupt/event signal to be configured

polarity — Polarity of the given signal , 2 constants are predefined
- ❑ GPIO_RISING
- ❑ GPIO_FALLING

Return Value none

GPIO_maskLowClear — *Clears bits which cause a CPU interrupt or EDMA event*

Function

```
void   GPIO_maskLowClear(
    GPIO_Handle    hGpio,
    Uint32         pinId
);
```

Arguments

hGpio — Handle to GPIO device, see `GPIO_open()`

pinId — Pin ID to the associated pin to be cleared

Return Value none

Description This function clears the bits of given pins in Mask Low Register. See also `GPIO_maskLowSet()` function.

Example

```
/* Clears one pin mask */
GPIO_maskLowClear (hGpio,GPIO_PIN2);
/* Clears several pins */
Uint32 PinID= GPIO_PIN2 | GPIO_PIN3;
GPIO_maskLowClear (hGpio,PinID);
```

GPIO_maskLowSet *Sets bits which cause a CPU interrupt or EDMA event*

Function

```
void   GPIO_maskLowSet(
    GPIO_Handle    hGpio,
    Uint32         pinId
);
```

Arguments hGpio Handle to GPIO device, see `GPIO_open()`

pinId Pin ID to the associated pin to be set

Return Value none

Description This function sets the bits of given pins to generate an interrupt/event based on corresponding GPxDL or inverted GPxVAL values. See also the `GPIO_maskLowClear()` function.

Example

```
/* Sets one pin mask */
GPIO_maskLowSet (hGpio,GPIO_PIN2);
/* Sets several pins */
Uint32 PinID= GPIO_PIN2 | GPIO_PIN3;
GPIO_maskLowSet (hGpio,PinID);
```

GPIO_maskHighClear *Clears bits which cause a CPU interrupt or EDMA event*

Function

```
void   GPIO_maskHighClear(
    GPIO_Handle    hGpio,
    Uint32         pinId
);
```

Arguments hGpio Handle to GPIO device, see `GPIO_open()`

pinId Pin ID to the associated pin to be cleared

Return Value none

Description This function clears the bits of given pins in Mask High Register. See also `GPIO_maskHighSet()` function.

Example

```
/* Clears one pin mask */
GPIO_maskHighClear (hGpio,GPIO_PIN2);
/* Clears several pins */
Uint32 PinID= GPIO_PIN2 | GPIO_PIN3;
GPIO_maskHighClear (hGpio,PinID);
```

GPIO_maskHighSet *Sets bits which cause a CPU interrupt or EDMA event*

Function

```
void  GPIO_maskHighSet(
    GPIO_Handle   hGpio,
    Uint32        pinId
);
```

Arguments

hGpio Handle to GPIO device, see `GPIO_open()`

pinId Pin ID to the associated pin to be set

Return Value

none

Description

This function sets the bits of given pins to generate an interrupt/event based on corresponding GPxDH or GPxVAL values. See also the `GPIO_maskHighClear()` function.

Example

```
/* Sets one pin mask */
GPIO_maskHighSet (hGpio,GPIO_PIN2);
/* Sets several pins */
Uint32 PinID= GPIO_PIN2 | GPIO_PIN3;
GPIO_maskHighSet (hGpio,PinID);
```

GPIO_pinDisable *Disables the General Purpose Input/Output pins*

Function

```
void   GPIO_pinDisable (
    GPIO_Handle    hGpio,
    Uint32         pinId
);
```

Arguments

hGpio — Handle to GPIO device, see `GPIO_open()`

pinId — pin ID to the associated pin to be cleared

Return Value

none

Description

This function disables the given GPIO pins by setting the associated bits to 0 under the GPEN register. This function is used after having enabled some pins. See also the `GPIO_pinEnable()` function.

Example

```
/* Enables Pins */
GPIO_pinEnable(hGpio,GPIO_PIN1 | GPIO_PIN2 | GPIO_PIN3);
...
/* Disable GP1 pin */
GPIO_pinDisable(hGpio,GPIO_PIN1);
```

GPIO_pinDirection *Sets the direction of the given pins as input or output*

Function

```
Uint32   GPIO_pinDirection(
    GPIO_Handle    hGpio,
    Uint32         pinId
);
```

Arguments

hGpio — Handle to GPIO device, see `GPIO_open()`

pinId — Pin ID to the associated pin to be cleared

direction — Determines the direction of the given pins, 2 constants are predefined:

- ❏ GPIO_INPUT
- ❏ GPIO_OUTPUT

Return Value

CurrentSet — Returns the current pin direction setting

Description

This function sets the associated direction bits of given pins as input or output. Applies only if the given are enabled previously.

Example

```
Uint32 Current_dir;
/* Sets GP1 as input pin */
Current_dir = GPIO_pinDirection(hGpio,GPIO_PIN1,GPIO_INPUT);
/* Sets GP2 and GP3 as output pins */
Uint32 PinID= GPIO_PIN2 | GPIO_PIN3;
Current_dir = GPIO_ pinDirection(hGpio,PinID,GPIO_OUTPUT);
```

GPIO_pinEnable *Enables the General Purpose Input/Output pins*

Function

```
void   GPIO_pinEnable(
    GPIO_Handle    hGpio,
    Uint32         pinId
);
```

Arguments

hGpio Handle to GPIO device, see `GPIO_open()`

pinId Pin ID to the associated pin to be cleared

Return Value

none

Description

This function enables the given GPIO pins by setting the associated bits to 1 under the GPEN register. This function is used after using the given pins as GPIO pins. See also the `GPIO_pinDisable()` function.

Example

```
/* Enables Pins */
GPIO_pinEnable(hGpio,GPIO_PIN1 | GPIO_PIN2 | GPIO_PIN3);
```

GPIO_pinRead *Gets value of given pins*

Function

```
Uint32   GPIO_pinRead(
    GPIO_Handle    hGpio,
    Uint32         pinId
);
```

Arguments

hGpio Handle to GPIO device, see `GPIO_open()`

pinId Pin ID to the associated pin to be set

Return Value

val 0 or 1

Description

If the specified pin has been previously configured as an input, this function returns the value “0” or “1”. If the specified pin has been configured as an output pin, this function returns the value to be driven on the pin.

Example

```
Uint32 val;
/* returns value of pin #2 */
val = GPIO_pinRead (hGpio,GPIO_PIN2);
```

GPIO_pinWrite

Writes value of given output pins

Function

```
void  GPIO_pinWrite(
    GPIO_Handle     hGpio,
    Uint32          pinId,
    Uint32          val
);
```

Arguments

hGpio Handle to GPIO device, see `GPIO_open( )`

pinId Pin ID to the associated pin to be set

val Value to be driven on the given output pin: 0 or 1

Return Value

none

Description

This function sets the value 0 or 1 to be driven on given output pins.

Example

```
Uint32 val;
/* Sets value of one pin to 1*/
GPIO_pinWrite(hGpio,GPIO_PIN2,1);
/* Sets values of several pins to 0*/
Uint32 PinID= GPIO_PIN2 | GPIO_PIN3;
GPIO_ pinWrite(hGpio,PinID,0);
```

GPIO_PINx

Compile constant dedicated to identify each GPIO pin

Constant

GPIO_PINx with x from 0 to 15

Description

Set of constants that takes the value of the masks of the associated pins.

These constants are used by the GPIO functions that use pinID as the input parameter. Bits of several pins can be set simultaneously by using the logic OR between the masks.

Example

```
/* Enables pins */
GPIO_pinEnable (hGpio,GPIO_PIN2 | GPIO_PIN3);
/* Sets Pin3 as an output pin */
Current_dir = GPIO_pinDirection(hGpio,GPIO_PIN3, 1)
/* Sets one pin mask */
GPIO_maskHighSet (hGpio,GPIO_PIN2);
```

GPIO_read

Reads data from a set of pins

Function

```
Uint32  GPIO_read(
    GPIO_Handle   hGpio,
    Uint32        pinMask
);
```

Arguments

hGpio	Handle to GPIO device, see `GPIO_open()`
pinMask	GPIO pin mask for a set of pins

Return Value

Uint32 — Returns the value read on the pins for the pinMask

Description

This function reads data from a set of pins passed on as a pinmask to the function. See also `GPIO_write()`, `GPIO_pinWrite()` and `GPIO_pinRead()`.

Example

```
pinVal = GPIO_read(hGpio,GPIO_PIN8|GPIO_PIN7|GPIO_PIN6);
```

GPIO_SUPPORT

Compile-time constant

Constant

GPIO_SUPPORT

Description

Compile-time constant that has a value of 1 if the device supports the GPIO module and 0 otherwise. You are not required to use this constant.
Note: The GPIO module is not supported on devices without the GPIO peripheral.

Example

```
#if (GPIO_SUPPORT)
  /* user GPIO configuration /
#endif
```

GPIO_write

Writes the value to the specified set of GPIO pins

Function

```
void  GPIO_write(
    GPIO_Handle   hGpio,
    Uint32        pinMask,
    Uint32        val
);
```

Arguments

hGpio	Handle to GPIO device, see `GPIO_open()`
pinMask	GPIO pin mask
val	bit value

Return Value

none

Description

This function writes the value to a set of GPIO pins. See also `GPIO_read()`.

Example

```
GPIO_ write(hGpio,GPIO_PIN2|GPIO_PIN3,0x4);
```

Chapter 11

HPI Module

This chapter describes the HPI module, lists the API functions and macros within the module, and provides an HPI API reference section.

Topic **Page**

11.1 Overview

The HPI module has a simple API for configuring the HPI registers. Functions are provided for reading HPI status bits and setting interrupt events. For C64x™ devices, write and Read memory addresses can be accessed.

Table 11–1 shows the API functions within the HPI module.

Table 11–1. HPI APIs

Syntax	Type	Description	See page ...
HPI_getDspint	F	Reads the DSPINT bit from the HPIC register	11-5
HPI_getEventId	F	Obtain the IRQ event associated with the HPI device	11-5
HPI_getFetch	F	Reads the FETCH flag from the HPIC register and returns its value.	11-5
HPI_getHint	F	Returns the value of the HINT bit of the HPIC register	11-6
HPI_getHrdy	F	Returns the value of the HRDY bit of the HPIC register	11-6
HPI_getHwob	F	Returns the value of the HWOB bit of the HPIC register	11-6
HPI_getReadAddr	F	Returns the Read memory address (HPIAR C64x only)	11-6
HPI_getWriteAddr	F	Returns the Write memory address (HPIAW C64x only)	11-7
HPI_setDspint	F	Writes the value to the DSPINT field of the HPIC register	11-7
HPI_setHint	F	Writes the value to the HINT field of the HPIC register	11-7
HPI_setReadAddr	F	Sets the Read memory address (HPIAR C64x only)	11-8
HPI_setWriteAddr	F	Sets the Write memory address (HPIAW C64x only)	11-8
HPI_SUPPORT	C	A compile-time constant whose value is 1 if the device supports the HPI module	11-8

Note: F = Function; C = Constant

11.2 Macros

There are two types of HPI macros: those that access registers and fields, and those that construct register and field values.

Table 11–2 lists the HPI macros that access registers and fields, and Table 11–3 lists the HPI macros that construct register and field values. The macros themselves are found in Chapter 24, *Using the HAL Macros*.

HPI macros are not handle-based.

Table 11–2. HPI Macros that Access Registers and Fields

Macro	Description/Purpose	See page ...
HPI_ADDR(<REG>)	Register address	24-12
HPI_RGET(<REG>)	Returns the value in the peripheral register	24-18
HPI_RSET(<REG>,x)	Register set	24-20
HPI_FGET(<REG>,<FIELD>)	Returns the value of the specified field in the peripheral register	24-13
HPI_FSET(<REG>,<FIELD>,fieldval)	Writes *fieldval* to the specified field in the peripheral register	24-15
HPI_FSETS(<REG>,<FIELD>,<SYM>)	Writes the symbol value to the specified field in the peripheral	24-17
HPI_RGETA(addr,<REG>)	Gets register for a given address	24-19
HPI_RSETA(addr,<REG>,x)	Sets register for a given address	24-20
HPI_FGETA(addr,<REG>,<FIELD>)	Gets field for a given address	24-13
HPI_FSETA(addr,<REG>,<FIELD>, fieldval)	Sets field for a given address	24-16
HPI_FSETSA(addr,<REG>,<FIELD>, <SYM>)	Sets field symbolically for a given address	24-17

Table 11–3. HPI Macros that Construct Register and Field Values

Macro	Description/Purpose	See page ...
HPI_<REG>_DEFAULT	Register default value	24-21
HPI_<REG>_RMK()	Register make	24-23
HPI_<REG>_OF()	Register value of ...	24-22
HPI_<REG>_<FIELD>_DEFAULT	Field default value	24-24
HPI_FMK()	Field make	24-14
HPI_FMKS()	Field make symbolically	24-15
HPI_<REG>_<FIELD>_OF()	Field value of ...	24-24
HPI_<REG>_<FIELD>_<SYM>	Field symbolic value	24-24

11.3 Functions

HPI_getDspint — *Reads 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()) {
}
```

HPI_getEventId — *Obtains IRQ event associated with 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();
```

HPI_getFetch — *Reads FETCH flag from HPIC register and returns its value*

Function Uint32 HPI_getFetch();

Arguments none

Return Value FETCH Returns the value 0 (always read at 0)

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

Example

```
flag = HPI_getFetch();
```

HPI_getHint — *Returns value of HINT bit of 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();
```

HPI_getHrdy — *Returns value of HRDY bit of 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();
```

HPI_getHwob — *Returns value of HWOB bit of 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();
```

HPI_getReadAddr — *Returns the Read memory address (HPIAR C64x devices only)*

Function Uint32 HPI_getReadAddr();

Arguments none

Return Value HPIAR Read Memory Address

Description This function returns the read memory address set under the HPIAR register (supported by C64x devices only)

Example

```
Uint32 addR;
addR = HPI_getReadAddr();
```

HPI_getWriteAddr *Returns the Write memory address (HPIAW C64x devices only)*

Function Uint32 HPI_getWriteAddr();

Arguments none

Return Value HPIAW Write Memory Address

Description This function returns the write memory address set under the HPIAW register (supported by C64x devices only)

Example

```
Uint32 addW;
addW = HPI_getWriteAddr();
```

HPI_setDspint *Writes value to DSPINT field of HPIC register*

Function

```
void HPI_setDspint(
    Uint32 Val
);
```

Arguments Val Value to write to DSPINT: 1 (writing 0 has no effect)

Return Value none

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

Example

```
HPI_setDspint(0);
HPI_setDspint(1);
```

HPI_setHint *Writes value to HINT field of HPIC register*

Function

```
void HPI_setHint(
    Uint32 Val
);
```

Arguments Val Value to write to HINT: 0 or 1

Return Value none

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

Example

```
HPI_setHint(0);
HPI_setHint(1);
```

HPI_setReadAddr — *Sets the Read memory address (HPIAR C64x devices only)*

Function

```
void HPI_setReadAddr(
    Uint32 address;
);
```

Arguments address Read Memory Address to be set

Return Value none

Description This function sets the read memory address in the HPIAR register (supported by C64x devices only)

Example

```
Uint32 addR = 0x80000400;
HPI_setReadAddr(addR);
```

HPI_setWriteAddr — *Sets the Write memory address (HPIAW C64x devices only)*

Function

```
void HPI_setWriteAddr(
    Uint32 address;
);
```

Arguments address Write Memory Address to be set

Return Value none

Description This function sets the write memory address in the HPIAW register (supported by C64x devices only)

Example

```
Uint32 addW = 0x80000000;
HPI_setWriteAddr(addW);
```

HPI_SUPPORT — *Compile-time constant*

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
```

Chapter 12

I2C Module

This chapter describes the I2C module, lists the API functions and macros within the module, and provides an I2C API reference section.

Topic **Page**

12.1 Overview

The inter–integrated circuit (I2C) module provides an interface between a TMS320c6000 DSP and other devices compliant with Phillips Semiconductors Inter–IC bus (I2C–bus) specification version 2.1 and connected by way of an I2C–bus.

Refer to *SPRU175A TMS320c6000 DSP Inter–Integrated Circuit (I2C) Module Reference Guide* for more details.

Table 12–1 lists the configuration structure for use with the I2C functions. Table 12–2 lists the functions and constants available in the CSL I2C module.

Table 12–1. I2C Configuration Structures

Structure	Purpose	See page ...
I2C_Config	Structure used to configure an I2C interface	12-6

Table 12–2. I2C APIs

(a) Primary I2C Functions

Syntax	Type	Description	See page ...
I2C_close	F	Closes a previously opened I2C device	12-7
I2C_config	F	Configures an I2C using the configuration structure	12-7
I2C_configArgs	F	Configures an I2C using register values	12-8
I2C_open	F	Opens an I2C device for use	12-9
I2C_reset	F	Resets an I2C device	12-9
I2C_resetAll	F	Resets all I2C device registers	12-10
I2C_sendStop	F	Generates a stop condition	12-10
I2C_start	F	Generates a start condition	12-10

(b) Secondary I2C Functions and Constants

Syntax	Type	Description	See page ...
I2C_bb	F	Returns the bus–busy status	12-11
I2C_getConfig	F	Reads the current I2C configuration values	12-11
I2C_getEventId	F	Obtains the event ID for the specified I2C devices	12-12
I2C_getRcvAddr	F	Returns the data receive register address	12-12

Table 12–2. I2C APIs

Syntax	Type	Description	See page ...
I2C_getXmtAddr	F	Returns the data transmit register address	12-12
I2C_intClear	F	Clears the highest priority interrupt flag	12-13
I2C_intClearAll	F	Clears all interrupt flags	12-13
I2C_intEvtDisable	F	Disables the specified I2C interrupt	12-14
I2C_intEvtEnable	F	Enables the specified I2C interrupt	12-14
I2C_OPEN_RESET	C	I2C reset flag, used while opening	12-15
I2C_outOfReset	F	De–asserts the I2C device from reset	12-15
I2C_readByte	F	Performs an 8–bit data read	12-16
I2C_rfull	F	Returns the overrun status of the receiver	12-16
I2C_rrdy	F	Returns the receive data ready interrupt flag value	12-17
I2C_SUPPORT	C	Compile time constant whose value is 1 if the device supports the I2C module	12-15
I2C_writeByte	F	Writes and 8–bit value to the I2C data transmit register	12-17
I2C_xempty	F	Returns the transmitter underflow status	12-18
I2C_xrdy	F	Returns the data transmit ready status	12-18

Note: F = Function; C = Constant;

12.1.1 Using an I2C Device

To use an I2C device, the user must first open it and obtain a device handle using I2C_open(). Once opened, the device handle is used to call the other APIs.

The I2C device can be configured by passing an `I2C_Config` structure to `I2C_config()` or by passing register values to the `I2C_configArgs()` function. To assist in creating register values, the _RMK(make) macros construct register values based on field values. In addition, the symbol constants may be used for the field values.

Once the I2C is used and is no longer needed, it should be closed by passing the corresponding handle to `I2C_close()`.

12.2 Macros

There are two types of I2C macros: those that access registers and fields, and those that construct register and field values.

Table 12–3 lists the I2C macros that access registers and fields, and Table 12–4 lists the I2C macros that construct register and field values. The macros themselves are found in Chapter 24, *Using the HAL Macros*.

I2C macros are handle-based.

Table 12–3. I2C Macros that Access Registers and Fields

Macro	Description/Purpose	See page ...
I2C_ADDR(<REG>)	Register address	24-12
I2C_RGET(<REG>)	Returns the value in the peripheral register	24-18
I2C_RSET(<REG>,x)	Register set	24-20
I2C_FGET(<REG>,<FIELD>)	Returns the value of the specified field in the peripheral register	24-13
I2C_FSET(<REG>,<FIELD>,fieldval)	Writes *fieldval* to the specified field in the peripheral register	24-15
I2C_FSETS(<REG>,<FIELD>,<SYM>)	Writes the symbol value to the specified field in the peripheral	24-17
I2C_RGETA(addr,<REG>)	Gets register for a given address	24-19
I2C_RSETA(addr,<REG>,x)	Sets register for a given address	24-20
I2C_FGETA(addr,<REG>,<FIELD>)	Gets field for a given address	24-13
I2C_FSETA(addr,<REG>,<FIELD>, fieldval)	Sets field for a given address	24-16
I2C_FSETSA(addr,<REG>,<FIELD>, <SYM>)	Sets field symbolically for a given address	24-17
I2C_ADDRH(h,<REG>)	Returns the address of a memory-mapped register for a given handle	24-12
I2C_RGETH(h,<REG>)	Returns the value of a register for a given handle	24-19
I2C_RSETH(h,<REG>,x)	Sets the register value to x for a given handle	24-21
I2C_FGETH(h,<REG>,<FIELD>)	Returns the value of the field for a given handle	24-14
I2C_FSETH(h,<REG>,<FIELD>, fieldval)	Sets the field value to x for a given handle	24-16

Table 12–4. I2C Macros that Construct Register and Field Values

Macro	Description/Purpose	See page...
I2C_<REG>_DEFAULT	Register default value	24-21
I2C_<REG>_RMK()	Register make	24-23
I2C_<REG>_OF()	Register value of ...	24-22
I2C_<REG>_<FIELD>_DEFAULT	Field default value	24-24
I2C_FMK()	Field make	24-14
I2C_FMKS()	Field make symbolically	24-15
I2C_<REG>_<FIELD>_OF()	Field value of ...	24-24
I2C_<REG>_<FIELD>_<SYM>	Field symbolic value	24-24

12.3 Configuration Structure

I2C_Config *Structure used to configure an I2C interface*

Structure I2C_Config;

Members

Uint32	i2coar	Own address register
Uint32	i2cimr	Interrupt mask register
Uint32	i2cclkl	Clock control low register
Uint32	i2cclkh	Clock control high register
Uint32	i2ccnt	Data count register
Uint32	i2csar	Slave address register
Uint32	i2cmdr	Mode register
Uint32	i2cpsc	Prescalar register

Description This is the configuration structure used to dynamically configure the I2C device. The user should create and initialize this structure before passing its address to the `I2C_config()` function.

12.4 Functions

12.4.1 Primary Functions

I2C_close *Closes a previously opened I2C device*

Function

```
void   I2C_close(
    I2C_Handle hI2c
);
```

Arguments

hI2c — Device handle; see `I2C_open()`

Return Value

none

Description

This function closes a previously opened I2C device. The following tasks are performed:
1) The I2C event is disabled and cleared.
2) The I2C registers are set to their default values

Example

```
I2C_Handle hI2c;
...
I2C_close(hI2c);
```

I2C_config *Configures I2C using the configuration structure*

Function

```
void   I2C_config(
    I2C_Handle hI2c,
    I2C_Config *myConfig
);
```

Arguments

hI2c — Device handle; see `I2C_open()`

myConfig — Pointer to an initialized configuration structure

Return Value

none

Description

This function configures the I2C device using the configuration structure which contains members corresponding to each of the I2C registers. These values are directly written to the corresponding I2C device registers.

Example

```
I2C_Handle hI2c
I2C_Config myConfig
...
I2C_config(hI2c,&myConfig);
```

I2C_configArgs *Configures I2C using register values*

Function

```
void   I2C_configArgs(
    I2C_Handle hI2c
    Uint32     i2coar,
    Uint32     i2cimr,
    Uint32     i2cclkl,
    Uint32     i2cclkh,
    Uint32     i2ccnt,
    Uint32     i2csar,
    Uint32     i2cmdr,
    Uint32     i2cpsc
);
```

Arguments

hI2c	Device handle; see I2C_open()
i2coar	Own address register
i2cimr	Interrupt mask register
i2cclkl	Clock control low register
i2cclkh	Clock control high register
i2ccnt	Data count register
i2csar	Slave address register
i2cmdr	Mode register
i2cpsc	Prescalar register

Return Value

none

Description

This function configures the I2C module using the register values passed in as arguments.

Example

```
I2C_Handle hI2c;
...
IRQ_configArgs(hI2c,0x10,0x00,0x08,0x10,0x05,0x10,0x6E0,0x19);
```

I2C_open — *Opens and I2C device for use*

Function

```
I2C_Handle  I2C_open(
    Uint16    devNum
    Uint16    flags
);
```

Arguments

devNum — Specifies the device to be opened

flags — Open flags

- I2C_OPEN_RESET: resets the I2C

Return Value

I2C_Handle — Device handle

INV: open failed

Description

Before the I2C device can be used, it must be opened using this function. Once opened, it cannot be opened again until it is closed. (See I2C_close().) The return value is a unique device handle that is used in subsequent I2C API calls. If the open fails, 'INV' is returned.

Example

```
I2C_Handle hI2c;
...
hI2c = I2C_open(OPEN_RESET);
```

I2C_reset — *Resets an I2C device*

Function

```
void  I2C_reset(
    I2C_Handle hI2c
);
```

Arguments

hI2c — Device handle; see `I2C_open()`

Return Value

none

Description

This function resets the I2C device specified by the handle.

Example

```
I2C_Handle hI2c;
...
I2C_reset(hI2c);
```

I2C_resetAll — *Resets all I2C device registers*

Function

```
void   I2C_resetAll(
    void
);
```

Arguments none

Return Value none

Description This function resets all I2C device registers.

Example

```
I2C_resetAll();
```

I2C_sendStop — *Generates a stop condition*

Function

```
void   I2C_sendStop(
    I2C_Handle hI2c
);
```

Arguments

hI2c	Device handle; see `I2C_open()`

Return Value none

Description This function sets the STP bit in the I2CMDR register, which generates stop conditions.

Example

```
I2C_Handle hI2c;
...
I2C_sendStop(hI2c);
```

I2C_start — *Generates a start condition*

Function

```
void   I2C_start(
    I2C_Handle hI2c
);
```

Arguments

hI2c	Device handle; see `I2C_open()`

Return Value none

Description This function sets the STP bit in the I2CMDR register, which generates data transmission/reception start condition. It is reset to ’0’ by the hardware after the start condition has been generated.

Example

```
I2C_Handle hI2c;
...
I2C_start(hI2c);
```

12.4.2 Auxiliary Functions and Constants

I2C_bb *Returns the bus–busy status*

Function

```
Uint32  I2C_bb(
    I2C_Handle  hI2c
);
```

Arguments

hI2c — Device handle; see `I2C_open()`

Return Value

Uint32 — bus status

- ❏ 0 – free
- ❏ 1 – busy

Description

This function returns the state of the serial bus.

Example

```
I2C_Handle hI2c;
...
if(I2C_bb(hI2c)){
...
;
```

I2C_getConfig *Reads the current I2C configuration values*

Function

```
void  I2C_getConfig(
    I2C_Handle  hI2c,
    I2C_Config  *myConfig
);
```

Arguments

hI2c — Device handle; see `I2C_open()`

myConfig — Pointer to the configuration structure

Return Value

none

Description

This function gets the current I2C configuration values.

Example

```
I2C_Handle hI2c;
I2C_Config i2cCfg;
...
I2C_getConfig(hI2c, &i2cCfg);
```

I2C_getEventId — *Obtains the event ID for the specified I2C device*

Function

```
Uint32  I2C_getEventId(
    I2C_Handle hI2c
);
```

Arguments

hI2c	Device handle; see `I2C_open()`

Return Value

Uint32	Event ID

Description

This function returns the event ID of the interrupt associated with the I2C device.

Example

```
I2C_Handle hI2c;
Uint16  evt;
...
evt = I2C_getEventId(hI2c);
IRQ_enable(evt);
```

I2C_getRcvAddr — *Returns data receive register address*

Function

```
Uint32  I2C_getRcvAddr(
    I2C_Handle hI2c
);
```

Arguments

hI2c	Device handle; see `I2C_open()`

Return Value

Uint32	Data receive register address

Description

This function returns the data receive register address.

Example

```
I2C_Handle hI2c;
Uint32  val;
...
val = I2C_getRcvAddr(hI2c);
```

I2C_getXmtAddr — *Returns the data transmit register address*

Function

```
Uint32  I2C_getXmtAddr(
    I2C_Handle hI2c
);
```

Arguments

hI2c	Device handle; see `I2C_open()`

Return Value Uint32 Data transmit register address

Description This function returns the data transmit register address.

Example

```
I2C_Handle hI2c;
Uint32  val;
...
val = I2C_getXmtAddr(hI2c);
```

I2C_intClear *Clears the highest priority interrupt flag*

Function

```
Uint32  I2C_intClear(
    I2C_Handle hI2c
);
```

Arguments hI2c Device handle; see `I2C_open()`

Return Value Uint32 Interrupt vector register content

Description This function clears the interrupt flag. If there is more than one interrupt flag, it clears the highest priority flag and returns the content of the interrupt vector register (I2CIVR).

Example

```
I2C_Handle hI2c;
Uint32  val;
...
val = I2C_intClear(hI2c);
```

I2C_intClearAll *Clears all interrupt flags*

Function

```
void  I2C_intClearAll(
    I2C_Handle hI2c
);
```

Arguments hI2c Device handle; see `I2C_open()`

Return Value none

Description This function clears all the interrupt flags.

Example

```
I2C_Handle hI2c;
Uint32  val;
...
val = I2C_intClearAll(hI2c);
```

I2C_intEvtDisable *Disables the specified I2C interrupt*

Function

```
void   I2C_intEvtDisable(
    I2C_Handle hI2c,
    Uint32     maskFlag
);
```

Arguments

hI2c Device handle; see `I2C_open()`

maskFlag Interrupt mask

Return Value

none

Description

This function disables the I2C interrupt specified by the maskFlag.

maskFlag can be an OR–ed combination of one or more of the following:

- I2C_EVT_AL – Arbitration Lost Interrupt Enable
- I2C_EVT_NACK – No Acknowledgement Interrupt Enable
- I2C_EVT_ARDY – Register Access Ready Interrupt
- I2C_EVT_RRDY – Data Receive Ready Interrupt
- I2C_EVT_XRDY – Data Transmit Ready Interrupt

Example

```
I2C_Handle hI2c;
Uint32  maskFlag = I2C_EVT_AL|I2C_EVT_RRDY;
...
I2C_intEvtDisable(hI2c, maskFlag);
```

I2C_intEvtEnable *Enables the specified I2C interrupt*

Function

```
void   I2C_intEvtEnable(
    I2C_Handle hI2c,
    Uint32     maskFlag
);
```

Arguments

hI2c Device handle; see `I2C_open()`

maskFlag Interrupt mask

Return Value

none

Description This function enables the I2C interrupt specified by the maskFlag.

maskFlag can be an OR–ed combination of one or more of the following:

- I2C_EVT_AL – Arbitration Lost Interrupt Enable
- I2C_EVT_NACK – No Acknowledgement Interrupt Enable
- I2C_EVT_ARDY – Register Access Ready Interrupt
- I2C_EVT_RRDY – Data Receive Ready Interrupt
- I2C_EVT_XRDY – Data Transmit Ready Interrupt

Example

```
I2C_Handle hI2c;
Uint32  maskFlag = I2C_EVT_AL|I2C_EVT_RRDY;
...
I2C_intEvtEnable(hI2c, maskFlag);
```

I2C_OPEN_RESET *I2C reset flag, used while opening*

Constant I2C_OPEN_RESET

Description This flag is used while opening and I2C device. To open with reset, use I2C_OPEN_RESET. Otherwise, use 0.

Example See `I2C_open()`

I2C_outOfReset *De–asserts the I2C device from reset*

Function

```
void  I2C_outOfReset(
    I2C_Handle hI2c
);
```

Arguments hI2c Device handle; see `I2C_open()`

Return Value none

Description I2C comes out out reset by setting the IRS field of the I2CMDR register.

Example

```
I2C_Handle hI2c;
...
I2C_outOfReset(hI2c);
```

I2C_SUPPORT *Compile time constant*

Constant I2C_SUPPORT

Description Compile time constant that has a value of 1 if the device supports the I2C module and 0 otherwise. You are not required to use this constant. Currently, only the C6713 device supports this module.

Example

```
#if (I2C_SUPPORT)
  /* user I2C configuration */
#endif
```

I2C_readByte — *Performs an 8–bit data read*

Function

```
Uint8  I2C_readByte(
    I2C_Handle hI2c
);
```

Arguments

hI2c	Device handle; see `I2C_open()`

Return Value

Uint8	Received data

Description

This function performs a direct 8–bit read from the data receive register (I2CDRR). This function does not check the receive ready status. To check the receive ready status, use `I2C_rrdy()`.

Example

```
I2C_Handle hI2c;
Uint8  data;
...
data = I2C_readByte(hI2c);
```

I2C_rfull — *Returns the overrun status of the receiver*

Function

```
Uint32  I2C_rfull(
    I2C_Handle hI2c
);
```

Arguments

hI2c	Device handle; see `I2C_open()`

Return Value

Uint32 Overrun status

- ❑ 0 – Normal
- ❑ 1 – Overrun

Description

This function returns the overrun status of the receive shift register. This field is cleared by reading the data receive register or resetting the I2C.

Example

```
I2C_Handle hI2c;
...
if(I2C_rfull(hI2c)){
...
}
```

I2C_rrdy — *Returns the receive data ready interrupt flag value*

Function

```
Uint32  I2C_rrdy(
    I2C_Handle hI2c
);
```

Arguments

hI2c	Device handle; see `I2C_open()`

Return Value

Uint32 Interrupt flag value

- 0 – Receive Data Not Ready
- 1 – Receive Data Ready

Description

This function returns the receive data ready interrupt flag value. The bit is cleared to '0' when I2CDRR is read.

Example

```
I2C_Handle hI2c;
...
if(I2C_rrdy(hI2c)){
...
}
```

I2C_writeByte — *Writes an 8–bit value to the I2C data transmit register*

Function

```
void  I2C_writeByte(
    I2C_Handle hI2c,
    Uint8      val
);
```

Arguments

hI2c	Device handle; see `I2C_open()`
val	8–bit data to send

Return Value

none

Description

This function writes an 8–bit value to the I2C data transmit register. This function does not check the transfer ready status. To check the transfer ready status, use `I2C_xrdy()`.

Example

```
I2C_Handle hI2c;
...
I2C_writeByte(hI2c, 0x34);
```

I2C_xempty *Returns the transmitter underflow status*

Function

```
Uint32  I2C_xempty(
    I2C_Handle hI2c
);
```

Arguments

hI2c	Device handle; see `I2C_open()`

Return Value

Uint32 Underflow status

- 0 – Underflow

Description

This function returns the transmitter underflow status. The value is ’0’ when underflow occurs.

Example

```
I2C_Handle hI2c;
...
if(I2C_xempty(hI2c)){
...
}
```

I2C_xrdy *Returns the data transmit ready status*

Function

```
Uint32  I2C_xrdy(
    I2C_Handle hI2c
);
```

Arguments

hI2c	Device handle; see `I2C_open()`

Return Value

Uint32 Interrupt flag value

- 0 – Transmit Data Not Ready
- 1 – Transmit Data Ready

Description

This function returns the transmit data ready interrupt flag value.

Example

```
I2C_Handle hI2c;
...
if(I2C_xrdy(hI2c)){
...
}
```

Chapter 13

IRQ Module

This chapter describes the IRQ module, lists the API functions and macros within the module, and provides an IRQ API reference section.

Topic **Page**

13.1 Overview

The IRQ module is used to manage CPU interrupts.

Table 13–1 lists the configuration structure for use with the IRQ functions. Table 13–2 lists the functions and constants available in the CSL IRQ module.

Table 13–1. IRQ Configuration Structure

Structure	Purpose	See page ...
IRQ_Config	Interrupt dispatcher configuration structure	13-6

Table 13–2. IRQ APIs

(a) Primary IRQ Functions

Syntax	Type	Description	See page ...
IRQ_clear	F	Clears the event flag from the IFR register	13-9
IRQ_config	F	Dynamically configures an entry in the interrupt dispatcher table	13-9
IRQ_configArgs	F	Dynamically configures an entry in the interrupt dispatcher table	13-10
IRQ_disable	F	Disables the specified event	13-11
IRQ_enable	F	Enables the specified event	13-11
IRQ_globalDisable	F	Globally disables interrupts	13-12
IRQ_globalEnable	F	Globally enables interrupts	13-12
IRQ_globalRestore	F	Restores the global interrupt enable state	13-12
IRQ_reset	F	Resets an event by disabling and then clearing it	13-13
IRQ_restore	F	Restores an event enable state	13-13
IRQ_setVecs	F	Sets the base address of the interrupt vectors	13-14
IRQ_test	F	Allows testing of an event to see if its flag is set in the IFR register	13-14
(b) Auxiliary IRQ Functions			
IRQ_biosPresent	F	Tests if DSP/BIOS is used	13-14
IRQ_EVT_NNNN	C	These are the IRQ events	13-15
IRQ_getArg	F	Reads the user-defined interrupt service routine argument	13-16

Table 13–2. IRQ APIs (Continued)

Syntax	Type	Description	See page ...
IRQ_getConfig	F	Returns the current IRQ set-up using configuration structure	13-16
IRQ_map	F	Maps an event to a physical interrupt number by configuring the interrupt selector MUX registers	13-17
IRQ_nmiDisable	F	Disables the nmi interrupt event	13-18
IRQ_nmiEnable	F	Enables the nmi interrupt event	13-18
IRQ_resetAll	F	Resets all interrupt events by setting the GIE bit to 0 and then disabling and clearing them	13-18
IRQ_set	F	Sets specified event by writing to appropriate ISR register	13-19
IRQ_setArg	F	Sets the user-defined interrupt service routine argument	13-19
IRQ_SUPPORT	C	A compile time constant whose value is 1 if the device supports the IRQ module	13-20

Note: F = Function; C = Constant;

13.2 Macros

There are two types of IRQ macros: those that access registers and fields, and those that construct register and field values.

Table 13–3 lists the IRQ macros that access registers and fields, and Table 13–4 lists the IRQ macros that construct register and field values. The macros themselves are found in Chapter 24, *Using the HAL Macros*.

IRQ macros are not handle-based.

Table 13–3. IRQ Macros that Access Registers and Fields

Macro	Description/Purpose	See page ...
IRQ_ADDR(<REG>)	Register address	24-12
IRQ_RGET(<REG>)	Returns the value in the peripheral register	24-18
IRQ_RSET(<REG>,x)	Register set	24-20
IRQ_FGET(<REG>,<FIELD>)	Returns the value of the specified field in the peripheral register	24-13
IRQ_FSET(<REG>,<FIELD>,fieldval)	Writes *fieldval* to the specified field in the peripheral register	24-15
IRQ_FSETS(<REG>,<FIELD>,<SYM>)	Writes the symbol value to the specified field in the peripheral	24-17
IRQ_RGETA(addr,<REG>)	Gets register for a given address	24-19
IRQ_RSETA(addr,<REG>,x)	Sets register for a given address	24-20
IRQ_FGETA(addr,<REG>,<FIELD>)	Gets field for a given address	24-13
IRQ_FSETA(addr,<REG>,<FIELD>, fieldval)	Sets field for a given address	24-16
IRQ_FSETSA(addr,<REG>,<FIELD>, <SYM>)	Sets field symbolically for a given address	24-17

Table 13–4. IRQ Macros that Construct Register and Field Values

Macro	Description/Purpose	See page...
IRQ_<REG>_DEFAULT	Register default value	24-21
IRQ_<REG>_RMK()	Register make	24-23
IRQ_<REG>_OF()	Register value of ...	24-22
IRQ_<REG>_<FIELD>_DEFAULT	Field default value	24-24
IRQ_FMK()	Field make	24-14
IRQ_FMKS()	Field make symbolically	24-15
IRQ_<REG>_<FIELD>_OF()	Field value of ...	24-24
IRQ_<REG>_<FIELD>_<SYM>	Field symbolic value	24-24

13.3 Configuration Structure

IRQ_Config *Interrupt dispatcher configuration structure*

Structure

```
typedef struct {
    void  *funcAddr;
    Uint32 funcArg;
    Uint32 ccMask;
    Uint32 ieMask;
} IRQ_Config;
```

Members

funcAddr — This is the address of the interrupt service routine to be called when the interrupt happens. This function must be C-callable and must NOT be declared using the *interrupt* keyword. The prototype has the form:

```
void myIsr(
  Uint32 funcArg,
  Uint32 eventId
);
```

funcArg – user defined argument
eventId – the ID of the event that caused the interrupt

funcArg — This is an arbitrary user-defined argument that gets passed to the interrupt service routine. This is useful when the application code wants to pass information to an ISR without using global variables. This argument is also accessible using `IRQ_getArg()` and `IRQ_setArg()`.

ccMask — Cache control mask: determines how the DSP/BIOS dispatcher handles the cache settings when calling an interrupt service routine (ISR). When an interrupt occurs and that event is being handled by the dispatcher, the dispatcher modifies the cache settings based on this argument before calling the ISR. Then when the ISR exits and control is returned back to the dispatcher, the cache settings are restored back to their original state.

The following list shows valid values for ccMask:
(a) `IRQ_CCMASK_NONE`
(b) `IRQ_CCMASK_DEFAULT`

(c) IRQ_CCMASK_PCC_MAPPED
(d) IRQ_CCMASK_PCC_ENABLE
(e) IRQ_CCMASK_PCC_FREEZE
(f) IRQ_CCMASK_PCC_BYPASS
(g) IRQ_CCMASK_DCC_MAPPED
(h) IRQ_CCMASK_DCC_ENABLE
(i) IRQ_CCMASK_DCC_FREEZE
(j) IRQ_CCMASK_DCC_BYPASS

Only certain combinations of the above values are valid:
(a) and (b) are mutually exclusive with all others. This means that if (a) is used, it is used by itself, likewise for (b).

IRQ_CCMASK_NONE means do not touch the cache at all.
IRQ_CCMASK_DEFAULT has the same meaning.

If neither (a) nor (b) is used, then one value from (c) through (f) bitwise OR'ed with a value from (g) through (j) may be used. In other words, choose one value for the PCC control and one value for the DCC control. It is possible to use a PCC value without a DCC value and vise-versa.

ieMask

Interrupt enable mask: determines how interrupts are masked during the processing of the event. The DSP/BIOS interrupt dispatcher allows nested interrupts such that interrupts of higher priority may preempt those of lower priority (priority here is determined by hardware). The ieMask argument determines which interrupts to mask out during processing. Each bit in ieMask corresponds to bits in the interrupt enable register (IER). A "1" bit in ieMask means disable the corresponding interrupt. When processing the interrupt service routine is complete, the dispatcher restores IER back to its original state.

The user may specify a numeric value for the mask or use one of the following predefined symbols:

- IRQ_IEMASK_ALL
- IRQ_IEMASK_SELF
- IRQ_IEMASK_DEFAULT

Use IRQ_IEMASK_ALL to mask out all interrupts including self,

use IRQ_IEMASK_SELF to mask self (prevent an ISR from preempting itself), or use the default which is the same as IRQ_IEMASK_SELF.

Description

This is the configuration structure used to dynamically configure the DSP/BIOS interrupt dispatcher. The interrupt dispatcher may be statically configured using the configuration tool and also dynamically configured using the CSL functions IRQ_config(), IRQ_configArgs(), and IRQ_getConfig(). These functions allow the user to dynamically *hook* new interrupt service routines at runtime.

The DSP/BIOS dispatcher uses a lookup table to gather information for each interrupt. Each entry of this built-in table contains the same members as this configuration structure. Calling IRQ_config() simply copies the configuration structure members into the appropriate locations in the dispatch table.

Example 1

```
IRQ_Config myConfig = {
  myIsr,
  0x00000000,
  IRQ_CCMASK_DEFAULT,
  IRQ_IEMASK_DEFAULT
};
...
IRQ_config(eventId,&myConfig);
...
void myIsr(Uint32 funcArg, Uint32 eventId) {
  ...
}
```

Example 2

```
IRQ_Config myConfig = {
  myIsr,
  0x00000000,
  IRQ_CCMASK_PCC_ENABLE | IRQ_CCMASK_DCC_MAPPED,
  IRQ_IEMASK_ALL
};
...
IRQ_config(eventId,&myConfig);
...
void myIsr(Uint32 funcArg, Uint32 eventId) {
  ...
}
```

13.4 Functions

13.4.1 Primary IRQ Functions

IRQ_clear *Clears event flag*

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 interrupt flag register (IFR). If the event is not mapped to an interrupt, then no action is taken.

Example

```
IRQ_clear(IRQ_EVT_TINT0);
```

IRQ_config *Dynamically configures an entry in the interrupt dispatcher table*

Function

```
void IRQ_config(
    Uint32 eventId,
    IRQ_Config *config
);
```

Arguments

eventId — Event ID. See IRQ_EVT_NNNN for a complete list of events.

config — Pointer to a configuration structure that contains the new configuration information. See `IRQ_Config` for a complete description of this structure.

Return Value

none

Description

This function dynamically configures an entry in the interrupt dispatcher table with the information contained in the configuration structure.

To use this function, a DSP/BIOS configuration .cdb must be defined.

Two constraints must be met before this function has any effect:

1) The event must be mapped to an interrupt

2) The interrupt this event is mapped to must be using the dispatcher

If either of the above two conditions are not met, this function will have no effect.

Example

```
IRQ_Config myConfig = {
  myIsr,
  0x00000000,
  IRQ_CCMASK_DEFAULT,
  IRQ_IEMASK_DEFAULT
};
...
IRQ_config(eventId,&myConfig);
```

IRQ_configArgs *Dynamically configures an entry in the interrupt dispatcher table*

Function

```
void IRQ_configArgs(
    Uint32 eventId,
    void  *funcAddr,
    Uint32 funcArg,
    Uint32 ccMask,
    Uint32 ieMask
);
```

Arguments

eventId	Event ID. See IRQ_EVT_NNNN for a complete list of events.
funcAddr	Address of the interrupt service routine. See the `IRQ_Config` structure definition for more details.
funcArg	Argument that gets passed to the interrupt service routine See the `IRQ_Config` structure definition for more details.
ccMask	Cache control mask. See the `IRQ_Config` structure definition for more details.
ieMask	Interrupt enable mask. See the `IRQ_Config` structure definition for more details.

Return Value

none

Description

This function dynamically configures an entry in the interrupt dispatcher table. It does the same thing as `IRQ_config()` except this function takes the information as arguments rather than passed in a configuration structure.

This function dynamically configures an entry in the interrupt dispatcher table with the information passed in the arguments.

To use this function, a DSP/BIOS configuration .cdb must be defined.

Two constraints must be met before this function has any effect:

1) The event must be mapped to an interrupt

2) The interrupt this event is mapped to must be using the dispatcher

If either of the above two conditions are not met, this function will have no effect.

Example

```
IRQ_configArgs(
  eventId,
  myIsr,
  0x00000000,
  IRQ_CCMASK_DEFAULT,
  IRQ_IEMASK_DEFAULT
);
```

IRQ_disable *Disables specified event*

Function

```
Uint32 IRQ_disable(
    Uint32 eventId
);
```

Arguments

eventId Event ID. See IRQ_EVT_NNNN for a complete list of events.

Return Value

state Returns the old event state. Use with IRQ_restore().

Description

Disables the interrupt associated with the specified event by modifying the interrupt enable register (IER). If the event is not mapped to an interrupt, then no action is taken.

Example

```
IRQ_disable(IRQ_EVT_TINT0);
```

IRQ_enable *Enables 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 event by modifying the interrupt enable register (IER). If the event is not mapped to an interrupt, then no action is taken.

Example

```
IRQ_enable(IRQ_EVT_TINT0);
```

IRQ_globalDisable *Globally disables interrupts*

Function Uint32 IRQ_globalDisable();

Arguments none

Return Value gie Returns the old GIE value

Description This function globally disables interrupts by clearing the GIE bit of the CSR register. The old value of GIE is returned. This is useful for temporarily disabling global interrupts, then restoring them back.

Example

```
Uint32 gie;
gie = IRQ_globalDisable();
...
IRQ_globalRestore(gie);
```

IRQ_globalEnable *Globally enables interrupts*

Function void IRQ_globalEnable();

Arguments none

Return Value none

Description This function globally enables interrupts by setting the GIE bit of the CSR register to 1.This function must be called if the GIE bit is not set before enabling an interrupt event. See also IRQ_globalDisable();

Example

```
IRQ_globalEnable();
IRQ_enable(IRQ_EVT_TINT1);
```

IRQ_globalRestore *Restores the global interrupt enable state*

Function

```
void IRQ_globalRestore(
    Uint32 gie
);
```

Arguments gie Value to restore the global interrupt enable to, (0=disable, 1=enable)

Return Value none

Description This function restores the global interrupt enable state to the value passed in by writing to the GIE bit of the CSR register. This is useful for temporarily disabling global interrupts, then restoring them back.

Example

```
Uint32 gie;
gie = IRQ_globalDisable();
...
IRQ_globalRestore(gie);
```

IRQ_reset *Resets an event by disabling then clearing it*

Function

```
void IRQ_reset(
    Uint32 eventId
);
```

Arguments

eventId Event ID. See IRQ_EVT_NNNN for a complete list of events.

Return Value none

Description This function serves as a shortcut method of performing IRQ_disable(eventId) followed by IRQ_clear(eventId).

Example

```
eventId = DMA_getEventId(hDma);
IRQ_reset(eventId);
```

IRQ_restore *Restores an event-enable state*

Function

```
void IRQ_restore(
    Uint32 eventId,
    Uint32 ie
);
```

Arguments

eventId Event ID. See IRQ_EVT_NNNN for a complete list of events.

ie State to restore the event to (0=disable, 1=enable).

Return Value none

Description This function restores the enable state of the event to the value passed in. This is useful for temporarily disabling an event, then restoring it back.

Example

```
Uint32 ie;
ie = IRQ_disable(eventId);
...
IRQ_restore(ie);
```

IRQ_setVecs — *Sets the base address of the interrupt vectors*

Function

```
void *IRQ_setVecs(
    void *vecs
);
```

Arguments

vecs	Pointer to the interrupt vector table

Return Value

oldVecs	Returns a pointer to the old vector table

Description

Use this function to set the base address of the interrupt vector table.

CAUTION: Changing the interrupt vector table base can have adverse effects on your system because you will be effectively eliminating all interrupt settings that were there previously. The DSP/BIOS kernel and RTDX will more than likely fail if care is not taken when using this function.

Example

```
IRQ_setVecs((void*)0x800000000);
```

IRQ_test — *Allows testing event to see if its flag is set in IFR register*

Function

```
Uint 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 interrupt flag register (IFR). If the event is not mapped to an interrupt, then no action is taken and this function returns 0.

Example

```
while (!IRQ_test(IRQ_EVT_TINT0));
```

13.4.2 Auxiliary IRQ Functions and Constants

IRQ_biosPresent — *Detects if DSP/BIOS is present*

Function

```
Uint32 IRQ_biosPresent();
```

Arguments

none

Return Value

val	Detected value

Description Tests if DSP/BIOS is present. Returns 1 if present, and 0 otherwise.

Example

```
if (IRQ_biosPresent()) {
 IRQ_configArgs(
  eventId,
  myIsr,
  0x00000000,
  IRQ_CCMASK_DEFAULT,
  IRQ_IEMASK_DEFAULT
);
};
```

IRQ_EVT_NNNN *IRQ events*

Constant

IRQ_EVT_DSPINT
IRQ_EVT_TINT0
IRQ_EVT_TINT1
IRQ_EVT_TINT2 C64x only

IRQ_EVT_SDINT
IRQ_EVT_SDINTA C64x only
IRQ_EVT_SDINTB C64x only

IRQ_EVT_GPINT0 C64x only
IRQ_EVT_GPINT4 C64x only
IRQ_EVT_GPINT5 C64x only
IRQ_EVT_GPINT6 C64x only
IRQ_EVT_GPINT7 C64x only

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
IRQ_EVT_PCIWAKE
IRQ_EVT_UINTC64x only

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

IRQ_getArg *Reads the user defined interrupt service routine argument*

Function

```
Uint32 IRQ_getArg(
    Uint32 eventId
);
```

Arguments

eventId	Event ID. See IRQ_EVT_NNNN for a complete list of events.

Return Value

funcArg	Current value of the interrupt service routine argument. For more details, see the `IRQ_Config` structure definition.

Description This function reads the user defined argument from the interrupt dispatcher table and returns it to the user.

Two constraints must be met before this function has any effect:

1) The event must be mapped to an interrupt

2) The interrupt this event is mapped to must be using the dispatcher

If either of the above two conditions are not met, this function will have no effect.

Example

```
Uint32 a = IRQ_getArg(eventId);
```

IRQ_getConfig *Returns the current IRQ set-up using configuration structure*

Function

```
void IRQ_getConfig(
    Uint32 eventId,
    IRQ_Config *config
);
```

Arguments

eventId	Event ID. See IRQ_EVT_NNNN for a complete list of events.
config	Pointer to a configuration structure that will be filled in with information from the dispatcher table. See `IRQ_Config` for a complete description of this structure.

Return Value none

Description This function reads information from the interrupt dispatcher table and stores it in the configuration structure.

Two constraints must be met before this function has any effect:

1) The event must be mapped to an interrupt.

2) The interrupt this event is mapped to must be using the dispatcher.

If either of the above two conditions are not met, this function will have no effect.

Example

```
IRQ_Config myConfig;
...
IRQ_getConfig(eventId,&myConfig);
```

IRQ_map *Maps event to physical interrupt number*

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, 4 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);
```

IRQ_nmiDisable *Disables the NMI interrupt event*

Function void IRQ_nmiDisable();

Arguments none

Return Value none

Description This function disables the NMI interrupt by setting the corresponding bit in IER register to 0.

Example

```
IRQ_nmiDisable();
```

IRQ_nmiEnable *Enables the NMI interrupt event*

Function void IRQ_nmiEnable();

Arguments none

Return Value none

Description This function enables the NMI interrupt by setting the corresponding bit in IER register to 1. Note: When using the DSP/BIOS tool, NMIE interrupt is enabled automatically.

Example

```
IRQ_nmiEnable();
```

IRQ_resetAll *Resets all interrupts events supported by the chip device*

Function void IRQ_resetAll();

Arguments none

Return Value none

Description Resets all the interrupt events supported by the chip device by disabling the global interrupt enable bit (GIE) and then disabling and clearing all the interrupt bits of IER and IFR, respectively.

Example

```
IRQ_resetAll();
```

IRQ_set

Sets specified event by writing to 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 bit in the interrupt set register (ISR). This basically allows software triggering of events. If the event is not mapped to an interrupt, then no action is taken.

Example

```
IRQ_set(IRQ_EVT_TINT0);
```

IRQ_setArg

Sets the user-defined interrupt service routine argument

Function

```
void IRQ_setArg(
    Uint32 eventId,
    Uint32 funcArg
);
```

Arguments

eventId — Event ID. See IRQ_EVT_NNNN for a complete list of events.

funcArg — New value for the interrupt service routine argument. See the `IRQ_Config` structure definition for more details.

Return Value

none

Description

This function sets the user-defined argument in the interrupt dispatcher table.

Two constraints must be met before this function has any effect:

1) The event must be mapped to an interrupt

2) The interrupt this event is mapped to must be using the dispatcher

If either of the above two conditions are not met, this function will have no effect.

Example

```
IRQ_setArg(eventId,0x12345678);
```

IRQ_SUPPORT *Compile time constant*

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
```

Chapter 14

McASP Module

This chapter describes the McASP module, lists the API functions and macros within the module, discusses using a McASP device, and provides a McASP API reference section.

Topic **Page**

14.1 Overview

The McASP module contains a set of API functions for configuring the McASP registers.

Table 14–1 lists the configuration structure for use with the McASP functions. Table 14–2 lists the functions and constants available in the CSL McASP module.

Table 14–1. McASP Configuration Structures

Syntax	Type	Description	See page ...
MCASP_Config	S	Used to configure a McASP device	14-7
MCASP_ConfigGbl	S	Used to configure McASP global receive registers	14-7
MCASP_ConfigRcv	S	Used to configure McASP receive registers	14-8
MCASP_ConfigSrctl	S	Used to configure McASP serial control	14-8
MCASP_ConfigXmt	S	Used to configure McASP transmit registers	14-9

Table 14–2. McASP APIs

(a) Primary Functions

Syntax	Type	Description	See page ...
MCASP_close	F	Closes a McASP device previously opened via `MCASP_open( )`	14-10
MCASP_config	F	Configures the McASP device using the configuration structure	14-10
MCBSP_open	F	Opens a McASP device for use	14-11
MCASP_read32	F	Reads data when the receiver is configured to receive from the data bus	14-11
MCASP_reset	F	Resets McASP registers to their default values	14-12
MCASP_write32	F	Writes data when the transmitter is configured to transmit by the data bus	14-12

(b) Parameters and Constants

Syntax	Type	Description	See page ...
MCASP_DEVICE_CNT	C	McASP device count	14-13
MCASP_OPEN_RESET	C	McASP open reset flag	14-13

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

Table 14–2. McASP APIs (Continued)

Syntax	Type	Description	See page ...
MCASP_SetupClk	T	Parameters for McASP transmit and receive clock registers	14-14
MCASP_SetupFormat	T	Parameters for data stream format: XFMT–RFMT	14-14
MCASP_SetupFsync	T	Parameters for frame synchronization control: AFSXCTL–AFSRCTL	14-15
MCASP_SetupHclk	T	Parameters for McASP transmit and receive high registers	14-15
MCASP_SUPPORT	C	Compile time constant whose value is 1 if the device supports the McASP module	14-16

(c)Auxiliary Functions

Syntax	Type	Description	See page ...
MCASP_clearPins	F	Clears pins which are enabled as GPIO and output	14-16
MCASP_configDit	F	Configures XMASK, XTDM, and AFSXCTL registers for DIT transmission	14-17
MCASP_configGbl	F	Configures McASP device global registers	14-17
MCASP_configRcv	F	Configures McASP device receive registers	14-18
MCASP_configSrctl	F	Configures McASP device serial control registers	14-18
MCASP_configXmt	F	Configures McASP device transmit registers	14-19
MCASP_enableClk	F	Wakes up transmit and/or receive clock, depending on direction	14-19
MCASP_enableFsync	F	Enables frame sync if receiver has internal frame sync	14-20
MCASP_enableHclk	F	Wakes up transmit and or receive high clock, depending on direction	14-21
MCASP_enableSers	F	Enables transmit or receive serializers, depending on direction	14-22
MCASP_enableSm	F	Wakes up transmit and or receive state machine, depending on direction	14-23
MCASP_getConfig	F	Reads the current McASP configuration values	14-23
MCASP_getGblctl	F	Reads the GBLCTL register, depending on direction	14-24
MCASP_read32Cfg	F	Reads the data from rbufNum	14-25

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

Table 14–2. McASP APIs (Continued)

Syntax	Type	Description	See page ...
MCASP_resetRcv	F	Resets the receiver fields in the Global Control register	14-25
MCASP_resetXmt	F	Resets the transmitter fields in the Global Control register	14-26
MCASP_setPins	F	Sets pins which are enabled as GPIO and output	14-26
MCASP_setupClk	F	Sets up McASP transmit and receive clock registers	14-27
MCASP_setupFormat	F	Sets up McASP transmit and receive format registers	14-27
MCASP_setupFsync	F	Sets up McASP transmit and receive frame sync registers	14-28
MCASP_setupHclk	F	Sets up McASP transmit and receive high clock registers	14-28
MCASP_write32Cfg	F	Writes the val into rbufNum	14-29

(c) Interrupt Control Functions

Syntax	Type	Description	See page ...
MCASP_getRcvEventId	F	Retrieves the receive event ID for the given device	14-29
MCASP_getXmtEventId	F	Retrieves the transmit event ID for the given device	14-30

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

14.1.1 Using a McASP Device

To use a McASP device, the user must first open it and obtain a device handle using `MCASP_open()`. Once opened, the device handle should then be passed to other APIs along with other arguments. The McASP device can be configured by passing a `MCASP_Config` structure to `MCASP_config()`. To assist in creating register values, the *MCASP_RMK* (make) macros construct register values based on field values. Once the McASP device is no longer needed, it should be closed by passing the corresponding handle to `MCASP_close()`.

14.2 Macros

There are two types of McASP macros: those that access registers and fields, and those that construct register and field values.

Table 14–3 lists the McASP macros that access registers and fields, and Table 14–4 lists the McASP macros that construct register and field values. The macros themselves are found in Chapter 24, *Using the HAL Macros*.

The McASP module includes handle-based macros.

Table 14–3. McASP Macros that Access Registers and Fields

Macro	Description/Purpose	See page ...
MCASP_ADDR(<REG>)	Register address	24-12
MCASP_RGET(<REG>)	Returns the value in the peripheral register	24-18
MCASP_RSET(<REG>,x)	Register set	24-20
MCASP_FGET(<REG>,<FIELD>)	Returns the value of the specified field in the peripheral register	24-13
MCASP_FSET(<REG>,<FIELD>,fieldval)	Writes *fieldval* to the specified field in the peripheral register	24-15
MCASP_FSETS(<REG>,<FIELD>, <SYM>)	Writes the symbol value to the specified field in the peripheral	24-17
MCASP_RGETA(addr,<REG>)	Gets register for a given address	24-19
MCASP_RSETA(addr,<REG>,x)	Sets register for a given address	24-20
MCASP_FGETA(addr,<REG>,<FIELD>)	Gets field for a given address	24-13
MCASP_FSETA(addr,<REG>,<FIELD>, fieldval)	Sets field for a given address	24-16
MCASP_FSETSA(addr,<REG>,<FIELD>, <SYM>)	Sets field symbolically for a given address	24-17
MCASP_ADDRH(h,<REG>)	Returns the address of a memory-mapped register for a given handle	24-12
MCASP_RGETH(h,<REG>)	Returns the value of a register for a given handle	24-19
MCASP_RSETH(h,<REG>,x)	Sets the register value to x for a given handle	24-21
MCASP_FGETH(h,<REG>,<FIELD>)	Returns the value of the field for a given handle	24-14
MCASP_FSETH(h,<REG>,<FIELD>, fieldval)	Sets the field value to x for a given handle	24-16

Table 14–4. McASP Macros that Construct Register and Field Values

Macro	Description/Purpose	See page ...
MCASP_<REG>_DEFAULT	Register default value	24-21
MCASP_<REG>_RMK()	Register make	24-23
MCASP_<REG>_OF()	Register value of ...	24-22
MCASP_<REG>_<FIELD>_DEFAULT	Field default value	24-24
MCASP_FMK()	Field make	24-14
MCASP_FMKS()	Field make symbolically	24-15
MCASP_<REG>_<FIELD>_OF()	Field value of ...	24-24
MCASP_<REG>_<FIELD>_<SYM>	Field symbolic value	24-24

14.3 Configuration Structure

MCASP_Config *Structure used to configure a McASP device*

Structure MCASP_Config

Members

MCASP_ConfigGbl	*global	Global registers
MCASP_ConfigRcv	*receive	Receive registers
MCASP_ConfigXmt	*transmit	Transmit registers
MCASP_ConfigSrctl	*srctl	Serial control registers

Description This is the McASP configuration structure used to set up a McASP device. The user can create and initialize this structure and then pass its address to the `MCASP_config()` function.

MCASP_ConfigGbl *Structure used to configure McASP global registers*

Structure MCASP_ConfigGbl

Members

Uint32	pfunc	Specifies if the McASP pins are McASP or GPIO pins. Default is 0 = McASP.
Uint32	pdir	Specifies the direction of pins as input/output. Default is 0 = input.
Uint32	ditctl	Specifies the DIT configuration.
Uint32	dlbctl	Specifies the loopback mode and kind loopback (odd serializers (receiver) to even serializers(receivers) or vice versa).
Uint32	amute	Specifies the AMUTE register configuration.

Description This is the McASP configuration structure used to configure McASP device global registers. The user can create and initialize this structure and then pass its address to the `MCASP_configGbl()` function.

MCASP_ConfigRcv *Structure used to configure McASP receive registers*

Structure MCASP_ConfigRcv

Members

Uint32	rmask	Specifies the mask value for receive data.
Uint32	rfmt	Specifies the format for receive data.
Uint32	afsrctl	Specifies the receive frame sync configuration.
Uint32	aclkrctl	Specifies the receive serial clock configuration.
Uint32	ahclkrctl	Specifies the receive high clock configuration.
Uint32	rtdm	Specifies the active receive tdm slots.
Uint32	rintct	Specifies the active events for receive.
Uint32	rclkchk	Specifies the receive serial clock control configuration.

Description This is the McASP configuration structure used to configure McASP device receive registers. The user can create and initialize this structure and then pass its address to the `MCASP_configRcv()` function.

MCASP_ConfigSrctl *Structure used to configure McASP serial control registers.*

Structure MCASP_ConfigSrctl

Members

Uint32	srctl0	Configures the serial control for pin 0.
Uint32	srctl1	Configures the serial control for pin 1.
Uint32	srctl2	Configures the serial control for pin 2.
Uint32	srctl3	Configures the serial control for pin 3.
Uint32	srctl4	Configures the serial control for pin 4.
Uint32	srctl5	Configures the serial control for pin 5.
Uint32	srctl6	Configures the serial control for pin 6.
Uint32	srctl7	Configures the serial control for pin 7.
Uint32	srctl8	Configures the serial control for pin 8.
Uint32	srctl9	Configures the serial control for pin 9.

Uint32	srctl10	Configures the serial control for pin 10.
Uint32	srctl11	Configures the serial control for pin 11.
Uint32	srctl12	Configures the serial control for pin 12.
Uint32	srctl13	Configures the serial control for pin 13.
Uint32	srctl14	Configures the serial control for pin 14.
Uint32	srctl15	Configures the serial control for pin 15.

Description This is the McASP configuration structure used to configure McASP device serial control registers. The user can create and initialize this structure and then pass its address to the `MCASP_configSrctl()` function.

MCASP_ConfigXmt *Structure used to configure McASP transmit registers*

Structure MCASP_ConfigXmt

Members

Uint32	xmask	Specifies the mask value for transmit data.
Uint32	xfmt	Specifies the format for transmit data.
Uint32	afsxctl	Specifies the transmit frame sync configuration.
Uint32	aclkxctl	Specifies the transmit serial clock configuration.
Uint32	ahclkxctl	Specifies the transmit high clock configuration.
Uint32	xtdm	Specifies the active transmit tdm slots.
Uint32	xintct	Specifies the active events for transmit.
Uint32	xclkchk	Specifies the transmit serial clock control configuration.

Description This is the McASP configuration structure used to configure McASP device transmit registers. The user can create and initialize this structure and then pass its address to the `MCASP_configXmt()` function.

14.4 Functions

14.4.1 Primary Functions

MCASP_close *Closes a McASP device previously opened via MCASP_open()*

Function

```
void  MCASP_close(
    MCASP_Handle   hMcasp
);
```

Arguments

hMcasp Handle to McASP device, see `MCASP_open()`

Return Value

none

Description

This function closes a McASP device previously opened via `MCASP_open()`. The following tasks are performed: the registers for the McASP device are set to their defaults, and the McASP handle is closed.

Example

```
MCASP_close(hMcasp);
```

MCASP_config *Configures the McASP device using the configuration structure*

Function

```
void  MCASP_config(
    MCASP_Handle   hMcasp,
    MCASP_Config   *myConfig
);
```

Arguments

hMcasp Handle to McASP device. See `MCASP_open()`

myConfig Pointer to an initialized configuration structure

Return Value

none

Description

This function configures the McASP device using the configuration structure. The values of the structure members are written to the McASP registers. This structure is passed on to the `MCASP_config()` functions. See also `MCASP_getConfig()`, `MCASP_configGbl()`, `MCASP_configRcv()`, `MCASP_configXmt()`, and `MCASP_configSrctl()`.

Example

```
MCASP_Config MyConfig = {
…
MCASP_config(hMcasp,&MyConfig);
```

MCASP_open *Opens a McASP device*

Function

```
MCASP_Handle MCASP_open(
    int         devNum,
    Uint32      flags
);
```

Arguments

devNum — McBSP device to be opened:
- ❑ MCASP_DEV0
- ❑ MCASP_DEV1

flags — Open flags
- ❑ MCBSP_OPEN_RESET: resets the McASP

Return Value

Device Handle — Returns a device handle

Description

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

If the `MCASP_OPEN_RESET` is specified, the McASP device registers are set to their power-on defaults.

Example

```
MCASP_Handle hMcasp;
…
hMcasp = MCASP_open(MCASP_DEV0,MCASP_OPEN_RESET);
or
hMcasp = MCASP_open(MCASP_DEV1, 0);
```

MCASP_read32 *Reads data when the receiver is configured to receive from data bus*

Function

```
Uint32   MCASP_read32(
    MCASP_Handle    hMcasp
);
```

Arguments

hMcasp — Handle to McASP port. See `MCASP_open()`

Return Value

Uint32 — Returns the data received by McASP

Description

This function reads data when the receiver is configured to receive from the peripheral data bus.

Example

```
MCASP_Handle hMcasp;
...
val = MCASP_read32(hMcasp);// Read data from the Address space
                                    for the McASP

MCASP_Handle hMcasp;
Uint32 i;
extern far  dstBuf[8];
...
for (i = 0;i < 8; i++)
{
   val = MCASP_read32(hMcasp); //Reads data
}
```

MCASP_reset — *Resets McASP registers to their default values*

Function

```
void   MCASP_reset(
    MCASP_Handle    hMcasp
);
```

Arguments

hMcbsp Handle to McBSP port. See `MCBSP_open()`

Return Value

none

Description

This function resets the McASP registers to their default values.

Example

```
MCASP_Handle hMcasp;
...
MCASP_reset(hMcasp);
```

MCASP_write32 — *Writes data when the transmitter is configured to transmit by data bus*

Function

```
void   MCASP_write32(
    MCASP_Handle    hMcasp,
    Uint32              val
);
```

Arguments

hMcasp Handle to McASP port. See `MCASP_open()`

val Value to be transmitted

Return Value

none

Description This function writes data when the transmitter is configured to transmit to the peripheral data bus.

Example

```
MCASP_Handle hMcasp;
Uint32 val;
val = 30;
...
MCASP_write32(hMcasp);// Writes data into the Address space
                      for the McASP

MCASP_Handle hMcasp;
Uint32 i;
...
for (i = 0;i < 8; i++)
{
   MCASP_write32(hMcasp,i); // Writes data through the
                            peripheral data bus for McASP
}
```

14.4.2 Parameters and Constants

MCASP_DEVICE_CNT *McASP device count*

Constant MCASP_DEVICE_CNT

Description Compile-time constant that holds the number of McASP devices present on the current device.

MCASP_OPEN_RESET *McASP open reset flag*

Constant MCASP_OPEN_RESET

Description Compile-time constant that holds the number of McASP devices present on the current device.

Example See `MCASP_open()`.

MCASP_SetupClk *Parameters for McASP transmit and receive clock registers*

Structure MCASP_SetupClk

Members

Uint32	syncmode	Transmit and receive clock synchronous flag
Uint32	xclksrc	Transmit clock source
Uint32	xclkpol	Transmit clock polarity
Uint32	xclkdiv	Transmit clock div
Uint32	rclksrc	Receive clock source
Uint32	rclkpol	Receive clock polarity
Uint32	rclkdiv	Receive clock div

Description This is the clock configuration structure used to set up transmit and receive clocks for the McASP device. The user can create and initialize this structure and then pass its address to the `MCASP_setupClk()` function.

MCASP_SetupFormat *Parameters for data streams format: XFMT–RFMT*

Structure MCASP_SetupFormat

Members

Uint32	xbusel	Selects peripheral config/data bus for transmit
MCASP_Dsprep	xdsprep	DSP representation:Q31/Integer
Uint32	xslotsize	8–32 bits XSSZ field – XFMT register
Uint32	xwordsize	Rotation right
Uint32	xalign	Left/right aligned
Uint32	xpad	Pad value for extra bits
Uint32	xpbit	Which bit to pad the extra bits
Uint32	xorder	MSB/LSB XRVRS field – XFMT register
Uint32	xdelay	Bit delay – XFMT register
Uint32	rbusel	Selects peripheral config/data bus for receive
MCASP_Dsprep	rdsprep	DSP representation:Q31/Integer
Uint32	rslotsize	8–32 bits RSSZ field – RFMT register
Uint32	rwordsize	Rotation right
Uint32	ralign	Left/right aligned
Uint32	rpad	Pad value for extra bits
Uint32	rpbit	Which bit to pad the extra bits
Uint32	rorder	MSB/LSB XRVRS field – RFMT register
Uint32	rdelay	FSXDLY Bit delay – RFMT register

Description This is the format configuration structure used to set up transmit and receive formats for the McASP device. The user can create and initialize this structure and then pass its address to the `MCASP_setupFormat()` function.

MCASP_SetupFsync *Parameters for frame sync control: AFSXCTL – AFSRCTL*

Structure MCASP_SetupFsync

Members

Uint32	xmode	TDM–BURST: FSXMOD – AFSXCTL register
Uint32	xslotsize	Number of slots for TDM: FSXMOD – AFSXCTL register
Uint32	xfssrc	Internal/external AFSXE – AFSXCTL register
Uint32	xfspol	Transmit clock polarity FSXPOL – AFSXCTL register
Uint32	fxwid	Transmit frame duration FXWID – AFSXCTL register
Uint32	rmode	TDM–BURST: FSRMOD – AFSRCTL register
Uint32	rslotsize	Number of slots for TDM: FSRMOD – AFSRCTL register
Uint32	rfssrc	Receive internal/external AFSRE – AFSRCTL register
Uint32	rfspol	Receive clock polarity FSRPOL – AFSRCTL register
Uint32	rxwid	Receive frame duration FRWID – AFSRCTL register

Description This is the frame sync configuration structure used to set up transmit and receive frame sync for the McASP device. The user can create and initialize this structure and then pass its address to the `MCASP_setupFsync()` function.

MCASP_SetupHclk *Parameters for McASP transmit and receive high clock registers*

Structure MCASP_SetupHclk

Members

Uint32	xhclksrc	Transmit high clock source
Uint32	xhclkpol	Transmit high clock polarity
Uint32	xhclkdiv	Transmit high clock div
Uint32	rhclksrc	Receive high clock source
Uint32	rhclkpol	Receive high clock polarity
Uint32	rhclkdiv	Receive high clock div

Description This is the high clock configuration structure used to set up transmit and receive high clocks for the McASP device. The user can create and initialize this structure and then pass its address to the `MCASP_setupHclk()` function.

MCASP_SUPPORT *Compile time constant*

Constant MCASP_SUPPORT

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

Currently, the C6713 device supports this module.

Example

```
#if (MCASP_SUPPORT)
  /* user MCASP configuration /
#endif
```

14.4.3 Auxiliary Functions

MCASP_clearPins *Clear pins which are enabled as GPIO and output*

Function

```
void  MCASP_clearPins(
    MCASP_Handle    hMcasp,
    Uint32          pins
)
```

Arguments

hMcasp Handle to McASP device. See `MCASP_open()`

pins Mask value for the pins

Return Value none

Description This function sets the the PDCLR register with the mask value pins specified in 'pins'. This function is used for those McASP pins which are configured as GPIO and are in output direction. Writing a 1 clears the corresponding bit in PDOUT as 1. Writing a 0 leaves it unchanged. The PDCLR register is an alias of the PDOUT register.

Example

```
MCASP_Handle hMcasp;
...
MCASP_clearPins(hMcasp, 0x101);// Clears bits 0,4 in PDOUT
```

MCASP_configDit *Configures XMASK/XTDM/AFSXCTL registers for DIT transmission*

Function

```
void  MCASP_configDit(
    MCASP_Handle    hMcasp,
    Dsprep          dpsrep,
    Uint32          datalen
)
```

Arguments

hMcasp — Handle to McASP device. See `MCASP_open()`

dsprep — Q31/Integer

datalen — 16–24 bits

Return Value

none

Description

This function sets up XMAS, XTDM and AFSXCTL registers depending on the representation MCASP_Dsprep and datalen.

Example

```
MCASP_Handle hMcasp;
...
MCASP_configDit(hMcasp, 1, 24);//Set up DIT transmission for
                                 Q31 24–bit data type
MCASP_configDit(hMcasp, 0, 20);//Set up DIT transmission for
                                 Int 20–bit data type
```

MCASP_configGbl *Configures McASP device global registers*

Function

```
void  MCASP_configGbl(
    MCASP_Handle        hMcasp,
    MCASP_ConfigGbl     *myConfigGbl
)
```

Arguments

hMcasp — Handle to McASP device. See `MCASP_open()`

myConfigGbl — Pointer to the configuration structure

Return Value

none

Description

This function configures McASP device global registers using the configuration structure MCASP_ConfigGbl. The values of the structure–members are written to McASP registers. See also `MCASP_getConfig()`, `MCASP_config()`, `MCASP_configRcv()`, `MCASP_configXmt()`, and `MCASP_configSrctl()`.

Example

```
MCASP_ConfigGbl MyConfigGbl;
...
MCASP_configGbl(hMcasp, &MyConfigGbl);
```

MCASP_configRcv *Configures McASP device receive registers*

Function

```
void  MCASP_configRcv(
    MCASP_Handle        hMcasp,
    MCASP_ConfigRcv     *myConfigRcv
)
```

Arguments

hMcasp	Handle to McASP device. See `MCASP_open()`
myConfigRcv	Pointer to the configuration structure

Return Value none

Description This function configures McASP device receive registers using the configuration structure MCASP_ConfigRcv. The values of the structure–members are written to McASP registers. See also `MCASP_getConfig()`, `MCASP_config()`, `MCASP_configGbl()`, `MCASP_configXmt()`, and `MCASP_configSrctl()`.

Example

```
MCASP_ConfigRcv MyConfigRcv;
...
MCASP_configRcv(hMcasp, &MyConfigRcv);
```

MCASP_configSrctl *Configures McASP device serial control registers*

Function

```
void  MCASP_configSrctl(
    MCASP_Handle        hMcasp,
    MCASP_ConfigSrctl    *myConfigSrctl
)
```

Arguments

hMcasp	Handle to McASP device. See `MCASP_open()`
myConfigSrctl	Pointer to the configuration structure

Return Value none

Description This function configures McASP device serial control registers using the configuration structure MCASP_ConfigSrctl. The values of the structure–members are written to McASP registers. See also `MCASP_getConfig()`, `MCASP_config()`, `MCASP_configGbl()`, `MCASP_configXmt()`, and `MCASP_configRcv()`.

Example

```
MCASP_ConfigSrctl MyConfigSrctl;
...
MCASP_configSrctl(hMcasp, &MyConfigSrctl);
```

MCASP_configXmt *Configures McASP device transmit registers*

Function

```
void  MCASP_configXmt(
    MCASP_Handle        hMcasp,
    MCASP_ConfigXmt     *myConfigXmt
)
```

Arguments

hMcasp	Handle to McASP device. See `MCASP_open()`
myConfigXmt	Pointer to the configuration structure

Return Value

none

Description

This function configures McASP device transmit registers using the configuration structure MCASP_ConfigXmt. The values of the structure–members are written to McASP registers. See also `MCASP_getConfig()`, `MCASP_config()`, `MCASP_configGbl()`, `MCASP_configSrctl()`, and `MCASP_configRcv()`.

Example

```
MCASP_ConfigXmt MyConfigXmt;
...
MCASP_configXmt(hMcasp, &MyConfigXmt);
```

MCASP_enableClk *Wakes up transmit and/or receive clock, depending on direction*

Function

```
void  MCASP_enableClk(
    MCASP_Handle     hMcasp,
    Uint32           direction
)
```

Arguments

hMcasp	Handle to McASP device. See `MCASP_open()`
direction	direction of the clock

- ❏ MCASP_RCV
- ❏ MCASP_XMT
- ❏ MCASP_RCVXMT

- ❏ MCASP_XMTRCV

Return Value none

Description This function wakes up the transmit or receive (or both) clock out of reset by writing into RCLKRST and XCLKRST of GBLCTL. This function should only be used when the corresponding clock is internal.

Example

```
MCASP_Handle hMcasp;
...
MCASP_enableClk(hMcasp, MCASP_RCV); //Wakes up receive clock
MCASP_enableClk(hMcasp, MCASP_XMT); //Wakes up transmit clock
MCASP_enableClk(hMcasp, MCASP_XMTRCV); //Wakes up transmit and
                                       receive clock
MCASP_enableClk(hMcasp, MCASP_RCVXMT); //Wakes up receive and
                                       transmit clock
```

MCASP_enableFsync *Enables frame sync if receiver has internal frame sync*

Function

```
void  MCASP_enableFsync(
    MCASP_Handle     hMcasp,
    Uint32           direction
)
```

Arguments

hMcasp — Handle to McASP device. See `MCASP_open()`

direction — direction of frame sync

- ❏ MCASP_RCV
- ❏ MCASP_XMT
- ❏ MCASP_RCVXMT
- ❏ MCASP_XMTRCV

Return Value none

Description This function wakes up the transmit or recieve (or both) frame sync out of reset by writing into RFSRST and XFSRST of GBLCTL. This function should only be used when the corresponding frame sync is internal.

Example

```
MCASP_Handle hMcasp;
...
MCASP_enableFsync(hMcasp, MCASP_RCV); //Wakes up receive frame
                                      sync
MCASP_enableFsync(hMcasp, MCASP_XMT); //Wakes up transmit
                                      frame sync
MCASP_enableFsync(hMcasp, MCASP_XMTRCV); //Wakes up transmit
                                        and receive frame sync
MCASP_enableFsync(hMcasp, MCASP_RCVXMT); //Wakes up receive
                                        and transmit frame sync
```

MCASP_enableHclk *Wakes up transmit and/or receive high clock, depending on direction*

Function

```
void  MCASP_enableHclk(
    MCASP_Handle     hMcasp,
    Uint32           direction
)
```

Arguments

hMcasp — Handle to McASP device. See `MCASP_open()`

direction — direction of the high clock

- ❏ MCASP_RCV
- ❏ MCASP_XMT
- ❏ MCASP_RCVXMT
- ❏ MCASP_XMTRCV

Return Value

none

Description

This function wakes up the transmit or recieve (or both) high clock out of reset by writing into RHCLKRST and XHCLKRST of GBLCTL. This function should only be used when the corresponding high clock is internal.

Example

```
MCASP_Handle hMcasp;
...
MCASP_enableHclk(hMcasp, MCASP_RCV); //Wakes up receive high
                                     clock
MCASP_enableHclk(hMcasp, MCASP_XMT); //Wakes up transmit high
                                     clock
MCASP_enableHclk(hMcasp, MCASP_XMTRCV); //Wakes up transmit
                                       and receive high clock
MCASP_enableHclk(hMcasp, MCASP_RCVXMT); //Wakes up receive and
                                       transmit high clock
```

MCASP_enableSers *Enables transmit and/or receive serializers, depending on direction*

Function

```
void  MCASP_enableSers(
    MCASP_Handle     hMcasp,
    Uint32           direction
)
```

Arguments

hMcasp — Handle to McASP device. See `MCASP_open()`

direction — direction of the serializers

- ❏ MCASP_RCV
- ❏ MCASP_XMT
- ❏ MCASP_RCVXMT
- ❏ MCASP_XMTRCV

Return Value

none

Description

This function wakes up the transmit or recieve (or both) serializers out of reset by writing into RSRCRL and XSRCLR of GBLCTL.

Example

```
MCASP_Handle hMcasp;
...
MCASP_enableSers(hMcasp, MCASP_RCV); //Receive serializers are
                                      made active
MCASP_enableSers(hMcasp, MCASP_XMT); //Transmit serializers
                                      are made active
MCASP_enableSers(hMcasp, MCASP_XMTRCV); //Transmit and receive
                                     serializers are made active
MCASP_enableSers(hMcasp, MCASP_RCVXMT); //Receive and transmit
                                     serializers are made active
```

MCASP_enableSm *Wakes up transmit and/or receive state machine, depending on direction*

Function

```
void  MCASP_enableSm(
    MCASP_Handle     hMcasp,
    Uint32           direction
)
```

Arguments

hMcasp — Handle to McASP device. See `MCASP_open()`

direction — direction of the state machine

- ❏ MCASP_RCV
- ❏ MCASP_XMT
- ❏ MCASP_RCVXMT
- ❏ MCASP_XMTRCV

Return Value

none

Description

This function wakes up the transmit or recieve (or both) serializers out of reset by writing into RSMRST and XSMRST of GBLCTL.

Example

```
MCASP_Handle hMcasp;
...
MCASP_enableSm(hMcasp, MCASP_RCV); //Wakes up receive
                                   state machine
MCASP_enableSm(hMcasp, MCASP_XMT); //Wakes up transmit
                                   state machine
MCASP_enableSm(hMcasp, MCASP_XMTRCV); //Wakes up transmit and
                                      receive state machine
MCASP_enableSm(hMcasp, MCASP_RCVXMT); //Wakes up receive and
                                      transmit state machine
```

MCASP_getConfig *Reads the current McASP configuration values*

Function

```
void  MCASP_getConfig(
    MCASP_Handle       hMcasp,
    MCASP_Config       *config
)
```

Arguments

hMcasp — Handle to McASP device. See `MCASP_open()`

config — Pointer to the source configuration structure

Return Value none

Description This function gets the current McASP configuration values, as configured in the Global, Receive, Transmit, and Serial Control registers. See `MCASP_config()`.

Example

```
MCASP_Config mcaspCfg;
...
MCASP_getConfig(hMcasp, &mcaspCfg);
```

MCASP_getGblctl *Reads GBLCTL register, depending on the direction*

Function

```
Uint32  MCASP_getGblctl(
    MCASP_Handle      hMcasp,
    Uint32            direction
)
```

Arguments

hMcasp	Handle to McASP device. See `MCASP_open()`
direction	direction ❑ MCASP_RCV ❑ MCASP_XMT ❑ MCASP_RCVXMT ❑ MCASP_XMTRCV

Return Value Uint32 Returns GBCTL, depending on direction

Description This function returns the XGBLCTL value for MCASP_XMT direction, and the RGBLCTL value for MCASP_RCV direction. It returns GBLCTL otherwise.

Example

```
MCASP_Handle hMcasp;
Uint32 gblVal;
hMcasp = MCASP_open(MCASP_DEV0, MCASP_OPEN_RESET);
...
gblVal = MCASP_getGblctl(hMcasp, MCASP_RCV); //RGBLCTL
gblVal = MCASP_getGblctl(hMcasp, MCASP_XMT); //XGBLCTL
gblVal = MCASP_getGblctl(hMcasp, MCASP_XMTRCV); //GBLCTL
```

MCASP_read32Cfg *Reads the data from rbufNum*

Function

```
Uint32   MCASP_read32Cfg(
    MCASP_Handle    hMcasp,
    Uint32          rbufNum
);
```

Arguments

hMcasp Handle to McASP device. See `MCASP_open()`

rbufNum RBUF[0:15]

Return Value

Uint32 Returns data in RBUF[rbufNum]

Description

This function reads data from RBUF[0:15]. It should be used only when the corresponding AXR[0:15] is configured as a receiver and the receiver uses the peripheral configuration bus.

Example

```
MCASP_Handle hMcasp;
Uint32 val;
...
val = MCASP_read32Cfg(hMcasp,9);// Read data from RBUF9, which
                                    is configured as receiver
```

MCASP_resetRcv *Resets the receiver fields in the Global Control register*

Function

```
Uint32void   MCASP_resetRcv(
    MCASP_Handle    hMcasp
);
```

Arguments

hMcasp Handle to McASP device. See `MCASP_open()`

Return Value

none

Description

This function resets the state machine, clears the serial buffer, resets the frame synchronization generator, and resets clocks for the receiver. That is, it clears the RSRCLR, RSMRST, RFRST, RCLKRST, and RHCLKRST in GBLCTL.

Note: It takes 32 receive clock cycles for GBLCTL to update.

Example

```
MCASP_Handle hMcasp;
hMcasp = MCASP_open(MCASP_DEV0, MCASP_OPEN_RESET);
...
MCASP_resetRcv(hMcasp);
```

MCASP_resetXmt — *Resets the transmitter fields in the Global Control register*

Function

```
Uint32void   MCASP_resetXmt(
    MCASP_Handle    hMcasp
);
```

Arguments

hMcasp Handle to McASP device. See `MCASP_open()`

Return Value

none

Description

This function resets the state machine, clears the serial buffer, resets the frame synchronization generator, and resets clocks for the transmitter. That is, it clears the XSRCLR, XSMRST, XFRST, XCLKRST, and XHCLKRST in GBLCTL.

Note: It takes 32 transmit clock cycles for GBLCTL to update.

Example

```
MCASP_Handle hMcasp;
hMcasp = MCASP_open(MCASP_DEV0, MCASP_OPEN_RESET);
...
MCASP_resetXmt(hMcasp);
```

MCASP_setPins — *Sets pins which are enabled as GPIO and output*

Function

```
void  MCASP_setPins(
    MCASP_Handle    hMcasp,
    Uint32              pins
)
```

Arguments

hMcasp Handle to McASP device. See `MCASP_open()`

pins Mask value for the pins

Return Value

none

Description

This function sets up the the PDSET register with the mask value pins specified in 'pins'. This function is used for those McASP pins which are configured as GPIO and are in output direction. Writing a 1 sets the corresponding bit in PDOUT as 1. Writing a 0 leaves it unchanged. The PDSET register is an alias of the PDOUT register.

Example

```
MCASP_Handle hMcasp;
...
MCASP_setPins(hMcasp, 0x101);// Sets bits 0,4 in PDOUT
```

MCASP_setupClk *Sets up McASP transmit and receive clock registers*

Function

```
void  MCASP_setupClk(
    MCASP_Handle      hMcasp,
    MCASP_SetupClk   *setupclk
)
```

Arguments

hMcasp — Handle to McASP device. See `MCASP_open()`

setupclk — Pointer to the configuration structure

Return Value

none

Description

This function configures the McASP device clock registers using the configuration structure `MCASP_SetupClk`. The values of the structure members are written to McASP transmit and receive clock registers.

Example

```
MCASP_SetupClk setupclk;
...
MCASP_setupClk(hMcasp, &setupclk);
```

MCASP_setupFormat *Sets up McASP transmit and receive format registers*

Function

```
void  MCASP_setupFormat(
    MCASP_Handle      hMcasp,
    MCASP_SetupFormat   *setupformat
)
```

Arguments

hMcasp — Handle to McASP device. See `MCASP_open()`

setupformat — Pointer to the configuration structure

Return Value

none

Description

This function configures the McASP device format registers using the configuration structure `MCASP_SetupFormat`. The values of the structure members are written to McASP transmit and receive format registers.

Example

```
MCASP_SetupFormat setupformat;
...
MCASP_setupFormat(hMcasp, &setupformat);
```

MCASP_setupFsync *Sets up McASP transmit and receive frame sync registers*

Function

```
void  MCASP_setupFsync(
    MCASP_Handle      hMcasp,
    MCASP_SetupFsync   *setupfsync
)
```

Arguments

hMcasp	Handle to McASP device. See `MCASP_open()`
setupfsync	Pointer to the configuration structure

Return Value

none

Description

This function configures the McASP device frame sync registers using the configuration structure `MCASP_SetupFsync`. The values of the structure members are written to McASP transmit and receive frame sync registers.

Example

```
MCASP_SetupFsync setupfsync;
...
MCASP_setupFsync(hMcasp, &setupfsync);
```

MCASP_setupHclk *Sets up McASP transmit and receive high clock registers*

Function

```
void  MCASP_setupHclk(
    MCASP_Handle      hMcasp,
    MCASP_SetupHclk   *setuphclk
)
```

Arguments

hMcasp	Handle to McASP device. See `MCASP_open()`
setuphclk	Pointer to the configuration structure

Return Value

none

Description

This function configures the McASP device high clock registers using the configuration structure `MCASP_SetupHclk`. The values of the structure members are written to McASP transmit and receive high clock registers.

Example

```
MCASP_SetupHclk setuphclk;
...
MCASP_setupHclk(hMcasp, &setuphclk);
```

MCASP_write32Cfg *Writes the val into xbufNum*

Function

```
void   MCASP_write32Cfg(
    MCASP_Handle    hMcasp,
    Uint32          xbufNum,
    Uint32          val
);
```

Arguments

hMcasp	Handle to McASP device. See `MCASP_open()`
xbufNum	XBUF[0:15]
val	Value to be transmitted

Return Value none

Description This function writes data into XBUF[0:15]. It should be used only when the corresponding AXR[0:15] is configured as a transmitter and the transmitter uses the peripheral configuration bus.

Example

```
MCASP_Handle hMcasp;
...
MCASP_write32Cfg(hMcasp,4);// Write data into XBUF4, which
                                    is configured as transmitter
```

14.4.4 Interrupt Control Functions

MCASP_getRcvEventId *Returns the receive event ID*

Function

```
Uint32 MCASP_getRcvEventId(
    MCASP_Handle hMcasp
);
```

Arguments

hMcasp	Handle to McASP device. See `MCASP_open()`

Return Value Uint32 Receiver event ID

Description Retrieves the receive event ID for the given device.

Example

```
MCASP_Handle hMcasp
Uint32 eventNo;
hMcasp = MCASP_open(MCASP_DEV0, MCASP_OPEN_RESET);
...
eventNo = MCASP_getRcvEventId(hMcasp);
```

MCASP_getXmtEventId

Returns the transmit event ID

Function

```
Uint32 MCASP_getXmtEventId(
    MCASP_Handle hMcasp
);
```

Arguments

hMcasp	Handle to McASP device. See `MCASP_open()`

Return Value

Uint32	Transmit event ID

Description

Retrieves the transmit event ID for the given device.

Example

```
MCASP_Handle hMcasp
Uint32 eventNo;
hMcasp = MCASP_open(MCASP_DEV0, MCASP_OPEN_RESET);
...
eventNo = MCASP_getXmtEventId(hMcasp);
```

Chapter 15

McBSP Module

This chapter describes the McBSP module, lists the API functions and macros within the module, discusses using a McBSP port, and provides a McBSP API reference section.

Topic **Page**

15.1 Overview

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

Table 15–1 lists the configuration structure for use with the McBSP functions. Table 15–2 lists the functions and constants available in the CSL McBSP module.

Table 15–1. McBSP Configuration Structure

Syntax	Type	Description	See page ...
MCBSP_Config	S	Used to setup a McBSP port	15-7

Table 15–2. McBSP APIs

(a) Primary Functions

Syntax	Type	Description	See page ...
MCBSP_close	F	Closes a McBSP port previously opened via `MCBSP_open()`	15-10
MCBSP_config	F	Sets up the McBSP port using the configuration structure	15-10
MCBSP_configArgs	F	Sets up the McBSP port using the register values passed in	15-12
MCBSP_open	F	Opens a McBSP port for use	15-14
MCBSP_start	F	Starts the McBSP device	15-15

(b) Auxiliary Functions and Constants

Syntax	Type	Description	See page ...
MCBSP_enableFsync	F	Enables the frame sync generator for the given port	15-16
MCBSP_enableRcv	F	Enables the receiver for the given port	15-16
MCBSP_enableSrgr	F	Enables the sample rate generator for the given port	15-17
MCBSP_enableXmt	F	Enables the transmitter for the given port	15-17
MCBSP_getConfig	F	Reads the current McBSP configuration values	15-17
MCBSP_getPins	F	Reads the values of the port pins when configured as general purpose I/Os	15-18
MCBSP_getRcvAddr	F	Returns the address of the data receive register (DRR)	15-18

Note: F = Function; C = Constant

Table 15–2. McBSP APIs (Continued)

Syntax	Type	Description	See page ...
MCBSP_getXmtAddr	F	Returns the address of the data transmit register, DXR	15-19
MCBSP_PORT_CNT	C	A compile time constant that holds the number of serial ports present on the current device	15-19
MCBSP_read	F	Performs a direct 32-bit read of the data receive register DRR	15-19
MCBSP_reset	F	Resets the given serial port	15-19
MCBSP_resetAll	F	Resets all serial ports supported by the device	15-20
MCBSP_rfull	F	Reads the RFULL bit of the serial port control register	15-20
MCBSP_rrdy	F	Reads the RRDY status bit of the SPCR register	15-21
MCBSP_rsyncerr	F	Reads the RSYNCERR status bit of the SPCR register	15-21
MCBSP_setPins	F	Sets the state of the serial port pins when configured as general purpose IO	15-22
MCBSP_SUPPORT	C	A compile time constant whose value is 1 if the device supports the McBSP module	15-22
MCBSP_write	F	Writes a 32-bit value directly to the serial port data transmit register, DXR	15-23
MCBSP_xempty	F	Reads the XEMPTY bit from the SPCR register	15-23
MCBSP_xrdy	F	Reads the XRDY status bit of the SPCR register	15-23
MCBSP_xsyncerr	F	Reads the XSYNCERR status bit of the SPCR register	15-24

(c) Interrupt Control Functions

Syntax	Type	Description	See page ...
MCBSP_getRcvEventId	F	Retrieves the receive event ID for the given port	15-24
MCBSP_getXmtEventId	F	Retrieves the transmit event ID for the given port	15-25

Note: F = Function; C = Constant

15.1.1 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_config()` or by passing register values to the `MCBSP_configArgs()` function. To assist in creating register values, the *MCBSP_MK* (make) macros construct register values based on field values. In addition, the symbol constants 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.

15.2 Macros

There are two types of McBSP macros: those that access registers and fields, and those that construct register and field values.

Table 15–3 lists the McBSP macros that access registers and fields, and Table 15–4 lists the McBSP macros that construct register and field values. The macros themselves are found in Chapter 24, *Using the HAL Macros*.

The McBSP module includes handle-based macros.

Table 15–3. McBSP Macros that Access Registers and Fields

Macro	Description/Purpose	See page ...
MCBSP_ADDR(<REG>)	Register address	24-12
MCBSP_RGET(<REG>)	Returns the value in the peripheral register	24-18
MCBSP_RSET(<REG>,x)	Register set	24-20
MCBSP_FGET(<REG>,<FIELD>)	Returns the value of the specified field in the peripheral register	24-13
MCBSP_FSET(<REG>,<FIELD>,fieldval)	Writes *fieldval* to the specified field in the peripheral register	24-15
MCBSP_FSETS(<REG>,<FIELD>, <SYM>)	Writes the symbol value to the specified field in the peripheral	24-17
MCBSP_RGETA(addr,<REG>)	Gets register for a given address	24-19
MCBSP_RSETA(addr,<REG>,x)	Sets register for a given address	24-20
MCBSP_FGETA(addr,<REG>,<FIELD>)	Gets field for a given address	24-13
MCBSP_FSETA(addr,<REG>,<FIELD>, fieldval)	Sets field for a given address	24-16
MCBSP_FSETSA(addr,<REG>,<FIELD>, <SYM>)	Sets field symbolically for a given address	24-17
MCBSP_ADDRH(h,<REG>)	Returns the address of a memory-mapped register for a given handle	24-12
MCBSP_RGETH(h,<REG>)	Returns the value of a register for a given handle	24-19
MCBSP_RSETH(h,<REG>,x)	Sets the register value to x for a given handle	24-21
MCBSP_FGETH(h,<REG>,<FIELD>)	Returns the value of the field for a given handle	24-14
MCBSP_FSETH(h,<REG>,<FIELD>, fieldval)	Sets the field value to x for a given handle	24-16

Table 15–4. McBSP Macros that Construct Register and Field Values

Macro	Description/Purpose	See page ...
MCBSP_<REG>_DEFAULT	Register default value	24-21
MCBSP_<REG>_RMK()	Register make	24-23
MCBSP_<REG>_OF()	Register value of ...	24-22
MCBSP_<REG>_<FIELD>_DEFAULT	Field default value	24-24
MCBSP_FMK()	Field make	24-14
MCBSP_FMKS()	Field make symbolically	24-15
MCBSP_<REG>_<FIELD>_OF()	Field value of ...	24-24
MCBSP_<REG>_<FIELD>_<SYM>	Field symbolic value	24-24

15.3 Configuration Structure

MCBSP_Config *Used to setup 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

Configuration structure for C64x devices only:

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 rcere0	Enhanced Receive channel enable register 0 value
Uint32 rcere1	Enhanced Receive channel enable register 1 value
Uint32 rcere2	Enhanced Receive channel enable register 2 value

Uint32 rcere3	Enhanced Receive channel enable register 3 value
Uint32 xcere0	Enhanced Transmit channel enable register 0 value
Uint32 xcere1	Enhanced Transmit channel enable register 1 value
Uint32 xcere2	Enhanced Transmit channel enable register 2 value
Uint32 xcere3	Enhanced Transmit channel enable register 3 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_config()` function. You can use literal values or the *MCBSP_RMK* 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_config(hMcbsp,&MyConfig);

/* Configuration structure for C64x devices only */
MCBSP_Config MyConfig = {
  0x00012001, /* spcr ..*/
  0x00010140, /* rcr  ..*/
  0x00010140, /* xcr  ..*/
  0x00000000, /* srgr ..*/
  0x00000000, /* mcr  ..*/
  0x00000000, /* rcere0 */
  0x00000000, /* rcere1 */
  0x00000000, /* rcere2 */
  0x00000000, /* rcere3 */
  0x00000000, /* xcere0 */
  0x00000000, /* xcere1 */
  0x00000000, /* xcere2 */
  0x00000000, /* xcere3 */
  0x00000000  /* pcr  ..*/
};
…
MCBSP_config(hMcbsp,&MyConfig);
```

15.4 Functions

15.4.1 Primary Functions

MCBSP_close *Closes 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

```
MCBSP_close(hMcbsp);
```

MCBSP_config *Sets up McBSP port using configuration structure*

Function

```
void MCBSP_config(
    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_configArgs()` and `MCBSP_Config`.

Example

```
#if (!C64_SUPPORT)
MCBSP_Config MyConfig = {
  0x00012001, /* spcr */
  0x00010140, /* rcr  */
  0x00010140, /* xcr  */
  0x00000000, /* srgr */
  0x00000000, /* mcr  */
  0x00000000, /* rcer */
  0x00000000, /* xcer */
  0x00000000  /* pcr  */
};
#else

/* Configuration structure for C64x devices only */
MCBSP_Config MyConfig = {
  0x00012001, /* spcr */
  0x00010140, /* rcr  */
  0x00010140, /* xcr  */
  0x00000000, /* srgr */
  0x00000000, /* mcr  */
  0x00000000, /* rcere0 */
  0x00000000, /* rcere1 */
  0x00000000, /* rcere2 */
  0x00000000, /* rcere3 */
  0x00000000, /* xcere0 */
  0x00000000, /* xcere1 */
  0x00000000, /* xcere2 */
  0x00000000, /* xcere3 */
  0x00000000  /* pcr  */
};
#endif
…
MCBSP_config(hMcbsp,&MyConfig);
```

MCBSP_configArgs *Sets up McBSP port using register values passed in*

Function

```
void MCBSP_configArgs(
    MCBSP_Handle hMcbsp,
    Uint32 spcr,
    Uint32 rcr,
    Uint32 xcr,
    Uint32 srgr,
    Uint32 mcr,
    Uint32 rcer,
    Uint32 xcer,
    Uint32 pcr
);

For C64x devices:
void MCBSP_configArgs(
    MCBSP_Handle hMcbsp,
    Uint32 spcr,
    Uint32 rcr,
    Uint32 xcr,
    Uint32 srgr,
     Uint32 mcr,
    Uint32 rcere0,
    Uint32 rcere1,
    Uint32 rcere2,
    Uint32 rcere3,
    Uint32 xcere0,
    Uint32 xcere1,
    Uint32 xcere2,
    Uint32 xcere3,
    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

For C64x devices:

rcere0	Enhanced Receive channel enable register 0 value
rcere1	Enhanced Receive channel enable register 1 value
rcere2	Enhanced Receive channel enable register 2 value
rcere3	Enhanced Receive channel enable register 3 value
xcere0	Enhanced Transmit channel enable register 0 value
xcere1	Enhanced Transmit channel enable register 1 value
xcere2	Enhanced Transmit channel enable register 2 value
xcere3	Enhanced Transmit channel enable register 3 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_config()`.

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

Example

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

/* C64x devices */
MCBSP_configArgs(hMcbsp,
  0x00012001, /* spcr   */
  0x00010140, /* rcr    */
  0x00010140, /* xcr    */
  0x00000000, /* srgr   */
  0x00000000, /* mcr    */
  0x00000000, /* rcere0 */
  0x00000000, /* rcere1 */
  0x00000000, /* rcere2 */
  0x00000000, /* rcere3 */
  0x00000000, /* xcere0 */
  0x00000000, /* xcere1 */
  0x00000000, /* xcere2 */
  0x00000000, /* xcere3 */
  0x00000000  /* pcr    */
);
```

MCBSP_open *Opens 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 (if supported by the C64x device)

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);
```

MCBSP_start *Starts McBSP device*

Function

```
void MCBSP_start(
    MCBSP_Handle hMcbsp,
    Uint32 startMask,
    Uint32 SampleRateGenDelay
);
```

Arguments

hMcbsp — Handle to McBSP port. See `MCBSP_open()`

startMask — Allows setting of the different start fields using the following macros:

- MCBSP_XMIT_START: start transmit (XRST)
- MCBSP_RCV_START: start receive (RRST)
- MCBSP_SRGR_START: start Sample rate generator (GRST)
- MCBSP_SRGR_FRAMESYNC: Start frame sync. Generation (FRST)

SampleRateGenDelay — Sample rate generated delay. McBSP logic requires two SRGR clock periods after enabling the sample rate generator for itsl logic to stabilize.
Use this parameter to provide the appropriate delay.
Value = 2 x SRGR clock period/ 4 x C6x Instruction cycle
Default value is 0xFFFFFFFF

Return Value none

Description Use this function to start a transmit and/or receive operation for a McBSP port by passing the handle and mask.

Equivalent to MCBSP_enableXmt(), MCBSP_enableRcv(), MCBSP_enableSrgr(), and MCBSP_enableFsync().

Example

```
MCBSP_start( hMcbsp, MCBSP_RCV_START, 0x00003000);
MCBSP_start(   hMcbsp,   MCBSP_RCV_START   |   MCBSP_XMT_START,
 0x00003000);
```

15.4.2 Auxiliary Functions and Constants

MCBSP_enableFsync *Enables frame sync generator for 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);
```

MCBSP_enableRcv *Enables receiver for 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);
```

MCBSP_enableSrgr *Enables sample rate generator for 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);
```

MCBSP_enableXmt *Enables transmitter for 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);
```

MCBSP_getConfig *Reads the current McBSP configuration values*

Function

```
void MCBSP_getConfig(
    MCBSP_Handle hMcbsp,
    MCBSP_Config *config
);
```

Arguments

hMcbsp Handle to McBSP port. See `MCBSP_open()`

config Pointer to a configuration structure.

Return Value

none

Description

Get McBSP current configuration value

Example

```
MCBSP_config mcbspCfg;
MCBSP_getConfig(hMcbsp,&msbspCfg);
```

MCBSP_getPins *Reads values of 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) {
  ...
}
```

MCBSP_getRcvAddr *Returns address of 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);
```

MCBSP_getXmtAddr *Returns address of data transmit register, DXR*

Function

```
Uint32 MCBSP_getXmtAddr(
    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);
```

MCBSP_PORT_CNT *Compile-time constant*

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
```

MCBSP_read *Performs direct 32-bit read of 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);
```

MCBSP_reset *Resets 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.

Actions Taken:

- ❏ All serial port registers are set to their power-on defaults. The PCR register will be reset to the McBSP reset value and not the device reset value
- ❏ All associated interrupts are disabled and cleared

Example

```
MCBSP_reset(hMcbsp);
```

MCBSP_resetAll *Resets all serial ports supported by the chip device*

Function void MCBSP_resetAll();

Arguments none

Return Value none

Description Resets all serial ports supported by the chip device

Executed Actions:

- ❏ All serial port registers are set to their power-on defaults. The PCR register will be reset to the McBSP reset value and not the device reset value
- ❏ All associated interrupts are disabled and cleared

Example

```
MCBSP_resetAll();
```

MCBSP_rfull *Reads RFULL bit of serial port control register*

Function

```
Uint32 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)) {
 …
}
```

MCBSP_rrdy *Reads RRDY status bit of SPCR register*

Function

```
Uint32 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)) {
  …
}
```

MCBSP_rsyncerr *Reads RSYNCERR status bit of SPCR register*

Function

```
Uint32 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)) {
  …
}
```

MCBSP_setPins *Sets state of 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
);
```

MCBSP_SUPPORT *Compile-time constant*

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
```

MCBSP_write — *Writes 32-bit value directly to 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);
```

MCBSP_xempty — *Reads XEMPTY bit from SPCR register*

Function

```
Uint32 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)) {
  …
}
```

MCBSP_xrdy — *Reads XRDY status bit of SPCR register*

Function

```
Uint32 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)) {
  …
}
```

MCBSP_xsyncerr *Reads XSYNCERR status bit of SPCR register*

Function

```
Uint32 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_ xsyncerr(hMcbsp)) {
  …
}
```

15.4.3 Interrupt Control Functions

MCBSP_getRcvEventId *Retrieves transmit event ID for 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);
```

MCBSP_getXmtEventId *Retrieves transmit event ID for 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);
```

Chapter 16

PCI Module

This chapter describes the PCI module, lists the API functions and macros within the module, discusses the three application domains, and provides a PCI API reference section.

Topic **Page**

16.1 Overview

The PCI module APIs cover the following three application domains:

- ❑ APIs that are dedicated to DSP-PCI Master transfers (mainly starting with the prefix `xfr`)
- ❑ APIs that are dedicated to EEPROM operations such as write, read, and erase (starting with the prefix `eeprom`)
- ❑ APIs that are dedicated to power management

Table 16–1 lists the configuration structure for use with the PCI functions. Table 16–2 lists the functions and constants available in the CSL PCI module.

Table 16–1. PCI Configuration Structure

Syntax	Type	Description	See page ...
PCI_ConfigXfr	S	PCI configuration structure	16-6

Table 16–2. PCI APIs

Syntax	Type	Description	See page ...
PCI_curByteCntGet	F	Returns the current number of bytes left (CCNT)	16-7
PCI_curDspAddrGet	F	Returns the current DSP address (CDSPA)	16-7
PCI_curPciAddrGet	F	Returns the current PCI address (CPCIA)	16-7
PCI_dspIntReqClear	F	Clears the DSP-to-PCI interrupt request bit	16-8
PCI_dspIntReqSet	F	Sets the DSP-to-PCI interrupt request bit	16-8
PCI_eepromErase	F	Erases the specified EEPROM 16-bit address	16-8
PCI_eepromEraseAll	F	Erases the whole EEPROM	16-9
PCI_eepromIsAutoCfg	F	Tests if the PCI reads the configure values from EEPROM	16-9
PCI_eepromRead	F	Reads a 16-bit data from the EEPROM	16-9
PCI_eepromSize	F	Returns EEPROM size	16-10
PCI_eepromTest	F	Tests if EEPROM present	16-10
PCI_eepromWrite	F	Writes a 16-bit data into the EEPROM	16-10

Note: F = Function; C = Constant
† Not supported by 6415/6416 devices

Table 16–2. PCI APIs (Continued)

Syntax	Type	Description	See page ...
PCI_eepromWriteAll	F	Writes a 16-bit data through the whole EEPROM	16-11
PCI_EVT_NNNN	C	PCI events	16-11
PCI_inClear	F	Clears the specified event flag of PCIIS register	16-11
PCI_intDisable	F	Disables the specified PCI event	16-12
PCI_intEnable	F	Enables the specified PCI event	16-12
PCI_intTest	F	Tests an event to see if its flag is set in the PCIIS	16-12
PCI_pwrStatTest†	F	Tests if Current State is equal to Requested State	16-13
PCI_pwrStatUpdate†	F	Updates the Power-Management State	16-13
PCI_SUPPORT	C	Compile time constant	16-13
PCI_xfrByteCntSet	F	Sets the number of bytes to be transferred	16-14
PCI_xfrConfig	F	Configures the PCI registers related to the data transfer between the DSP and PCI	16-14
PCI_xfrConfigArgs	F	Configures the PCI registers related to the data transfer between the DSP and PCI	16-15
PCI_xfrEnable†	F	Enables the internal transfer request to the auxiliary DMA channel	16-15
PCI_xfrFlush	F	Flushes the current transaction	16-16
PCI_xfrGetConfig	F	Returns the current PCI register setting related to the transfer between the DSP and PCI	16-16
PCI_xfrHalt†	F	Prevents the PCI from performing an auxiliary. DMA transfer request	16-16
PCI_xfrStart	F	Enables the specified transaction	16-17
PCI_xfrTest	F	Tests if the transaction is complete	16-17

Note: F = Function; C = Constant
† Not supported by 6415/6416 devices

16.2 Macros

There are two types of PCI macros: those that access registers and fields, and those that construct register and field values.

Table 16–3 lists the PCI macros that access registers and fields, and Table 16–4 lists the PCI macros that construct register and field values. The macros themselves are found in Chapter 24, *Using the HAL Macros*.

PCI macros are not handle-based.

Table 16–3. PCI Macros that Access Registers and Fields

Macro	Description/Purpose	See page ...
PCI_ADDR(<REG>)	Register address	24-12
PCI_RGET(<REG>)	Returns the value in the peripheral register	24-18
PCI_RSET(<REG>,x)	Register set	24-20
PCI_FGET(<REG>,<FIELD>)	Returns the value of the specified field in the peripheral register	24-13
PCI_FSET(<REG>,<FIELD>,fieldval)	Writes *fieldval* to the specified field in the peripheral register	24-15
PCI_FSETS(<REG>,<FIELD>,<SYM>)	Writes the symbol value to the specified field in the peripheral	24-17
PCI_RGETA(addr,<REG>)	Gets register for a given address	24-19
PCI_RSETA(addr,<REG>,x)	Sets register for a given address	24-20
PCI_FGETA(addr,<REG>,<FIELD>)	Gets field for a given address	24-13
PCI_FSETA(addr,<REG>,<FIELD>, fieldval)	Sets field for a given address	24-16
PCI_FSETSA(addr,<REG>,<FIELD>, <SYM>)	Sets field symbolically for a given address	24-17

Table 16–4. PCI Macros that Construct Register and Field Values

Macro	Description/Purpose	See page ...
PCI_<REG>_DEFAULT	Register default value	24-21
PCI_<REG>_RMK()	Register make	24-23
PCI_<REG>_OF()	Register value of ...	24-22
PCI_<REG>_<FIELD>_DEFAULT	Field default value	24-24
PCI_FMK()	Field make	24-14
PCI_FMKS()	Field make symbolically	24-15
PCI_<REG>_<FIELD>_OF()	Field value of ...	24-24
PCI_<REG>_<FIELD>_<SYM>	Field symbolic value	24-24

16.3 Configuration Structure

PCI_ConfigXfr *Structure that sets up registers related to master transfer*

Structure PCI_ConfigXfr

Members

dspma	DSP master address register
pcima	PCI master address register
pcimc	PCI master control register

Description This is the PCI configuration structure used to set up the registers related to the master transfer. You can create and initialize this structure then pass its address to the `PCI_xfrConfigA()` function. You can use literal values.

Example

```
PCI_ConfigXfr myXfrConfig = {
0x80000000,  /* dspma register Addr to be read*/
0xFBE80000,  /* pcima register  XBIAS1 CPLD addr*/
0x00040000  /* pcimc register  4-byte transfer*/
);
   PCI_xfrConfig(&myXfrConfig);
```

16.4 Functions

PCI_curByteCntGet *Returns number of bytes left (CCNT)*

Function Uint32 PCI_curByteCntGet();

Arguments none

Return Value number of bytes

Description Returns number of bytes left on the current master transaction.

Example

```
Uint32 nbBytes;
nbBytes = PCI_curByteCntGet();
```

PCI_curDspAddrGet *Returns current DSP address*

Function Uint32 PCI_curDspAddrGet();

Arguments none

Return Value DSP Address

Description Returns the current DSP Address of the master transactions.

Example

```
Uint32 dspAddr;
dspAddr = PCI_curDspAddrGet();
```

PCI_curPciAddrGet *Returns current PCI address*

Function Uint32 PCI_curPciAddrGet();

Arguments none

Return Value PCI Address

Description Returns the current PCI Address of the master transactions.

Example

```
Uint32 pciAddr;
pciAddr = PCI_curPciAddrGet();
```

PCI_dspIntReqClear *Clears DSP-to-PCI interrupt request bit*

Function void PCI_dspIntReqClear();

Arguments none

Return Value none

Description Clears the DSP-to-PCI interrupt request bit of the RSTSRC register.

Example
```
PCI_dspIntReqClear();
```

PCI_dspIntReqSet *Sets DSP-to-PCI interrupt request bit*

Function void PCI_dspIntReqSet();

Arguments none

Return Value none

Description Sets the DSP-to-PCI interrupt request bit of the RSTSRC register.

Example PCI_dspIntReqSet();

PCI_eepromErase *Erases specified EEPROM byte*

Function
```
Uint32 PCI_eepromErase(
Uint32 eeaddr
);
```

Arguments eeaddr address of the 16-bit data to be erased

Return Value 0 or 1 (success)

Description Erases the 16-bit data at the specified address. The “Enable Write EWEN” is performed under this function.

Example
```
Uint32 success;
success = PCI_eepromErase(0x00000002);
```

PCI_eepromEraseAll *Erases entire EEPROM*

Function Uint32 PCI_eepromEraseAll()

Arguments none address of the 16-bit data to be erased

Return Value 0 or 1 (success)

Description Erases the full EEPROM.

Example

```
Uint32 success;
success = PCI_eepromEraseAll();
```

PCI_eepromIsAutoCfg *Tests if PCI reads configure-values from EEPROM*

Function Uint32 PCI_eepromIsAutoCfg();

Arguments none

Return Value 0 or 1

Description Tests if the PCI reads configure-values from EEPROM. Returns value of the EEAI field of EECTL register.

Example

```
Uint32 x;
x = PCI_eepromIsAutoCfg();
```

PCI_eepromRead *Reads 16-bit data from EEPROM*

Function

```
Uint16 PCI_eepromRead(
Uint32 eeaddr
);
```

Arguments eeaddr Address of the 16-bit data to be read from EEPROM.

Return Value value of the 16-bit data

Description Reads the 16-bit data at the specified address from EEPROM.

Example

```
Uint16 eepromdata;
eepromdata = PCI_eepromRead(0x00000001);
```

PCI_eepromSize *Returns EEPROM size*

Function Uint32 PCI_eepromSize();

Arguments none

Return Value value of the size code

Description Returns the code associated with the size of the EEPROM.

- 0x0 :000 No EEPROM present
- 0x1 :001 1K_EEPROM
- 0x2 :010 2K_EEPROM
- 0x3 :011 4K_EEPROM (6415/6416 devices support the 4K_EEPROM only)
- 0x4 :100 16K_EEPROM

Example

```
Uint32 eepromSZ;
eepromSZ = PCI_eepromSize();
```

PCI_eepromTest *Tests if EEPROM is present*

Function Uint32 PCI_eepromTest();

Arguments none

Return Value 0 or 1

Description Tests if EEPROM is present by reading the code size bits EESZ[2:0]

Example

```
Uint32 eepromIs;
eepromIs = PCI_eepromTest();
```

PCI_eepromWrite *Writes 8-bit data into EEPROM*

Function

```
Uint32 PCI_eepromWrite(
Uint32 eeaddr,
Uint16 eedata
);
```

Arguments

eeaddr Address of the byte to read from EEPROM.

eedata 16-bit data to be written.

Return Value 0 or 1

Description Writes the 16-bit data into the specified EEPROM address. The “Enable Write EWEN” is performed under this function.

Example

```
Uint16 x;
x = PCI_eepromWrite(0x123,0x8888);
```

PCI_eepromWriteAll — *Writes 16-bit data into entire EEPROM*

Function

```
Uint32 PCI_eepromWriteAll(
Uint16 eedata
);
```

Arguments eedata 16-bit data to be written.

Return Value 0 or 1

Description Writes the 16-bit data into the entire EEPROM.

Example

```
Uint16 x;
x = PCI_eepromWriteAll(0x1234);
```

PCI_EVT_NNN — *PCI events (PCIIEN register)*

Constant

PCI_EVT_DMAHALTED
PCI_EVT_PRST
PCI_EVT_EERDY
PCI_EVT_CFGERR
PCI_EVT_CFGDONE
PCI_EVT_MASTEROK
PCI_EVT_PWRHL
PCI_EVT_PWRLH
PCI_EVT_HOSTSW
PCI_EVT_PCIMASTER
PCI_EVT_PCITARGET
PCI_EVT_PWRMGMT

Description These are the PCI events. For more details regarding these events, refer to the *TMS320C6000 Peripherals Reference Guide* (SPRU190).

PCI_intClear — *Clears the specified event flag*

Function

```
void PCI_intClear(
Uint32 eventPci
);
```

Arguments eventPci See PCI_EVT_NNNN for a complete list of PCI events.

Return Value none

Description Clears the specified event flag of PCIIS register by writing ’1’ to the associated bit.

Example
```
PCI_intClear(PCI_EVT_MASTEROK);
```

PCI_intDisable *Disable specified PCI event*

Function
```
void PCI_intDisable(
Uint32 eventPci
);
```

Arguments eventPci See PCI_EVT_NNNN for a complete list of PCI events.

Return Value none

Description Disables the specified PCI event.

Example
```
PCI_intDisable(PCI_EVT_MASTEROK);
```

PCI_intEnable *Enables specified PCI event*

Function
```
void PCI_intEnable(
Uint32 eventPci
);
```

Arguments eventPci See PCI_EVT_NNNN for a complete list of PCI events.

Return Value none

Description Enables the specified PCI event.

Example
```
PCI_intEnable(PCI_EVT_MASTEROK);
```

PCI_intTest *Test if specified PCI event flag is set*

Function
```
Uint32 PCI_intTest(
Uint32 eventPci
);
```

Arguments eventPci See PCI_EVT_NNNN for a complete list of PCI events.

Return Value 0 or 1

Description Tests if the specified event flag was set in the PCIIS register.

Example

```
Uint32 x;
x = PCI_intTest(PCI_EVT_MASTEROK);
```

PCI_pwrStatTest *Tests if DSP has changed state*

Function Uint32 PCI_pwrStatTest();

Arguments none

Return Value Returns the following value if state change has occurred:

- ❏ 0:No State change request
- ❏ 1: Requested State D0/D1
- ❏ 2: Requested State D2
- ❏ 3: Requested State D3

Description Tests if the DSP has received an event related to a state change (not supported by 64x devices).

Example

```
PCI_pwrStatTest();
```

PCI_pwrStatUpdate *Updates current state of power management*

Function void PCI_pwrStatUpdate();

Arguments none

Return Value none

Description Updates the current state field of the PWDSRC register with the request state field value (not supported by 64x devices).

Example

```
PCI_pwrStatUpdate();
```

PCI_SUPPORT *Compile-time constant*

Constant PCI_SUPPORT

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

Currently, all devices support this module.

Example

```
#if (PCI_SUPPORT)
  /* user PCI configuration */
#endif
```

PCI_xfrByteCntSet *Sets number of bytes to be transferred*

Function

```
void PCI_xfrByteCntSet(
Uint16 nbbyte
);
```

Arguments

nbbyte	Number of bytes to be transferred for the next transaction. 1 < nbbyte < 65K max

Return Value

0 or 1

Description

Sets the number of bytes to transfer.

Example

```
PCI_xfrByteCntSet(0xFFFF); /* maximum of bytes */
```

PCI_xfrConfig *Sets up registers related to master transfer using config structure*

Function

```
void PCI_xfrConfig(
PCI_ConfigXfr *config
);
```

Arguments

config	Pointer to an initialized configuration structure.

Return Value

none

Description

Sets up the PCI registers related to the master transfer using the configuration structure. The values of the structure are written to the PCI registers. See also PCI_xfrConfigArgs() and PCI_ConfigXfr.

Example

```
PCI_ConfigXfr myXfrConfig = {
0x80000000,  /* dspma reg location data (src or dst)*/
0xFBE8000,  /* pcima reg CPLD XBISA */
0x0004000    /* pcimc register  */
);
PCI_xfrConfig(&myXfrConfig);
```

PCI_xfrConfigArgs *Sets up registers related to master transfer using register values*

Function void PCI_xfrConfigArgs(
Uint32 dspma,
Uint32 pcima,
Uint32 pcimc
);

Arguments

dspma	DSP master address register value.
pcima	PCI master address register value.
pcimc	PCI master control register value.

Return Value none

Description Sets up the PCI registers related to the master transfer using the register values passed in. The register values are written to the PCI registers. See also PCI_xfrConfig().

Example

```
PCI_xfrConfigArgs{
0x80001000,  /* dspma register */
0xFBE00000,  /* pcima register CPLD XBISA DSP reg*/
0x0100000    /* pcimc register 256-byte transfer*/
);
```

PCI_xfrEnable *Enables internal transfer request to auxiliary DMA channel*

Function void PCI_xfrEnable();

Arguments none

Return Value none

Description Enables the internal transfer request to the auxilliary DMA channel by clearing the HALT bit field of the HALT register (C620x/C670x only, not supported by 64x devices).

Example

```
PCI_xfrEnable();
```

PCI_xfrFlush — *Flushes current transaction*

Function void PCI_xfrFlush();

Arguments none

Return Value none

Description Flushes the current transaction. The transfer will stop and the FIFOs will be flushed.

Example

```
PCI_xfrFlush();
```

PCI_xfrGetConfig — *Reads configuration by returning values through config structure*

Function

```
void PCI_xfrGetConfig(
PCI_ConfigXfr *config
);
```

Arguments config Pointer to the returned configuration structure.

Return Value none

Description Reads the current PCI configuration by returning values through the configuration structure. The values of the PCI register are written to the configuration structure. See also PCI_ConfigXfr.

Example

```
PCI_ConfigXfr myXfrConfig;
PCI_xfrGetConfig(&myXfrConfig);
```

PCI_xfrHalt — *Terminates internal transfer requests to auxilliary DMA channel*

Function void PCI_xfrHalt();

Arguments none

Return Value none

Description Halts the internal transfer requests to the auxilliary DMA channel by setting the HALT bit field of the HALT register (C620x/C670x only, not supported by 64x devices).

Example

```
PCI_xfrHalt();
```

PCI_xfrStart — *Starts transaction*

Function

```
void PCI_xfrStart(
Uint32 modeXfr
);
```

Arguments

modeXfr — Specified one of the following transfer modes (macros):

- ❏ PCI_WRITE or 0x1
- ❏ PCI_READ_PREF or 0x2
- ❏ PCI_READ_NOPREF or 0x3

Return Value

none

Description

Starts the specified transaction.

Example

```
PCI_xfrStart(PCI_WRITE);
```

PCI_xfrTest — *Tests if current transaction is complete*

Function

Uint32 PCI_xfrTest();

Arguments

none

Return Value

0 to 7

Description

Tests the status of the master transaction and returns one of the following status values:

- ❏ PCI_PCIMC_START_FLUSH: Transaction not started/flush current transaction
- ❏ PCI_PCIMC_START_WRITE: Start a master write transaction
- ❏ PCI_PCIMC_START_READPREF: Start a master read transaction to prefetchable memory
- ❏ PCI_PCIMC_START_READNOPREF: Start a master read transaction to nonprefetchable memory
- ❏ PCI_PCIMC_START_CONFIGWRITE: Start a configuration write
- ❏ PCI_PCIMC_START_CONFIGREAD: Start a configuration read
- ❏ PCI_PCIMC_START_IOWRITE: Start an I/O write
- ❏ PCI_PCIMC_START_IOREAD: Start an I/O read

Example

```
PCI_xfrTest();
```

Chapter 17

PLL Module

This chapter describes the PLL module, lists the API functions and macros within the module, discusses the three application domains, and provides a PLL API reference section.

Topic **Page**

17.1 Overview

This module provides functions and macros to configure the PLL controller. The PLL controller peripheral is in charge of controlling the DSP clock.

Table 17–1 lists the configuration structure for use with the PLL functions. Table 17–2 lists the functions and constants available in the CSL PLL module.

Table 17–1. PLL Configuration Structures

Syntax	Type	Description	See page ...
PLL_Config	S	Structure used to configure the PLL controller	17-6
PLL_Init	S	Structure used to initialize the PLL controller	17-6

Table 17–2. PLL APIs

Syntax	Type	Description	See page ...
PLL_bypass	F	Sets the PLL in bypass mode	17-7
PLL_clkTest	F	Checks and returns the oscillator input stable condition	17-7
PLL_config	F	Configures the PLL using the configuration structure	17-8
PLL_configArgs	F	Configures the PLL using register fields as arguments	17-8
PLL_deassert	F	Releases the PLL from reset	17-9
PLL_disableOscDiv	F	Disables the oscillator divider OD1	17-9
PLL_disablePllDiv	F	Disables the specified divider	17-9
PLL_enable	F	Enables the PLL	17-10
PLL_enableOscDiv	F	Enables the oscillator divider OD1	17-10
PLL_enablePllDiv	F	Enables the specified divider	17-10
PLL_getConfig	F	Reads the current PLL controller configuration values	17-11
PLL_getMultiplier	F	Returns the PLL multiplier value	17-11
PLL_getOscRatio	F	Returns the oscillator divide ratio	17-12
PLL_getPllRatio	F	Returns the PLL divide ratio	17-12
PLL_init	F	Initializes the PLL using the `PLL_Init` structure	17-12
PLL_operational	F	Sets the PLL in operational mode	17-13

Note: F = Function; C = Constant

Table 17–2. PLL APIs (Continued)

Syntax	Type	Description	See page ...
PLL_pwrdwn	F	Sets the PLL in power down state	17-13
PLL_reset	F	Resets the PLL	17-14
PLL_setMultiplier	F	Sets the PLL multiplier value	17-14
PLL_setOscRatio	F	Sets the oscillator divide ratio (CLKOUT3 divider)	17-14
PLL_setPllRatio	F	Sets the PLL divide ratio	17-15
PLL_SUPPORT	C	A compile time constant whose value is 1 if the device supports PLL	17-16

Note: F = Function; C = Constant

17.1.1 Using the PLL Controller

The PLL controller can be used by passing an initialized `PLL_Config` structure to `PLL_config()` or by passing register values to the `PLL_configArgs()` function. To assist in creating register values, the _RMK(make) macros construct register values based on field values. In addition, the symbol constants may be used for the field values.

The PLL can also be initialized based on parameters by passing a `PLL_Init` structure to the `PLL_init()` function.

17.2 Macros

There are two types of PLL macros: those that access registers and fields, and those that construct register and field values.

Table 17–3 lists the PLL macros that access registers and fields, and Table 17–4 lists the PLL macros that construct register and field values. The macros themselves are found in Chapter 24, *Using the HAL Macros*.

PLL macros are not handle-based.

Table 17–3. PLL Macros that Access Registers and Fields

Macro	Description/Purpose	See page ...
PLL_ADDR(<REG>)	Register address	24-12
PLL_RGET(<REG>)	Returns the value in the peripheral register	24-18
PLL_RSET(<REG>,x)	Register set	24-20
PLL_FGET(<REG>,<FIELD>)	Returns the value of the specified field in the peripheral register	24-13
PLL_FSET(<REG>,<FIELD>,fieldval)	Writes *fieldval* to the specified field in the peripheral register	24-15
PLL_FSETS(<REG>,<FIELD>,<SYM>)	Writes the symbol value to the specified field in the peripheral	24-17
PLL_RGETA(addr,<REG>)	Gets register for a given address	24-19
PLL_RSETA(addr,<REG>,x)	Sets register for a given address	24-20
PLL_FGETA(addr,<REG>,<FIELD>)	Gets field for a given address	24-13
PLL_FSETA(addr,<REG>,<FIELD>, fieldval)	Sets field for a given address	24-16
PLL_FSETSA(addr,<REG>,<FIELD>, <SYM>)	Sets field symbolically for a given address	24-17

Table 17–4. PLL Macros that Construct Register and Field Values

Macro	Description/Purpose	See page ...
PLL_<REG>_DEFAULT	Register default value	24-21
PLL_<REG>_RMK()	Register make	24-23
PLL_<REG>_OF()	Register value of ...	24-22
PLL_<REG>_<FIELD>_DEFAULT	Field default value	24-24
PLL_FMK()	Field make	24-14
PLL_FMKS()	Field make symbolically	24-15
PLL_<REG>_<FIELD>_OF()	Field value of ...	24-24
PLL_<REG>_<FIELD>_<SYM>	Field symbolic value	24-24

17.3 Configuration Structures

PLL_Config *Structure used to configure the PLL controller*

Structure PLL_Config

Members

Uint32	pllcsr	PLL control/status register
Uint32	pllm	PLL multiplier control register
Uint32	plldiv0	PLL controller divider 0 register
Uint32	plldiv1	PLL controller divider 1 register
Uint32	plldiv2	PLL controller divider 2 register
Uint32	plldiv3	PLL controller divider 3 register
Uint32	oscdiv1	Oscillator divider 1 register

Description This is the PLL configuration structure used to configure the PLL controller. The user should create and initialize this structure before passing its address to the `PLL_config()` function.

PLL_Init *Structure used to initialize the PLL controller*

Structure PLL_Init

Members

Uint32	mdiv	PLL multiplier
Uint32	d0ratio	PLL divider 0 ratio
Uint32	d1ratio	PLL divider 1 ratio
Uint32	d2ratio	PLL divider 2 ratio
Uint32	d3ratio	PLL divider 3 ratio
Uint32	od1ratio	Oscillator divider 1 ratio

Description This is the PLL initialization structure used to initialize the PLL controller. The user should create and initialize this structure before passing its address to the `PLL_init()` function.

17.4 Functions

PLL_bypass — *Sets the PLL in bypass mode*

Function

```
void PLL_bypass(
    void
);
```

Arguments none

Return Value none

Description This function sets the PLL in bypass mode wherein Divider D0 and PLL are bypassed. SYSCLK1 to SYSCLK3 are divided down directly from the input reference clock.

Example

```
PLL_bypass();
```

PLL_clkTest — *Checks and returns the oscillator input stable condition*

Function

```
void Uint32  PLL_clkTest(
    void
);
```

Arguments none

Return Value Uint32 Oscillator condition

- ❑ 0 – Not stable
- ❑ 1 – Stable

Description This function checks and returns the oscillator input stable condition.

0 – OSCIN/CLKIN input not yet stable. This is true if the synchronous counter has not finished counting.

1 – OSCIN/CLKIN input is stable. This is true if any one of the following three cases is true:

- Synchronous counter has finished counting the number of OSCIN/CLKIN cycles
- Synchronous counter is disabled
- Test mode

Example

```
Uint32 val;
val = PLL_clkTest();
```

PLL_config — *Configures the PLL using the configuration structure*

Function

```
void PLL_config(
    PLL_Config *myConfig
);
```

Arguments

myConfig Pointer to the configuration structure

Return Value

none

Description

This function configures the PLL controller using the configuration structure. The values of the structure variables are written to the PLL controller registers.

Example

```
PLL_Config MyConfig
...
PLL_config(&MyConfig);
```

PLL_configArgs — *Configures the PLL controller using register fields as arguments*

Function

```
void    PLL_configArgs(
    Uint32      pllcsr,
    Uint32      pllm,
    Uint32      plldiv0,
    Uint32      plldiv1,
    Uint32      plldiv2,
    Uint32      plldiv3,
    Uint32      oscdiv1
)
```

Arguments

pllcsr PLL control/status register

pllm PLL multiplier control register

plldiv0 PLL controller divider 0 register

plldiv1 PLL controller divider 1 register

plldiv2 PLL controller divider 2 register

plldiv3 PLL controller divider 3 register

oscdiv1 Oscillator divider 1 register

Return Value

none

Description This function configures the PLL controller as per the register field values given.

Example
```
PLL_configArgs(0x8000,0x01,0x800A,0x800B,0x800C,0x800D,0x0009);
```

PLL_deassert *Releases the PLL from reset*

Function
```
void PLL_deassert(
    void
);
```

Arguments none

Return Value none

Description This function releases the PLL from reset.

Example
```
PLL_deassert();
```

PLL_disableOscDiv *Disables the oscillator divider OD1*

Function
```
void PLL_disableOscDiv(
    void
);
```

Arguments none

Return Value none

Description This function disables the oscillator divider OD1.

Example
```
PLL_disableOscDiv();
```

PLL_disablePllDiv *Disables the specified divider*

Function
```
void PLL_disablePllDiv(
    Uint32    divId
);
```

Arguments divId Divider ID

- ❏ PLL_DIV0 – Divider 0
- ❏ PLL_DIV1 – Divider 1

- PLL_DIV2 – Divider 2
- PLL_DIV3 – Divider 3

Return Value none

Description This function disables the divider specified by the 'divId' parameter.

Example
```
PLL_disablePllDiv(PLL_DIV0);
```

PLL_enable *Enables the PLL*

Function
```
void PLL_enable(
    void
);
```

Arguments none

Return Value none

Description This function enables the PLL and sets it in 'PLL' mode.
Note that here divider D0 is not bypassed. SYSCLK1 to SYSCLK3 are divided down directly from the input reference clock.

Example
```
PLL_enable();
```

PLL_enableOscDiv *Enables the oscillator divider OD1*

Function
```
void PLL_enableOscDiv(
    void
);
```

Arguments none

Return Value none

Description This function enables the oscillator divider OD1.

Example
```
PLL_enableOscDiv();
```

PLL_enablePllDiv *Enables the specified divider*

Function
```
void PLL_enablePllDiv(
    Uint32    divId
);
```

Arguments divId Divider ID

- PLL_DIV0 – Divider 0
- PLL_DIV1 – Divider 1
- PLL_DIV2 – Divider 2
- PLL_DIV3 – Divider 3

Return Value none

Description This function enables the divider specified by the 'divId' parameter.

Example
```
PLL_enablePllDiv(PLL_DIV0);
```

PLL_getConfig *Reads the current PLL controller configuration values*

Function
```
void PLL_getConfig(
    PLL_Config  *myConfig
);
```

Arguments myConfig Pointer to the configuration structure

Return Value none

Description This function gets the current PLL configuration values.

Example
```
PLL_Config pllCfg;
...
PLL_getConfig(&pllCfg);
```

PLL_getMultiplier *Returns the PLL multiplier value*

Function
```
Uint32 PLL_getMultiplier(
    void
);
```

Arguments none

Return Value Uint32 PLL multiplier value. See `PLL_setMultiplier()`.

Description This function gets the current PLL multiplier value. For PLL multiplier values, see `PLL_setMultiplier()`.

Example
```
Uint32 val;
val = PLL_getMultiplier();
```

PLL_getOscRatio — *Returns the oscillator divide ratio*

Function

```
Uint32 PLL_getOscRatio(
    void
);
```

Arguments none

Return Value Uint32 Oscillator divide ratio. See `PLL_setOscRatio()`.

Description This function returns the oscillator divide ratio. For oscillator divide values, see `PLL_setOscRatio()`.

Example

```
Uint32 val;
val = PLL_getOscRatio();
```

PLL_getPllRatio — *Returns the PLL divide ratio*

Function

```
Uint32 PLL_getPllcRatio(
    void
);
```

Arguments divId PLL divider ID. See `PLL_setPllRatio()`.

Return Value Uint32 PLL divide ratio. See `PLL_setPllRatio()`.

Description This function returns the PLL divide ratio. For PLL divide values, see `PLL_setPllRatio()`.

Example

```
Uint32 val;
val = PLL_getPllRatio(PLL_DIV0);
```

PLL_init — *Initialize PLL using PLL_Init structure*

Function

```
void PLL_init(
    PLL_Init   *myInit
);
```

Arguments myInit Pointer to the initialization structure.

Return Value none

Description This function initializes the PLL controller using the PLL_Init structure. The values of the structure variables are written to the corresponding PLL controller register fields.

Example

```
PLL_Init myInit;
...
PLL_init(&myInit);
```

PLL_operational *Sets PLL in operational mode*

Function

```
void PLL_operational(
    void
);
```

Arguments none

Return Value none

Description This function sets the PLL in operational mode. See `PLL_pwrdwn()`. This function enables the PLL and Divider 0 path.

Example

```
PLL_operational();
```

PLL_pwrdwn *Sets the PLL in power down mode*

Function

```
void PLL_pwrdwn(
    void
);
```

Arguments none

Return Value none

Description This function sets the PLL in power down state. Divider D0 and the PLL are bypassed. SYSCLK1 to SYSCLK3 are divided down directly from the input reference clock.

Example

```
PLL_prwdwn();
```

PLL_reset — *Resets the PLL device*

Function

```
void PLL_reset(
    void
);
```

Arguments none

Return Value none

Description This function asserts reset to the PLL.

Example

```
PLL_reset();
```

PLL_setMultiplier — *Sets the PLL multiplier value*

Function

```
void PLL_setMultiplier(
    Uint32    val
);
```

Arguments val Multiplier select

Return Value none

Description This function sets the PLL multiplier value.

PLL multiplier select

00000 = x1	00001=x2	00010=x3	00011=x4,
00000 = x5	00001=x6	00010=x7	00011=x8,
00000 = x9	00001=x10	00010=x11	00011=x12,
00000 = x13	00001=x14	00010=x15	00011=x16
00000 = x17	00001=x18	00010=x19	00011=x20,
00000 = x21	00001=x22	00010=x23	00011=x24,
00000 = x25	00001=x26	00010=x28	00011=x28,
00000 = x29	00001=x30	00010=x31	00011=Not Supported

Example

```
PLL_setMultiplier(0x04);
```

PLL_setOscRatio — *Sets the oscillator divide ratio (CLKOUT3 divider)*

Function

```
void PLL_setOscRatio(
    Uint32    val
);
```

Arguments val Divider values

Return Value none

Description This function sets the oscillator divide ratio (CLKOUT3 divider).

Divider values

00000 = /1	00001=/2	00010=/3	00011=/4,
00000 = /5	00001=/6	00010=/7	00011=/8,
00000 = /9	00001=/10	00010=/11	00011=/12,
00000 = /13	00001=/14	00010=/15	00011=/16
00000 = /17	00001=/18	00010=/19	00011=/20,
00000 = /21	00001=/22	00010=/23	00011=/24,
00000 = /25	00001=/26	00010=/28	00011=/28,
00000 = /29	00001=/30	00010=/31	00011=/32

Example

```
PLL_setOscRatio(0x05);
```

PLL_setPllRatio *Sets the PLL divide ratio*

Function

```
void PLL_setPllDiv(
    Uint32    divId,
    Uint32    val
);
```

Arguments

divId Divider ID

- ❏ PLL_DIV0 – Divider 0
- ❏ PLL_DIV1 – Divider 1
- ❏ PLL_DIV2 – Divider 2
- ❏ PLL_DIV3 – Divider 3

val Divider values

Return Value none

Description This function sets the divide ratio for the clock divider specified by the ’divId’ parameter.

PLL_SUPPORT

Description This function sets the divide ratio for the clock divider specified by the ’divId’ parameter.

Divider values

00000 = /1	00001=/2	00010=/3	00011=/4,
00000 = /5	00001=/6	00010=/7	00011=/8,
00000 = /9	00001=/10	00010=/11	00011=/12,
00000 = /13	00001=/14	00010=/15	00011=/16
00000 = /17	00001=/18	00010=/19	00011=/20,
00000 = /21	00001=/22	00010=/23	00011=/24,
00000 = /25	00001=/26	00010=/28	00011=/28,
00000 = /29	00001=/30	00010=/31	00011=/32

Example

```
PLL_setPllDiv(PLL_DIV0,0x05);
```

PLL_SUPPORT *Compile time constant*

Constant PLL_SUPPORT

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

Currently, only the C6713 device supports this module.

Example

```
#if (PLL_SUPPORT)
  /* user PLL configuration */
#endif
```

Chapter 18

PWR Module

This chapter describes the PWR module, lists the API functions and macros within the module, and provides a PWR API reference section.

Topic **Page**

18.1 Overview

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

Table 18–1 lists the configuration structure for use with the PWR functions. Table 18–2 lists the functions and constants available in the CSL PWR module.

Table 18–1. PWR Configuration Structure

Syntax	Purpose	See page ...
PWR_Config	Structure used to set up the PWR options	18-5

Table 18–2. PWR APIs

Syntax	Type	Description	See page ...
PWR_config	F	Sets up the PWR register using the configuration structure	18-6
PWR_configArgs	F	Sets up the power-down logic using the register value passed in	18-6
PWR_getConfig	F	Reads the current PWR configuration values	18-7
PWR_powerDown	F	Forces the DSP to enter a power-down state	18-7
PWR_SUPPORT	C	A compile time constant whose value is 1 if the device supports the PWR module	18-7

Note: F = Function; C = Constant

18.2 Macros

There are two types of PWR macros: those that access registers and fields, and those that construct register and field values.

Table 18–3 lists the PWR macros that access registers and fields, and Table 18–4 lists the PWR macros that construct register and field values. The macros themselves are found in Chapter 24, *Using the HAL Macros*.

PWR macros are not handle-based.

Table 18–3. PWR Macros that Access Registers and Fields

Macro	Description/Purpose	See page ...
PWR_ADDR(<REG>)	Register address	24-12
PWR_RGET(<REG>)	Returns the value in the peripheral register	24-18
PWR_RSET(<REG>,x)	Register set	24-20
PWR_FGET(<REG>,<FIELD>)	Returns the value of the specified field in the peripheral register	24-13
PWR_FSET(<REG>,<FIELD>,fieldval)	Writes *fieldval* to the specified field in the peripheral register	24-15
PWR_FSETS(<REG>,<FIELD>,<SYM>)	Writes the symbol value to the specified field in the peripheral	24-17
PWR_RGETA(addr,<REG>)	Gets register for a given address	24-19
PWR_RSETA(addr,<REG>,x)	Sets register for a given address	24-20
PWR_FGETA(addr,<REG>,<FIELD>)	Gets field for a given address	24-13
PWR_FSETA(addr,<REG>,<FIELD>, fieldval)	Sets field for a given address	24-16
PWR_FSETSA(addr,<REG>,<FIELD>, <SYM>)	Sets field symbolically for a given address	24-17

Table 18–4. PWR Macros that Construct Register and Field Values

Macro	Description/Purpose	See page ...
PWR_<REG>_DEFAULT	Register default value	24-21
PWR_<REG>_RMK()	Register make	24-23
PWR_<REG>_OF()	Register value of ...	24-22
PWR_<REG>_<FIELD>_DEFAULT	Field default value	24-24
PWR_FMK()	Field make	24-14
PWR_FMKS()	Field make symbolically	24-15
PWR_<REG>_<FIELD>_OF()	Field value of ...	24-24
PWR_<REG>_<FIELD>_<SYM>	Field symbolic value	24-24

18.3 Configuration Structure

PWR_Config *Structure used to set up the PWR options*

Structure PWR_Config

Members Uint32 pdctl Power-down control register (6202 and 6203 devices)

Description This is the PWR configuration structure used to set up the PWR option of the 6202 device. You create and initialize this structure and then pass its address to the `PWR_config()` function. You can use literal values or the *_RMK* macros to create the structure member values.

Example

```
PWR_Config pwrCfg = {
  0x00000000
};
…
PWR_config(&pwrCfg);
```

18.4 Functions

PWR_config — *Sets up the PWR register using the configuration structure*

Function

```
void PWR_config(
  PWR_Config *config
);
```

Arguments

config	Pointer to a configuration structure.

Return Value

none

Description

Sets up the PWR register using the configuration structure.

Example

```
PWR_Config pwrCfg = {
  0x00000000
};
PWR_config(&pwrCfg);
```

PWR_configArgs — *Sets up power-down logic using register value passed in*

Function

```
void PWR_configArgs(
  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_PDCTL_RMK* macro to create the register value based on field values.

Example

```
PWR_configArgs(0x00000000);
```

PWR_getConfig *Reads the current PWR configuration values*

Function

```
void PWR_config(
  PWR_Config *config
);
```

Arguments

config Pointer to a configuration structure.

Return Value

none

Description

Gets PWR current configuration value.

Example

```
PWR_Config pwrCfg;
PWR_getConfig(&pwrCfg);
```

PWR_powerDown *Forces DSP to enter 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* (SPRU190) for a description of the power-down modes.

Example

```
PWR_powerDown(PWR_PD2);
```

PWR_SUPPORT *Compile-time constant*

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.

PWR_SUPPORT

Currently, all devices support this module.

Example

```
#if (PWR_SUPPORT)
  /* user PWR configuration /
#endif
```

Chapter 19

TCP Module

This chapter describes the TCP module, lists the API functions and macros within the module, discusses how to use the TCP, and provides a TCP API reference section.

Topic **Page**

19.1 Overview

Currently, there is one TMS320C6000™ device with a turbo coprocessor (TCP): the TMS320C6416. The TCP is intended to be serviced using the EDMA for most accesses, but the CPU must first configure the TCP control values. There are also a number of functions available to the CPU to monitor the TCP status and access decision and output parameter data.

Table 19–1 lists the configuration structures for use with the TCP functions. Table 19–2 lists the functions and constants available in the CSL TCP module.

Table 19–1. TCP Configuration Structures

Syntax	Type	Description	See page ...
TCP_BaseParams	S	Structure used to set basic TCP parameters	19-8
TCP_ConfigIc	S	Structure containing the IC register values	19-9
TCP_Params	S	Structure containing all channel characteristics	19-10

Table 19–2. TCP APIs

Syntax	Type	Description	See page ...
TCP_calcSubBlocksSA	F	Calculates the sub-blocks within a frame for standalone mode	19-13
TCP_calcSubBlocksSP	F	Calculates the sub-frames and -blocks within a frame for shared processing mode	19-13
TCP_calcCountsSA	F	Calculates the number of elements for each data buffer to be transmitted to/from the TCP using the EDMA for standalone mode	19-13
TCP_calcCountsSP	F	Calculates the number of elements for each data buffer to be transmitted to/from the TCP using the EDMA for shared processing mode	19-13
TCP_calculateHd	F	Calculates hard decisions for shared processing mode	19-14
TCP_ceil	F	Ceiling function	19-14
TCP_deinterleaveExt	F	De-interleaves extrinsics data for shared processing mode	19-15
TCP_demuxInput	F	Demultiplexes input into two working data sets for shared processing mode	19-16
TCP_END_NATIVE	C	Value indicating native endian format	19-16

Note: F = Function; C = Constant

Table 19–2. TCP APIs (Continued)

Syntax	Type	Description	See page ...
TCP_END_PACKED32	C	Value indicating little endian format within packed 32-bit words	19-16
TCP_errTest	F	Returns the error bit of ERR register	19-17
TCP_FLEN_MAX	F	Maximum frame length	19-17
TCP_genIc	F	Generates the `TCP_ConfigIc` struct based on the TCP parameters provided by the `TCP_Params` struct	19-17
TCP_genParams	F	Function used to set basic TCP parameters	19-18
TCP_getAccessErr	F	Returns access error flag	19-18
TCP_getAprioriEndian	F	Returns Apriori data endian configuration	19-18
TCP_getExtEndian	F	Returns the Extrinsics data endian configuration	19-19
TCP_getFrameLenErr	C	Returns the frame length error status	19-20
TCP_getIcConfig	F	Returns the IC values already programmed into the TCP	19-20
TCP_getInterEndian	F	Returns the Interleaver Table data endian configuration	19-20
TCP_getInterleaveErr	F	Returns the interleaver table error status	19-21
TCP_getLastRelLenErr	F	Returns the error status for a bad reliability length	19-21
TCP_getModeErr	F	Returns the error status for a bad TCP mode	19-22
TCP_getNumIt	F	Returns the number of iterations performed by the TCP	19-22
TCP_getOutParmErr	F	Returns the output parameters error status	19-22
TCP_getProlLenErr	F	Returns the error status for an invalid prolog length	19-23
TCP_getRateErr	F	Returns the error status for an invalid rate	19-23
TCP_getRelLenErr	F	Returns the error status for an invalid reliability length	19-23
TCP_getSubFrameErr	F	Returns the error status indicating an invalid number of sub frames	19-24
TCP_getSysParEndian	F	Returns the Systematics and Parities data endian configuration	19-24
TCP_icConfig	F	Stores the IC values into the TCP	19-25
TCP_icConfigArgs	F	Stores the IC values into the TCP using arguments	19-25

Note: F = Function; C = Constant

Table 19–2. TCP APIs (Continued)

Syntax	Type	Description	See page ...
TCP_interleaveExt	F	Interleaves extrinsics data for shared processing mode	19-27
TCP_makeTailArgs	F	Builds the Tail values used for `IC6–IC11`	19-27
TCP_MAP_MAP1A	C	Value indicating that the first iteration of a MAP1 decoding	19-28
TCP_MAP_MAP1B	C	Value indicating a MAP1 decoding (any iteration after the first)	19-28
TCP_MAP_MAP2	C	Value indicating a MAP2 decoding	19-28
TCP_MODE_SA	C	Value indicating standalone processing mode	19-28
TCP_MODE_SP	C	Value indicating shared processing mode	19-28
TCP_normalCeil	F	Normalized ceiling function	19-29
TCP_pause	F	Pauses the TCP	19-29
TCP_RATE_1_2	C	Value indicating a rate of 1/2	19-29
TCP_RATE_1_3	C	Value indicating a rate of 1/3	19-29
TCP_RATE_1_4	C	Value indicating a rate of 1/4	19-29
TCP_RLEN_MAX	C	Maximum reliability length	19-30
TCP_setAprioriEndian	F	Sets the Apriori data endian configuration	19-30
TCP_setExtEndian	F	Sets the Extrinsics data endian configuration	19-30
TCP_setInterEndian	F	Sets the Interleaver Table data endian configuration	19-31
TCP_setNativeEndian	F	Sets all data formats to be native (not packed data)	19-31
TCP_setPacked32Endian	F	Sets all data formats to be packed data	19-32
TCP_setParams	F	Generates `IC0–IC5` based on the channel parameters	19-32
TCP_setSysParEndian	F	Sets the Systematics and Parities data endian configuration	19-33
TCP_STANDARD_3GPP	C	Value indicating the 3GPP standard	19-33
TCP_STANDARD_IS2000	C	Value indicating the IS2000 standard	19-33
TCP_start	F	Starts the TCP	19-33
TCP_statError	F	Returns the error status	19-34

Note: F = Function; C = Constant

Table 19–2. TCP APIs (Continued)

Syntax	Type	Description	See page ...
TCP_statPause	F	Returns the pause status	19-34
TCP_statRun	F	Returns the run status	19-34
TCP_statWaitApriori	F	Returns the Apriori data status	19-35
TCP_statWaitExt	F	Returns the Extrinsics data status	19-35
TCP_statWaitHardDec	F	Returns the Hard Decisions status	19-35
TCP_statWaitIc	F	Returns the IC values status	19-36
TCP_statWaitInter	F	Returns the Interleaver Table status	19-36
TCP_statWaitOutParm	F	Returns the Output Parameters status	19-36
TCP_statWaitSysPar	F	Returns the Systematics and Parities data status	19-37
TCP_tailConfig	F	Generates `IC6–IC11` by calling either `TCP_tailConfig3GPP` or `TCP_tailConfigIS2000`	19-37
TCP_tailConfig3GPP	F	Generates tail values for 3GPP channel data	19-38
TCP_tailConfigIs2000	F	Generates tail values for IS2000 channel data	19-39
TCP_unpause	F	Unpauses the TCP	19-40

Note: F = Function; C = Constant

19.1.1 Using the TCP

To use the TCP, you must first configure the control values, or IC values, that will be sent via the EDMA to program its operation. To do this, the `TCP_Params` structure and `TCP_XabData` pointer are passed to `TCP_icConfig()`. `TCP_Params` contains all of the channel characteristics and `TCP_XabData` is a pointer to the tail data located at the end of the received channel data. This configuration function returns a pointer to the IC values that are to be sent using the EDMA. If desired, the configuration function can be bypassed and the user can generate each IC value independently, using several *TCP_RMK* (make) macros that construct register values based on field values. In addition, the symbol constants may be used for the field values.

When operating in big endian mode, the CPU must configure the format of all the data to be transferred to and from the TCP. This is accomplished by programming the TCP Endian register (`TCP_END`). Typically, the data will all be of the same format, either following the native element size (either 8-bit or 16–bit) or being packed into a 32-bit word. This being the case, the endian

mode values can be set using a single function call to either `TCP_setNativeEndian()` or `TCP_setPacked32Endian()`. Alternatively, the data format of individual data types can be programmed with independent functions.

The user can monitor the status of the TCP during operation and also monitor error flags if there is a problem.

19.2 Macros

There are two types of TCP macros: those that access registers and fields, and those that construct register and field values. These are not required as all TCP configuring and monitoring can be done through the provided functions. These TCP functions make use of a number of macros.

Table 19–3 lists the TCP macros that access registers and fields. Table 19–4 lists the TCP macros that construct register and field values. The macros themselves are found in Chapter 24, *Using the HAL Macros*.

The TCP module includes handle-based macros.

Table 19–3. TCP Macros that Access Registers and Fields

Macro	Description/Purpose	See page ...
TCP_ADDR(<REG>)	Register address	24-12
TCP_RGET(<REG>)	Returns the value in the peripheral register	24-18
TCP_RSET(<REG>,x)	Register set	24-20
TCP_FGET(<REG>,<FIELD>)	Returns the value of the specified field in the peripheral register	24-13
TCP_FSET(<REG>,<FIELD>,fieldval)	Writes *fieldval* to the specified field in the peripheral register	24-15
TCP_FSETS(<REG>,<FIELD>,<SYM>)	Writes the symbol value to the specified field in the peripheral	24-17
TCP_RGETA(addr,<REG>)	Gets register for a given address	24-19
TCP_RSETA(addr,<REG>,x)	Sets register for a given address	24-20
TCP_FGETA(addr,<REG>,<FIELD>)	Gets field for a given address	24-13
TCP_FSETA(addr,<REG>,<FIELD>, fieldval)	Sets field for a given address	24-16
TCP_FSETSA(addr,<REG>,<FIELD>, <SYM>)	Sets field symbolically for a given address	24-17

Table 19–4. TCP Macros that Construct Register and Field Values

Macro	Description/Purpose	See page ...
TCP_<REG>_DEFAULT	Register default value	24-21
TCP_<REG>_RMK()	Register make	24-23
TCP_<REG>_OF()	Register value of ...	24-22
TCP_<REG>_<FIELD>_DEFAULT	Field default value	24-24
TCP_FMK()	Field make	24-14
TCP_FMKS()	Field make symbolically	24-15
TCP_<REG>_<FIELD>_OF()	Field value of ...	24-24
TCP_<REG>_<FIELD>_<SYM>	Field symbolic value	24-24

19.3 Configuration Structures

TCP_BaseParams *Structure used to set basic TCP Parameters*

Structure TCP_BaseParams

Members

TCP_Standard	standard; TCP Decoded Standard3GPP/IS2000
TCP_Rate	rate; Code rate
Uint16	frameLen; Frame Length
Uint8	prologSize; Prolog Size
Uint8	maxIter; Maximum Iteration
Uint8	snr; SNR Threshold
Uint8	intFlag; Interleaver flag
Uint8	outParmFlag Output Parameter read flag

Description This is the TCP base parameters structure used to set up the TCP programmable parameters. You create the object and pass it to the TCP_genParams() function which returns the TCP_Params structure.

Example

```
TCP_BaseParams tcpBaseParam0 = {
    TCP_STANDARD_3GPP,  /* Decoder Standard  */
    TCP_RATE_1_3,        /* Rate  */
    40,   /*Frame Length (FL: 40 to 20730)*/
    24,   /*Prolog Size (P: 24 to 48)  */
     8,   /*Max of Iterations (MAXIT–SA mode only)*/
     0,   /*SNR Threshold (SNR – SA mode only) */
     1,   /*Interleaver Write Flag   */
     1    /*Output Parameters Read Flag   */
};
  …
TCP_genParams(&tcpBaseParam0, &tcpParam0);
```

TCP_ConfigIc

Structure containing the IC register values

Structure

```
typedef struct {
    Uint32 ic0;
    Uint32 ic1;
    Uint32 ic2;
    Uint32 ic3;
    Uint32 ic4;
    Uint32 ic5;
    Uint32 ic6;
    Uint32 ic7;
    Uint32 ic8;
    Uint32 ic9;
    Uint32 ic10;
    Uint32 ic11;
} TCP_ConfigIc;
```

Members

ic0	Input Configuration word 0 value
ic1	Input Configuration word 1 value
ic2	Input Configuration word 2 value
ic3	Input Configuration word 3 value
ic4	Input Configuration word 4 value
ic5	Input Configuration word 5 value
ic6	Input Configuration word 6 value
ic7	Input Configuration word 7 value
ic8	Input Configuration word 8 value
ic9	Input Configuration word 9 value
ic10	Input Configuration word 10 value
ic11	Input Configuration word 11 value

Description

This is the TCP input configuration structure that holds all of the configuration values that are to be transferred to the TCP via the EDMA. Though using the EDMA is highly recommended, the values can be written to the TCP using the CPU with the `TCP_icConfig()` function.

Example

```
extern TCP_Params *params;
extern TCP_UserData *xabData;
TCP_ConfigIc *config;
...
TCP_genIc(params, xabData, config);
...
```

TCP_Params

Structure containing all channel characteristics

Structure

```
typedef struct {
    TCP_Standard standard;
    TCP_Mode     mode;
    TCP_Map      map;
    TCP_Rate     rate;
    Uint32       intFlag;
    Uint32       outParmFlag;
    Uint32       frameLen;
    Uint32       subFrameLen;
    Uint32       relLen;
    Uint32       relLenLast;
    Uint32       prologSize;
    Uint32       numSubBlock;
    Uint32       numSubBlockLast;
    Uint32       maxIter;
    Uint32       snr;
    Uint32       numInter;
    Uint32       numSysPar;
    Uint32       numApriori;
    Uint32       numExt;
    Uint32       numHd;
} TCP_Params;
```

Members

standard	The 3G standard used: 3GPP or IS2000 The available constants are: ❏ TCP_STANDARD_3GPP ❏ TCP_STANDARD_IS2000
mode	The processing mode: Shared or Standalone The available constants are: ❏ TCP_MODE_SA ❏ TCP_MODE_SP
map	The map mode constants are: ❏ TCP_MAP_MAP1A ❏ TCP_MAP_MAP1B ❏ TCP_MAP_MAP2
rate	The rate: 1/2, 1/3,1/4 The rate constants are: ❏ TCP_RATE_1_2 ❏ TCP_RATE_1_3 ❏ TCP_RATE_1_4
intFlag	Interleaver write flag
outParmFlag	Output parameters flag
frameLen	Number of symbols in the frame to be decoded
subFrameLen	The number of symbols in a sub-frame
relLen	Reliability length
relLenLast	Reliability length of the last sub-frame
prologSize	Prolog Size
numSubBlock	Number of sub-blocks
numSubBlockLast	Number of sub-blocks in the last sub-frame
maxIter	Maximum number of iterations
snr	Signal to noise ratio threshold
numInter	Number of interleaver words per event
numSysPar	Number of systematics and parities words per event
numApriori	Number of apriori words per event
numExt	Number of extrinsics per event
numHd	Number of hard decisions words per event

TCP_Params

Description

This is the TCP parameters structure that holds all of the information concerning the user channel. These values are used to generate the appropriate input configuration values for the TCP and to program the EDMA.

Example

```
extern TCP_Params *params;
extern TCP_UserData *xabData;
TCP_ConfigIc *config;
...
TCP_genIc(params, xabData, config);
...
```

19.4 Functions

TCP_calcSubBlocksSA *Calculates sub-blocks for standalone processing*

Function void TCP_calcSubBlocksSA(TCP_Params *configParms);

Arguments ConfigParms Configuration parameters

Return Value none

Description Divides the data frames into sub-blocks for standalone processing mode.

Example
```
TCP_calcSubBlocksSA(configParms);
```

TCP_calcSubBlocksSP *Calculates sub-blocks for shared processing*

Function Uint32 TCP_calcSubBlocksSP(TCP_Params *configParms;

Arguments ConfigParms Configuration parameters

Return Value numSubFrames Number of sub-frames

Description Divides the data frames into sub-frames and sub-blocks for shared processing mode. The number of subframes into which the data frame was divided is returned.

Example
```
Uint32 numSubFrames;
  NumSubFrames = TCP_calcSubBlocksSP(configParms);
```

TCP_calcCountsSA *Calculates the count values for standalone processing*

Function Void TCP_calcCountsSA(TCP_Params *configParms);

Arguments configParms Configuration parameters

Return Value none

Description This function calculates all of the count values required to transfer all data to/from the TCP using the EDMA. This function is for standalone processing mode.

Example
```
TCP_calcCountsSA(configParms);
```

TCP_calcCountsSP *Calculates the count values for shared processing*

Function Void TCP_calcCountsSP(TCP_Params *configParms);

Arguments configParms Configuration parameters

Return Value none

Description This function calculates all of the count values required to transfer all data to/from the TCP using the EDMA. This function is for shared processing mode.

Example

```
TCP_calcCountsSP(configParms);
```

TCP_calculateHd *Calculate hard decisions*

Function

```
void TCP_calculateHd(
    const TCP_ExtrinsicData *restrict extrinsicsMap1,
    const TCP_ExtrinsicData *restrict apriori,
    const TCP_UserData *restrict channel_data,
    Uint32 *restrict hardDecisions,
    Uint16 numExt,
    Uint8  rate);
```

Arguments

extrinsicsMap1	Extrinsics data following MAP1 decode
apriori	Apriori data following MAP2 decode
channel_data	Input channel data
harddecisions	Hard decisions
numext	Number of extrinsics
rate	Channel rate

Return Value none

Description This function calculates the hard decisions following multiple MAP decodings in shared processing mode.

Example

```
<...Iterate through MAP1 and MAP2 decodes...>
void TCP_calculateHd(extrinsicsMap1, apriori,
      channel_data, hardDecisions, numExt, rate);
```

TCP_ceil *Ceiling function*

Function Uint32 TCP_ceil(Uint32 val, Uint32 pwr2);

Arguments

val	Value to be augmented
pwr2	The power of two by which val must be divisible

Return Value

ceilVal	The smallest number which when multiplied by 2^pwr2 is greater than val.

Description This function calculates the ceiling for a given value and a power of 2. The arguments follow the formula: ceilVal * 2^pwr2 = ceiling(val, pwr2).

Example

```
numSysPar = TCP_ceil((frameLen * rate), 4);
```

TCP_deinterleaveExt *De-interleave extrinsics data*

Function

```
Void TCP_deinterleaveExt(
    TCP_ExtrinsicData *restrict aprioriMap1,
    const TCP_ExtrinsicData *restrict extrinsicsMap2,
    const Uint16 *restrict interleaverTable,
    Uint32  numExt);
```

Arguments

aprioriMap1	Apriori data for MAP1 decode
extrinsicsMap2	Extrinsics data following MAP2 decode
interleaverTable	Interleaver data table
numExt	Number of Extrinsics

Return Value none

Description This function de-interleaves the MAP2 extrinsics data to generate apriori data for the MAP1 decode. This function is for use in performing shared processing.

Example

```
<...MAP 2 decode...>
TCP_deinterleaveExt(aprioriMap2, extrinsicsMap1,
                    interleaverTable, numExt);
<...MAP 1 decode...>
```

TCP_demuxInput *Demultiplexes the input data*

Function

Void TCP_demuxInput(Uint32 rate,
Uint32 frameLen,
const TCP_UserData *restrict input,
const Uint16 *restrict interleaver,
TCP_ExtrinsicData *restrict nonInterleaved,
TCP_ExtrinsicData *restrict interleaved);

Arguments

rate	Channel rate
frameLen	Frame length
input	Input channel data
interleaver	Interleaver data table
nonInterleaved	Non-interleaved input data
interleaved	Interleaved input data

Return Value

none

Description

This function splits the input data into two working sets. One set contains the non-interleaved input data and is used with the MAP 1 decoding. The other contains the interleaved input data and is used with the MAP2 decoding. This function is used in shared processing mode.

Example

```
TCP_demuxInput(rate, frameLen, input,
      interleaver, nonInterleaved, interleaved);
```

TCP_END_NATIVE *Value indicating native endian format*

Constant

TCP_END_NATIVE

Description

This constant allows selection of the native format for all data transferred to and from the TCP. That is to say that all data is contiguous in memory with incrementing addresses.

TCP_END_PACKED32 *Value indicating little endian format within packed 32-bit words*

Constant

TCP_END_PACKED32

Description

This constant allows selection of the packed 32-bit format for data transferred to and from the TCP. That is to say that all data is packed into 32-bit words in little endian format and these words are contiguous in memory.

TCP_errTest — *Returns the error code*

Function Uint32 TCP_errTest();

Arguments none

Return Value Error code Code error value

Description Returns an ERR bit indicating what TCP error has occurred.

Example

```
/* check whether an error has occurred */
if (TCP_errorStat()){
  error = TCP_ErrGet();
} /* end if */
```

TCP_FLEN_MAX — *Maximum frame length*

Constant TCP_FLEN_MAX

Description This constant equals the maximum frame length programmable into the TCP.

TCP_genIc — *Generates the TCP_ConfigIc structure*

Function

```
void TCP_genIc(
    TCP_Params *restrict configParms,
    TCP_UserData *restrict xabData,
    TCP_ConfigIc *restrict configIc
)
```

Arguments

configParms	Pointer to Channel parameters structure
xabData	Pointer to tail values at the end of the channel data
configIc	Pointer to Input Configuration structure

Return Value none

Description Generates the required input configuration values needed to program the TCP based on the parameters provided by configParms.

Example

```
extern TCP_Params *params;
extern TCP_UserData *xabData;
TCP_ConfigIc *config;
...
TCP_genIc(params, xabData, config);
...
```

TCP_genParams *Sets basic TCP Parameters*

Function

```
Uint32 TCP_genParams(
    TCP_BaseParams *configBase,
    TCP_Params *configParams
)
```

Arguments

configBase	Pointer to TCP_BaseParams structure
configParams	Output TCP_Params structure pointer

Return Value number of sub-blocks

Description Copies the basic parameters under the output TCP_Params parameters structure and returns the number of sub-blocks.

Example

```
Uint32 numSubblk;
TCP_Params tcpParam0;
TCP_ConfigIc *config;
numSubblk = TCP_genParams(&tcpBaseParam0, &tcpParam0);
```

TCP_getAccessErr *Returns access error flag*

Function Uint32 TCP_getAccessErr();

Arguments none

Return Value Error value of access error bit

Description Returns the ACC bit value indicating whether an invalid access has been made to the TCP during operation.

Example

```
/* check whether an invalid access has been made */
if (TCP_getAccessErr()){
  ...
} /* end if */
```

TCP_getAprioriEndian *Returns Apriori data endian configuration*

Function Uint32 TCP_getAprioriEndian();

Arguments none

Return Value Endian Endian setting for apriori data

Description Returns the value programmed into the TCP_END register for the apriori data indicating whether the data is in its native 8-bit format (‘1’) or consists of values packed in little endian format into 32-bit words (‘0’). This should always be ‘0’ for little endian operation.

See also `TCP_aprioriEndianSet, TCP_nativeEndianSet, TCP_packed32EndianSet, TCP_extEndianGet, TCP_interEndianGet, TCP_sysParEndianGet, TCP_extEndianSet, TCP_interEndianSet, TCP_sysParEndianSet.`

Example

```
If (TCP_getAprioriEndian()){
  ...
} /* end if */
```

TCP_getExtEndian *Returns the Extrinsics data endian configuration*

Function Uint32 TCP_getExtEndian();

Arguments none

Return Value Endian Endian setting for extrinsics data

Description Returns the value programmed into the TCP_END register for the extrinsics data indicating whether the data is in its native 8-bit format (‘1’) or consists of values packed in little endian format into 32-bit words (‘0’). This should always be ‘0’ for little endian operation.

See also `TCP_setExtEndian, TCP_setNativeEndian, TCP_setPacked32Endian, TCP_getAprioriEndian, TCP_getInterEndian, TCP_getSysParEndian, TCP_setAprioriEndian, TCP_setInterEndian, TCP_setSysParEndian.`

Example

```
If (TCP_getExtEndian()){
  ...
} /* end if */
```

TCP_getFrameLenErr *Returns the frame length error status*

Function Uint32 TCP_getFrameLenErr();

Arguments none

Return Value Error flag Boolean indication of frame length error

Description Returns a Boolean value indicating whether an invalid frame length has been programmed in the TCP during operation.

Example

```
/* check whether an invalid access has been made */
if (TCP_getFrameLenErr()){
  ...
} /* end if */
```

TCP_getIcConfig *Returns the IC values already programmed into the TCP*

Function void TCP_getIcConfig(TCP_ConfigIc *config)

Arguments config Pointer to Input Configuration structure

Return Value none

Description Reads the input configuration values currently programmed into the TCP.

Example

```
TCP_ConfigIc *config;
...
TCP_getIcConfig(config);
...
```

TCP_getInterEndian *Returns the interleaver table data endian*

Function Uint32 TCP_getInterEndian();

Arguments none

Return Value Endian Endian setting for interleaver table data

Description Returns the value programmed into the TCP_END register for the interleaver table data indicating whether the data is in its native 8-bit format (‘1’) or consists of values packed in little endian format into 32-bit words (‘0’). This should always be ‘0’ for little endian operation.

See also `TCP_setExtEndian, TCP_setNativeEndian, TCP_setPacked32Endian, TCP_getAprioriEndian, TCP_getExtEndian, TCP_getSysParEndian, TCP_setAprioriEndian, TCP_setInterEndian, TCP_setSysParEndian.`

Example

```
If (TCP_getInterEndian()){
  ...
} /* end if */
```

TCP_getInterleaveErr *Returns the interleaver table error status*

Function Uint32 TCP_getInterleaveErr();

Arguments none

Return Value Error flag value of interleaver table error bit

Description Returns an INTER value bit indicating whether the TCP was incorrectly programmed to receive an interleaver table. An interleaver table can only be sent when operating in standalone mode. This bit indicates if an interleaver table was sent when in shared processing mode.

Example

```
/* check whether the TCP was programmed to receive
   an interleaver table when in shared processing
   mode. */
if (TCP_getInterleaveErr()){
  ...
} /* end if */
```

TCP_getLastRelLenErr *Returns the error status for a bad reliability length*

Function Uint32 TCP_getLastRelLenErr();

Arguments none

Return Value Error flag value of an error for the reliability length of the last subframe (LR bit)

Description Returns the LR bit value indicating whether the TCP was programmed with a bad reliability length for the last subframe. The reliability length must be greater than or equal to 40 to be valid.

Example

```
/* check whether the TCP was programmed with a bad
   reliability length for the last frame. */
if (TCP_getLastRelLenErr()){
  ...
} /* end if */
```

TCP_getModeErr — *Returns the error status for a bad TCP mode*

Function Uint32 TCP_getModeErr();

Arguments none

Return Value Error flag Value of mode error bit

Description Returns the MODE bit value indicating whether an invalid MAP mode was programmed into the TCP. Only values of 4, 5, and 7 are valid.

Example

```
/* check whether the TCP was programmed using an
   invalid mode. */
if (TCP_getModeErr()){
  ...
} /* end if */
```

TCP_getNumIt — *Returns the number of iterations performed by the TCP*

Function Uint32 TCP_getNumit();

Arguments none

Return Value iterations The number of iterations performed by the TCP

Description Returns the number of iterations executed by the TCP in standalone processing mode. This function reads the output parameters register. Alternatively, the EDMA can be used to transfer the output parameters following the hard decisions (recommended).

Example

```
numIter = TCP_getNumit();
```

TCP_getOutParmErr — *Returns the output parameter*

Function Uint32 TCP_getOutParmErr();

Arguments none

Return Value Error flag value of output parameters error

Description Returns the OP bit value indicating whether the TCP was programmed to transfer output parameters in shared processing mode. The output parameters are only valid when operating in standalone mode.

Example

```
/* check whether the TCP was programmed to provide
   output parameters when in Shared Processing mode. */
if (TCP_getOutParmErr()){
  ...
} /* end if */
```

TCP_getProlLenErr *Returns the error status for an invalid prolog length*

Function Uint32 TCP_getProlLenErr();

Arguments none

Return Value Error flag Value of Prolog Length error

Description Returns the P bit value indicating whether an invalid prolog length has been programmed into the TCP.

Example

```
/* check whether an invalid prolog length has been
   programmed. */
if (TCP_getProlLenErr()){
  ...
} /* end if */
```

TCP_getRateErr *Returns the error status for an invalid rate*

Function Uint32 TCP_getRateErr();

Arguments none

Return Value Error flag Value of rate error

Description Returns the RATE bit value indicating whether an invalid rate has been programmed into the TCP.

Example

```
/* check whether an invalid rate has been programmed */
if (TCP_getRateErr()){
  ...
} /* end if */
```

TCP_getRelLenErr *Returns the error status for and invalid reliability length*

Function Uint32 TCP_getRelLenErr();

Arguments none

Return Value Error flag Value of reliability length error

Description Returns the R bit value indicating whether an invalid reliability length has been programmed into the TCP.

Example

```
/* check whether an invalid reliability length has been
   programmed. */
if (TCP_getRelLenErrG()){
  ...
} /* end if */
```

TCP_getSubFrameErr *Returns sub-frame error flag*

Function Uint32 TCP_getSubFrameErr();

Arguments none

Return Value Error flag Boolean indication of sub-frame error

Description Returns a Boolean value indicating whether the sub-frame length programmed into the TCP is invalid.

Example

```
/* check whether an invalid sub-frame length has been
   programmed. */
if (TCP_getSubFrameErr()){
  ...
} /* end if */
```

TCP_getSysParEndian *Returns Systematics and Parities data endian configuration*

Function Uint32 TCP_getSysParEndian();

Arguments none

Return Value Endian Endian setting for systematics and parities data

Description Returns the value programmed into the TCP_END register for the systematics and parities data, indicating whether the data is in its native 8-bit format (‘1’) or consists of values packed in little endian format into 32-bit words (‘0’). This should always be ‘0’ for little endian operation.

See also `TCP_setSysParEndian, TCP_setNativeEndian, TCP_setPacked32Endian.`

Example

```
If (TCP_getSysParEndian()){
  ...
} /* end if */
```

TCP_icConfig — *Stores the IC values into the TCP*

Function void TCP_icConfig(TCP_ConfigIc *config)

Arguments Config Pointer to Input Configuration structure

Return Value none

Description Stores the input configuration values currently programmed into the TCP. This is not the recommended means by which to program the TCP, as it is more efficient to transfer the IC values using the EDMA, but can be used in test code.

Example

```
extern TCP_Params *params;
extern TCP_UserData *xabData;
TCP_ConfigIc *config;
...
TCP_genIc(params, xabData, config);
TCP_icConfig(config);
...
```

TCP_icConfigArgs — *Stores the IC values into the TCP using arguments*

Function

```
Void TCP_icConfigArgs(
    Uint32 ic0,
    Uint32 ic1,
    Uint32 ic2,
    Uint32 ic3,
    Uint32 ic4,
    Uint32 ic5,
    Uint32 ic6,
    Uint32 ic7,
    Uint32 ic8,
    Uint32 ic9,
    Uint32 ic10,
    Uint32 ic11
)
```

Arguments

ic0 Input Configuration word 0 value

ic1 Input Configuration word 1 value

ic2 Input Configuration word 2 value

ic3	Input Configuration word 3 value
ic4	Input Configuration word 4 value
ic5	Input Configuration word 5 value
ic6	Input Configuration word 6 value
ic7	Input Configuration word 7 value
ic8	Input Configuration word 8 value
ic9	Input Configuration word 9 value
ic10	Input Configuration word 10 value
ic11	Input Configuration word 11 value

Return Value none

Description Stores the input configuration values currently programmed into the TCP. This is not the recommended means by which to program the TCP, as it is more efficient to transfer the IC values using the EDMA, but can be used in test code.

Example

```
TCP_icConfigArgs(
  0x00283200          /* IC0  */
  0x00270000          /* IC1  */
  0x00080118          /* IC2  */
  0x001E0014          /* IC3  */
  0x00000000          /* IC4  */
  0x00000002          /* IC5  */
  0x00E3E6F2          /* IC6  */
  0x00E40512          /* IC7  */
  0x00000000          /* IC8  */
  0x00F5FA1E          /* IC9  */
  0x00F00912          /* IC10 */
  0x00000000          /* IC11 */
);
```

TCP_interleaveExt *Interleaves extrinsics data*

Function

Void TCP_interleaveExt(
    TCP_ExtrinsicData *restrict aprioriMap2,
    const TCP_ExtrinsicData *restrict extrinsicsMap1,
    const Uint16 *restrict interleaverTable,
    Uint32 numExt);

Arguments

aprioriMap2	Apriori data for MAP2 decode
extrinsicsMap1	Extrinsics data following MAP1 decode
interleaverTable	Interleaver data table
NumExt	Number of Extrinsics

Return Value

none

Description

This function interleaves the MAP1 extrinsics data to generate apriori data for the MAP2 decode. This function is for use in performing shared processing.

Example

```
<...MAP 1 decode...>
TCP_interleaveExt(aprioriMap2, extrinsicsMap1,
                  InterleaverTable, numExt);
<...MAP 2 decode...>
```

TCP_makeTailArgs *Generates the Tail values used for ICCIC11*

Function

Uint32 TCP_makeTailArgs(
    Uint8 byte31_24,
    Uint8 byte23_16,
    Uint8 byte15_8,
    Uint8 byte7_0
)

Arguments

byte31_24	Byte to be placed in bits 31–24 of the 32-bit value
byte23_16	Byte to be placed in bits 23–16 of the 32-bit value
byte15_8	Byte to be placed in bits 15–8 of the 32-bit value
byte7_0	Byte to be placed in bits 7–0 of the 32-bit value

Return Value none

Description Formats individual bytes into a 32-bit word

Example

```
tail1 = TCP_makeTailArgs(0         , xabData[10],
                         xabData[8], xabData[6]);
```

TCP_MAP_MAP1A *Value indicating the first iteration of a MAP1 decoding*

Constant TCP_MAP_MAP1A

Description This constant allows selection of the Map 1 decoding mode used when operating in shared processing mode on the first iteration through the data. The first iteration through the Map 1 decoding is unique in that no apriori data is set to the TCP.

TCP_MAP_MAP1B *Value indicating a MAP1 decoding (any iteration after the first)*

Constant TCP_MAP_MAP1B

Description This constant allows selection of the Map 1decoding mode used when operating in shared processing mode on any but the first iteration through the data. The first iteration through the Map 1 decoding is unique in that no apriori data is set to the TCP.

TCP_MAP_MAP2 *Value indicating a MAP2 decoding*

Constant TCP_MAP_MAP2

Description This constant allows selection of the Map 2decoding mode used when operating in shared processing mode.

TCP_MODE_SA *Value indicating standalone processing*

Constant TCP_MODE_SA

Description This constant allows selection of standalone processing mode.

TCP_MODE_SP *Value indicating shared processing mode*

Constant TCP_MODE_SP

Description This constant allows selection of shared processing mode.

TCP_normalCeil — *Normalized ceiling function*

Function Uint32 TCP_normalCeil(Uint32 val1, Uint32 val2) ;

Arguments

val1	Value to be augmented
val2	Value by which val1 must be divisible

Return Value

ceilVal	The smallest number greater than or equal to val1 that is divisible by val2.

Description Returns the smallest number greater than or equal to val1 that is divisible by val2.

Example

```
winSize = TCP_normalCeil(winSize, numSlidingWindow);
```

TCP_pause — *Pauses the TCP by writing a '1' to the pause bit in TCP_EXE*

Function void TCP_pause();

Arguments none

Return Value none

Description This function pauses the TCP by writing a '1' to the PAUSE field of the TCP_EXE register. See also `TCP_start()` and `TCP_unpause()`.

Example

```
TCP_pause();
```

TCP_RATE_1_2 — *Value indicating a rate of 1/2*

Constant TCP_RATE_1_2

Description This constant allows selection of a rate of 1/2.

TCP_RATE_1_3 — *Value indicating a rate of 1/3*

Constant TCP_RATE_1_3

Description This constant allows selection of a rate of 1/3.

TCP_RATE_1_4 — *Value indicating a rate of 1/4*

Constant TCP_RATE_1_4

Description This constant allows selection of a rate of 1/4.

TCP_RLEN_MAX *Maximum reliability length*

Constant TCP_RLEN_MAX

Description This constant equals the maximum reliability length programmable into the TCP.

TCP_setAprioriEndian *Sets Apriori data endian configuration*

Function Void TCP_setAprioriEndian(Uint32 endianMode);

Arguments Endian Endian setting for apriori data

Return Value none

Description This function programs TCP to view the format of the apriori data as either native 8-bit format (‘1’) or values packed into 32-bit words in little endian format (‘0’). This should always be ‘0’ for little endian operation.

See also `TCP_getAprioriEndian, TCP_setNativeEndian, TCP_setPacked32Endian, TCP_getExtEndian, TCP_getInterEndian, TCP_getSysParEndian, TCP_setExtEndian, TCP_setInterEndian, TCP_setSysParEndian.`

Example

```
TCP_setAprioriEndian(TCP_END_PACKED32);
```

TCP_setExtEndian *Sets the Extrinsics data endian configuration*

Function Void TCP_setExtEndian(Uint32 endianMode);

Arguments Endian Endian setting for extrinsics data

Return Value none

Description This function programs TCP to view the format of the extrinsics data as either native 8-bit format (‘1’) or values packed into 32-bit words in little endian format (‘0’). This should always be ‘0’ for little endian operation.

See also `TCP_getExtEndian, TCP_setNativeEndian, TCP_setPacked32Endian, TCP_getAprioriEndian, TCP_getInterEndian, TCP_getSysParEndian, TCP_setAprioriEndian, TCP_setInterEndian, TCP_setSysParEndian.`

Example `TCP_setAprioriEndian(TCP_END_PACKED32);`

TCP_setInterEndian *Sets the interleaver table data endian*

Function Void TCP_setInterEndian(Uint32 endianMode);

Arguments

Endian	Endian setting for interleaver table data The following constants can be used: ❏ TCP_END_PACKED32 or 0 ❏ TCP_END_NATIVE or 1

Return Value none

Description This function programs TCP to view the format of the interleaver table data as either native 8-bit format ('1') or values packed into 32-bit words in little endian format ('0'). This should always be '0' for little endian operation.

See also `TCP_getInterEndian, TCP_setNativeEndian, TCP_setPacked32Endian, TCP_getAprioriEndian, TCP_getExtEndian, TCP_getSysParEndian, TCP_setAprioriEndian, TCP_setExtEndian, TCP_setSysParEndian.`

Example
```
TCP_setInterEndian(TCP_END_PACKED32);
```

TCP_setNativeEndian *Sets all data formats to be native (not packed)*

Function void TCP_setNativeEndian();

Arguments none

Return Value none

Description This function programs the TCP to view the format of all data as native 8-/16-bit format. This should only be used when running in big endian mode.

See also `TCP_setExtEndian, TCP_setPacked32Endian, TCP_getAprioriEndian, TCP_getExtEndian, TCP_getSysParEndian, TCP_setAprioriEndian, TCP_setInterEndian, TCP_setSysParEndian.`

Example
```
TCP_setNativeEndian();
```

TCP_setPacked32Endian *Sets all data formats to packed data*

Function void TCP_setPacked32Endian();

Arguments none

Return Value none

Description This function programs the TCP to view the format of all data as packed data in 32-bit words. This should always be used when running in little endian mode and should be used in big endian mode only if the CPU is formatting the data.

See also `TCP_setNativeEndian, TCP_setExtEndian, TCP_getAprioriEndian, TCP_getExtEndian, TCP_getSysParEndian, TCP_setAprioriEndian, TCP_setInterEndian, TCP_setSysParEndian.`

Example

```
TCP_setPacked32Endian();
```

TCP_setParams *Generates IC0–C5 based on channel parameters*

Function

```
void TCP_setParams(
    TCP_Params   *configParms,
    TCP_ConfigIc *configIc
);
```

Arguments

configParms Pointer to the user channel parameters structure

configIc Pointer to the IC values structure

Return Value none

Description This function generates the input control values IC0–IC5 based on the user channel parameters contained in the `configParms` structure.

Example

```
extern TCP_Params *configParms;
TCP_ConfigIc *configIC;
  ...
TCP_setParams(configParms, configIc);
```

TCP_setSysParEndian — *Sets Systematics and Parities data endian configuration*

Function Void TCP_setSysParEndian(Uint32 endianMode);

Arguments Endian Endian setting for systematics and parities data

Return Value none

Description This function programs the TCP to view the format of the systematics and parities data as either native 8-bit format (‘1’) or values packed into 32-bit words in little endian format (‘0’). This should always be ‘0’ for little endian operation.

See also `TCP_getSysParEndian, TCP_setNativeEndian, TCP_setPacked32Endian.`

Example

```
TCP_setSysParEndian(TCP_END_PACKED32);
```

TCP_STANDARD_3GPP — *Value indicating the 3GPP standard*

Constant TCP_STANDARD_3GPP

Description This constant allows selection of the 3GPP standard.

TCP_STANDARD_IS2000 — *Value indicating the IS2000 standard*

Constant TCP_STANDARD_IS2000

Description This constant allows selection of the IS2000 standard.

TCP_start — *Starts the TCP by writing a ’1’ to the start bit in TCP_EXE*

Function void TCP_start();

Arguments none

Return Value none

Description This function starts the TCP by writing a ‘1’ to the START field of the TCP_EXE register. See also `TCP_pause()` and `TCP_unpause().`

Example

```
TCP_start();
```

TCP_statError *Returns the error status*

Function Uint32 TCP_statError();

Arguments none

Return Value Error status Value of error bit

Description Returns the ERR bit value indicating whether any TCP error has occurred.

Example

```
/* check whether an error has occurred */
if (TCP_statError()){
  ...
} /* end if */
```

TCP_statPause *Returns the pause status*

Function Uint32 TCP_statPause();

Arguments none

Return Value Status Boolean status

Description Returns a Boolean status indicating whether the TCP is paused or not.

Example

```
/* pause the TCP */
TCP_pause();
/* wait for pause to take place */
while (!TCP_statPause());
```

TCP_statRun *Returns the run status*

Function Uint32 TCP_statRun();

Arguments none

Return Value Status Boolean status

Description Returns a Boolean status indicating whether the TCP is running

Example

```
/* start the TCP */
TCP_start();
/* check that the TCP is running */
while (!TCP_statRun());
```

TCP_statWaitApriori *Returns the apriori data status*

Function Uint32 TCP_statWaitApriori();

Arguments none

Return Value Status Boolean WAP status

Description Returns the WAP bit status indicating whether the TCP is waiting to receive apriori data.

Example

```
/* check if TCP is waiting on apriori data */
if (TCP_statWaitApriori()){
  ...
} /* end if */
```

TCP_statWaitExt *Returns the extrinsics data*

Function Uint32 TCP_statWaitExt();

Arguments none

Return Value Status Boolean REXTstatus

Description Returns the REXT bit status indicating whether the TCP is waiting for extrinsic data to be read.

Example

```
/* check if TCP has extrinsic data pending */
if (TCP_statWaitExt()){
  ...
} /* end if */
```

TCP_statWaitHardDec *Returns the hard decisions data status*

Function Uint32 TCP_statWaitHardDec();

Arguments none

Return Value Status RHD status

Description Returns the RHD bit status indicating whether the TCP is waiting for the hard decisions data to be read.

Example

```
/* check if TCP has hard decisions data pending*/
if (TCP_statWaitHardDec()){
  ...
} /* end if */
```

TCP_statWaitIc *Returns the IC data status*

Function Uint32 TCP_statWaitIc();

Arguments none

Return Value Status WIC status

Description Returns the WIC bit status indicating whether the TCP is waiting to receive new IC values.

Example

```
/* check if TCP is waiting on new IC values */
if (TCP_statWaitIc()){
  ...
} /* end if */
```

TCP_statWaitInter *Returns the interleaver table data status*

Function Uint32 TCP_statWaitInter();

Arguments none

Return Value Status WINT status

Description Returns the WINT status indicating whether the TCP is waiting to receive interleaver table data.

Example

```
/* check if TCP is waiting on interleaver data */
if (TCP_statWaitInter()){
  ...
} /* end if */
```

TCP_statWaitOutParm *Returns the output parameters data status*

Function Uint32 TCP_statWaitOutParm();

Arguments none

Return Value Status ROP status

Description Returns the ROP bit status indicating whether the TCP is waiting for the output parameters to be read.

Example

```
/* check if TCP has output parameters data pending */
if (TCP_statWaitOutParm()){
  ...
} /* end if */
```

TCP_statWaitSysPar — *Returns the systematics and parities data status*

Function Uint32 TCP_statWaitSysPar();

Arguments none

Return Value Status WSP status

Description Returns the WSP bit status indicating whether the TCP is waiting to receive systematic and parity data.

Example

```
/* check if TCP is waiting on systematic and parity
   data */
if (TCP_statWaitSysPar()){
  ...
} /* end if */
```

TCP_tailConfig — *Generates IC6–IC11 tail values*

Function

```
void TCP_tailConfig(
    TCP_Standard  standard,
    TCP_Mode      mode,
    TCP_Map       map,
    TCP_Rate      rate,
    TCP_UserData *xabData,
    TCP_ConfigIc *configIc
);
```

Arguments

standard	3G standard
mode	Processing mode
map	Map mode for shared processing
rate	Rate
xabData	Pointer to the tail data
configIc	Pointer to the IC values structure

Return Value none

Description This function generates the input control values IC6–IC11 based on the processing to be performed by the TCP. These values consist of the tail data following the systematics and parities data.

This function actually calls specific tail generation functions depending on the standard followed: `TCP_tailConfig3GPP` or `TCP_tailConfigIS2000`.

Example

```
extern TCP_Params   *configParms;
extern TCP_UserData *userData;
TCP_ConfigIc *configIC;
TCP_Standard  standard = configParms->standard;
TCP_Mode      mode     = configParms->mode;
TCP_Map       map      = configParms->map;
TCP_Rate      rate     = configParms->rate;
Uint16        index    = configParms->frameLen * rate;
TCP_UserData *xabData  = &userData[index];
  ...
TCP_setParams(standard, mode, map, rate, xabData,
              configIc);
```

TCP_tailConfig3GPP *Generates IC6–IC11 tail values for G3PP channels*

Function

```
void TCP_tailConfig3GPP(
    TCP_Mode     mode,
    TCP_Map      map,
    TCP_UserData *xabData,
    TCP_ConfigIc *configIc
);
```

Arguments

mode	Processing mode
map	Map mode for shared processing
xabData	Pointer to the tail data
configIc	Pointer to the IC values structure

Return Value

none

Description

This function generates the input control values IC6–IC11 for 3GPP channels. These values consist of the tail data following the systematics and parities data. This function is called from the generic `TCP_tailConfig` function.

See also: `TCP_tailConfig` and `TCP_tailConfigIS2000`.

Example

```
extern TCP_Params   *configParms;
extern TCP_UserData *userData;
TCP_ConfigIc *configIC;
TCP_Mode      mode     = configParms->mode;
TCP_Map       map      = configParms->map;
Uint16        index    = configParms->frameLen * rate;
TCP_UserData *xabData  = &userData[index];
  ...
TCP_setParams(mode, map, xabData, configIc);
```

TCP_tailConfigIS2000 *Generates IC6–IC11 tail values for IS2000 channels*

Function

```
Void TCP_tailConfigIS2000(
    TCP_Mode      mode,
    TCP_Map       map,
    TCP_Rate      rate,
    TCP_UserData *xabData,
    TCP_ConfigIc *configIc
);
```

Arguments

Mode	Processing mode
Map	Map mode for shared processing
Rate	Rate
XabData	Pointer to the tail data
configIc	Pointer to the IC values structure

Return Value

none

Description

This function generates the input control values IC6 – IC11 for IS2000 channels. These values consist of the tail data following the systematic and parity data. This function is called from the generic `TCP_tailConfig` function.

See also: `TCP_tailConfig` and `TCP_tailConfig3GPP`.

Example

```
extern TCP_Params   *configParms;
extern TCP_UserData *userData;
TCP_ConfigIc *configIC;
TCP_Mode      mode     = configParms->mode;
TCP_Map       map      = configParms->map;
TCP_Rate      rate     = configParms->rate;
Uint16                index        =  configParms->frameLen  *
rate;TCP_UserData *xabData  = &userData[index];
  ...
TCP_setParams(standard, mode, map, rate, xabData,
              configIc);
```

TCP_unpause

Unpauses the TCP by writing a '1' to the unpause bit in TCP_EXE

Function void TCP_unpause();

Arguments none

Return Value none

Description This function un-pauses the TCP by writing a '1' to the UNPAUSE field of the TCP_EXE register. See also `TCP_start()` and `TCP_pause()`.

Example

```
TCP_pause();
...
TCP_unpause();
```

Chapter 20

TIMER Module

This chapter describes the TIMER module, lists the API functions and macros within the module, discusses how to use a TIMER device, and provides a TIMER API reference section.

Topic **Page**

20.1 Overview

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

Table 20–1 lists the configuration structure for use with the TIMER functions. Table 20–2 lists the functions and constants available in the CSL TIMER module.

Table 20–1. TIMER Configuration Structure

Syntax	Type	Description	See page ...
TIMER_Config	S	Structure used to set up a timer device	20-6

Table 20–2. TIMER APIs

(a) Primary Functions

Syntax	Type	Description	See page ...
TIMER_close	F	Closes a previously opened timer device	20-7
TIMER_config	F	Configure timer using configuration structure	20-7
TIMER_configArgs	F	Sets up the timer using the register values passed in	20-8
TIMER_open	F	Opens a TIMER device for use	20-8
TIMER_pause	F	Pauses the timer	20-9
TIMER_reset	F	Resets the timer device associated to the handle	20-9
TIMER_resume	F	Resumes the timer after a pause	20-10
TIMER_start	F	Starts the timer device running	20-10

(b) Auxiliary Functions and Constants

Syntax	Type	Description	See page ...
TIMER_DEVICE_CNT	C	A compile time constant; number of timer devices present	20-10
TIMER_getBiosHandle	F	Returns the timer handle of the timer used by BIOS	20-11
TIMER_getConfig	F	Reads the current Timer configuration values	20-11
TIMER_getCount	F	Returns the current timer count value	20-12
TIMER_getDatIn	F	Reads the value of the TINP pin	20-12
TIMER_getEventId	F	Obtains the event ID for the timer device	20-12

Note: F = Function; C = Constant

Syntax	Type	Description	See page ...
TIMER_getPeriod	F	Returns the period of the timer device	20-13
TIMER_getTStat	F	Reads the timer status; value of timer output	20-13
TIMER_resetAll	F	Resets all timer devices	20-13
TIMER_setCount	F	Sets the count value of the timer	20-14
TIMER_setDatOut	F	Sets the data output value	20-14
TIMER_setPeriod	F	Sets the timer period	20-15
TIMER_SUPPORT	C	A compile time constant whose value is 1 if the device supports the TIMER module	20-15

Note: F = Function; C = Constant

20.1.1 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_config()` or by passing register values to the `TIMER_configArgs()` function. To assist in creating register values, there are *TIMER_RMK* (make) macros that construct register values based on field values. In addition, the symbol constants may be used for the field values.

20.2 Macros

There are two types of TIMER macros: those that access registers and fields, and those that construct register and field values.

Table 20–3 lists the TIMER macros that access registers and fields, and Table 20–4 lists the TIMER macros that construct register and field values. The macros themselves are found in Chapter 24, *Using the HAL Macros*.

The TIMER module includes handle-based macros.

Table 20–3. TIMER Macros that Access Registers and Fields

Macro	Description/Purpose	See page ...
TIMER_ADDR(<REG>)	Register address	24-12
TIMER_RGET(<REG>)	Returns the value in the peripheral register	24-18
TIMER_RSET(<REG>,x)	Register set	24-20
TIMER_FGET(<REG>,<FIELD>)	Returns the value of the specified field in the peripheral register	24-13
TIMER_FSET(<REG>,<FIELD>,fieldval)	Writes *fieldval* to the specified field in the peripheral register	24-15
TIMER_FSETS(<REG>,<FIELD>,<SYM>)	Writes the symbol value to the specified field in the peripheral	24-17
TIMER_RGETA(addr,<REG>)	Gets register for a given address	24-19
TIMER_RSETA(addr,<REG>,x)	Sets register for a given address	24-20
TIMER_FGETA(addr,<REG>,<FIELD>)	Gets field for a given address	24-13
TIMER_FSETA(addr,<REG>,<FIELD>, fieldval)	Sets field for a given address	24-16
TIMER_FSETSA(addr,<REG>,<FIELD>, <SYM>)	Sets field symbolically for a given address	24-17
TIMER_ADDRH(h,<REG>)	Returns the address of a memory-mapped register for a given handle	24-12
TIMER_RGETH(h,<REG>)	Returns the value of a register for a given handle	24-19
TIMER_RSETH(h,<REG>,x)	Sets the register value to x for a given handle	24-21
TIMER_FGETH(h,<REG>,<FIELD>)	Returns the value of the field for a given handle	24-14
TIMER_FSETH(h,<REG>,<FIELD>, fieldval)	Sets the field value to x for a given handle	24-16

Table 20–4. TIMER Macros that Construct Register and Field Values

Macro	Description/Purpose	See page ...
TIMER_<REG>_DEFAULT	Register default value	24-21
TIMER_<REG>_RMK()	Register make	24-23
TIMER_<REG>_OF()	Register value of ...	24-22
TIMER_<REG>_<FIELD>_DEFAULT	Field default value	24-24
TIMER_FMK()	Field make	24-14
TIMER_FMKS()	Field make symbolically	24-15
TIMER_<REG>_<FIELD>_OF()	Field value of ...	24-24
TIMER_<REG>_<FIELD>_<SYM>	Field symbolic value	24-24

20.3 Configuration Structure

TIMER_Config *Structure used to setup 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_config()` function. You can use literal values or the *_RMK* macros to create the structure member values.

Example

```
TIMER_Config MyConfig = {
  0x000002C0, /* ctl */
  0x00010000, /* prd */
  0x00000000  /* cnt */
};
…
TIMER_config(hTimer,&MyConfig);
```

20.4 Functions

20.4.1 Primary Functions

TIMER_close *Closes previously opened timer device*

Function

```
void TIMER_close(
    TIMER_Handle hTimer
);
```

Arguments

hTimer — Device handle. See `TIMER_open()`.

Return Value

none

Description

This function 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);
```

TIMER_config *Configure timer using configuration structure*

Function

```
void TIMER_config(
    TIMER_Handle hTimer,
    TIMER_Config *config
);
```

Arguments

hTimer — Device handle. See `TIMER_open()`.

config — Pointer to initialize configuration structure.

Return Value

none

Description

This function sets up the timer device using the configuration structure. The values of the structure are written to the TIMER registers. The timer control register (CTL) is written last. See `TIMER_configArgs()` and `TIMER_Config`.

Example

```
TIMER_Config MyConfig = {
  0x000002C0, /* ctl */
  0x00010000, /* prd */
  0x00000000  /* cnt */
};
...
TIMER_config(hTimer,&MyConfig);
```

TIMER_configArgs *Sets up timer using register values passed in*

Function

```
void TIMER_configArgs(
    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

This function 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_config()`.

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

Example

```
TIMER_configArgs (LTimer, 0x000002C0, 0x00010000, 0x00000000);
```

TIMER_open *Opens timer device for use*

Function

```
TIMER_Handle TIMER_open(
    int devNum,
    Uint32 flags
);
```

Arguments

devNum	Device Number: ❑ TIMER_DEVANY ❑ TIMER_DEV0 ❑ TIMER_DEV1 ❑ TIMER_DEV2
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);
```

TIMER_pause *Pauses timer*

Function

```
void TIMER_pause(
    TIMER_Handle hTimer
);
```

Arguments hTimer Device handle. See `TIMER_open()`.

Return Value none

Description This function pauses the timer. May be restarted using `TIMER_resume()`.

Example

```
TIMER_pause(hTimer);
…
TIMER_resume(hTimer);
```

TIMER_reset *Resets timer device associated to the Timer handle*

Function

```
void TIMER_reset(
    TIMER_Handle hTimer
);
```

Arguments hTimer Device handle. See `TIMER_open()`.

Return Value none

Description This function resets the timer device. Disables and clears the interrupt event and sets the timer registers to default values.

Example

```
TIMER_reset(hTimer);
```

TIMER_resume *Resumes timer after pause*

Function

```
void TIMER_resume(
    TIMER_Handle hTimer
);
```

Arguments hTimer Device handle. See `TIMER_open()`.

Return Value none

Description This function resumes the timer after a pause. See `TIMER_pause()`.

Example

```
TIMER_pause(hTimer);
…
TIMER_resume(hTimer);
```

TIMER_start *Starts timer device running*

Function

```
void TIMER_start(
    TIMER_Handle hTimer
);
```

Arguments hTimer Device handle. See `TIMER_open()`.

Return Value none

Description This function starts the timer device running. HLD of the CTL control register is released and the GO bit field is set.

Example

```
TIMER_start(hTimer);
```

20.4.2 Auxiliary Functions and Constants

TIMER_DEVICE_CNT *Compile time constant*

Constant TIMER_DEVICE_CNT

Description Compile-time constant; number of timer devices present.

TIMER_getBiosHandle

Returns the timer handle of the timer used by BIOS

Function TIMER_Handle TIMER_getBiosHandle();

Arguments none

Return Value hTimer Device handle. See `TIMER_open()`

Description This function returns the timer handle used by the CLK module under the DSP/BIOS Configuration Tool. This timer cannot be used as a timer, but the handle parameter allows you to use other Timer APIs – for example, accessing Timer GPIO pins.

Example

```
TIMER_Handle hTimerBios;
hTimerBios=TIMER_getBiosHandle();
tinp=TIMER_getDatIn(hTimerBios);
```

TIMER_getConfig

Reads the current TIMER configuration values

Function

```
void TIMER_getConfig(
    TIMER_Handle hTimer,
    TIMER_Config *config
);
```

Arguments hTimer Device handle. See `TIMER_open()`

config Pointer to a configuration structure.

Return Value none

Description This function reads the TIMER current configuration value

Example

```
TIMER_Config timerCfg;
TIMER_getConfig(hTimer,&timerCfg);
```

TIMER_getCount *Returns current timer count value*

Function

```
Uint32 TIMER_getCount(
    TIMER_Handle hTimer
);
```

Arguments

hTimer Device handle. See `TIMER_open()`.

Return Value

Count Value

Description

This function returns the current timer count value.

Example

```
cnt = TIMER_getCount(hTimer);
```

TIMER_getDatIn *Reads value of 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(hTimer);
```

TIMER_getEventId *Obtains event ID for 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);
```

TIMER_getPeriod *Returns period of timer device*

Function

```
Uint32 TIMER_getPeriod(
    TIMER_Handle hTimer
);
```

Arguments

hTimer		Device handle. See `TIMER_open()`.

Return Value

Period Value	Timer period

Description

This function returns the period of the timer device.

Example

```
p = TIMER_getPeriod(hTimer);
```

TIMER_getTstat *Reads 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

This function reads the timer status; value of timer output.

Example

```
status = TIMER_getTstat(hTimer);
```

TIMER_resetAll *Resets all timer devices supported by the chip device*

Function

```
void TIMER_resetAll();
```

Arguments

none

Return Value

none

Description

This function resets all timer devices supported by the chip device by clearing and disabling the interrupt event and setting the default timer register values for each timer device. See also TIMER_reset() function.

Example

```
TIMER_resetAll();
```

TIMER_setCount *Sets count value of timer*

Function

```
void TIMER_setCount(
    TIMER_Handle hTimer,
    Uint32 count
);
```

Arguments

hTimer	Device handle. See `TIMER_open()`.
count	Count value

Return Value

none

Description

This function sets the count value of the timer. The timer is not paused during the update.

Example

```
TIMER_setCount(hTimer,0x00000000);
```

TIMER_setDatOut *Sets 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

This function sets the data output value.

Example

```
TIMER_setDatOut(hTimer,0);
```

TIMER_setPeriod *Sets 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

This function sets the timer period. The timer is not paused during the update.

Example

```
TIMER_setPeriod(hTimer,0x00010000);
```

TIMER_SUPPORT *Compile time constant*

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 21

UTOPIA Module

This chapter describes the UTOPIA module, lists the API functions and macros within the module, discusses how to set the UTOPIA interface, and provides a UTOP API reference section.

Topic **Page**

21.1 Overview

For TMS320C64x™ devices, the UTOPIA consists of a transmit interface and a receive interface. Both interfaces are configurable via the configuration registers. The properties and functionalities of each interface can be set and controlled by using the CSL APIs dedicated to the UTOPIA interface.

Table 21–1 lists the configuration structure for use with the UTOP functions. Table 21–2 lists the functions and constants available in the CSL UTOPIA module

Table 21–1. UTOPIA Configuration Structure

Syntax	Type	Description	See page ...
UTOP_Config	S	The UTOPIA configuration structure used to set the control register and the Clock Detect register	21-6

Table 21–2. UTOPIA APIs

Syntax	Type	Description	See page ...
UTOP_config	F	Sets up the UTOPIA interface using the configuration structure	21-7
UTOP_configArgs	F	Sets up the UTOPIA control register and Clock detect register using the register values passed in	21-7
UTOP_enableRcv	F	Enables the receiver interface	21-8
UTOP_enableXmt	F	Enables the transmitter interface	21-8
UTOP_errClear	F	Clears a pending error bit	21-8
UTOP_errDisable	F	Disables an error bit event	21-9
UTOP_errEnable	F	Enables an error bit event	21-9
UTOP_errReset	F	Reset an error bit event by clearing and disabling the corresponding bits under EIPR and EIER respectively.	21-10
UTOP_errTest	F	Tests an error bit event	21-10
UTOP_getConfig	F	Reads the current UTOPIA configuration structure	21-11
UTOP_getEventId	F	Returns the CPU interrupt event number dedicated to the UTOPIA interface	21-11
UTOP_getRcvAddr	F	Returns the Slave Receive Queue Address.	21-11

Table 21–2. UTOPIA APIs (Continued)

Syntax	Type	Description	See page ...
UTOP_getXmtAddr	F	Returns the Slave Transmit Queue Address.	21-12
UTOP_intClear	F	Clears the relevant interrupt pending queue bit of the UTOPIA queue interfaces.	21-12
UTOP_intDisable	F	Disables the relevant interrupt queue bit of the UTOPIA queue interfaces.	21-12
UTOP_intEnable	F	Enables the relevant interrupt queue event of the UTOPIA queue interfaces.	21-13
UTOP_intReset	F	Clears and disables the interrupt queue event of the UTOPIA queue interfaces.	21-13
UTOP_intTest	F	Tests a queue event interrupt	21-14
UTOP_read	F	Reads from the slave receive queue	21-14
UTOP_SUPPORT	C	A compile time constant whose value is 1 if the device supports the UTOPIA module	21-14
UTOP_write	F	Writes into the slave transmit queue	21-15

Note: F = Function; C = Constant

21.1.1 Using UTOPIA APIs

To use the UTOPIA interfaces, you must first configure the Control register and the Clock Detect register by using the configuration structure to `UTOP_config()` or by passing register values to the `UTOP_configArgs()` function. To assist in creating a register value, there is the *UTOP_<REG>_RMK* (make) macro that builds register value based on field values. In addition, the symbol constants may be used for the field setting.

21.2 Macros

There are two types of UTOP macros: those that access registers and fields, and those that construct register and field values.

Table 21–3 lists the UTOP macros that access registers and fields, and Table 21–4 lists the UTOP macros that construct register and field values. The macros themselves are found in Chapter 24, *Using the HAL Macros*.

The UTOPIA module includes handle-based macros.

Table 21–3. UTOP Macros that Access Registers and Fields

Macro	Description/Purpose	See page ...
UTOP_ADDR(<REG>)	Register address	24-12
UTOP_RGET(<REG>)	Returns the value in the peripheral register	24-18
UTOP_RSET(<REG>,x)	Register set	24-20
UTOP_FGET(<REG>,<FIELD>)	Returns the value of the specified field in the peripheral register	24-13
UTOP_FSET(<REG>,<FIELD>,fieldval)	Writes *fieldval* to the specified field in the peripheral register	24-15
UTOP_FSETS(<REG>,<FIELD>,<SYM>)	Writes the symbol value to the specified field in the peripheral	24-17
UTOP_RGETA(addr,<REG>)	Gets register for a given address	24-19
UTOP_RSETA(addr,<REG>,x)	Sets register for a given address	24-20
UTOP_FGETA(addr,<REG>,<FIELD>)	Gets field for a given address	24-13
UTOP_FSETA(addr,<REG>,<FIELD>, fieldval)	Sets field for a given address	24-16
UTOP_FSETSA(addr,<REG>,<FIELD>, <SYM>)	Sets field symbolically for a given address	24-17

Table 21–4. UTOP Macros that Construct Register and Field Values

Macro	Description/Purpose	See page ...
UTOP_<REG>_DEFAULT	Register default value	24-21
UTOP_<REG>_RMK()	Register make	24-23
UTOP_<REG>_OF()	Register value of ...	24-22
UTOP_<REG>_<FIELD>_DEFAULT	Field default value	24-24
UTOP_FMK()	Field make	24-14
UTOP_FMKS()	Field make symbolically	24-15
UTOP_<REG>_<FIELD>_OF()	Field value of ...	24-24
UTOP_<REG>_<FIELD>_<SYM>	Field symbolic value	24-24

21.3 Configuration Structure

UTOP_Config *UTOP configuration structure*

Structure UTOP_Config

Members

Uint32 ucr	UTOP control register value
Uint32 cdr	UTOP Clock Detect register value

Description This is the UTOP configuration structure used to set up the UTOPIA registers. You create and initialize this structure then pass its address to the `UTOP_config()` function. You can use literal values or the *_RMK* macros to create the structure register value.

Example

```
UTOP_Config MyConfig = {
  0x00010001, /* ucr */  0x00FF00FF, /* cdr */
  };
…
UTOP_config(MyConfig);
```

21.4 Functions

UTOP_config

Sets up UTOP modes using a configuration structure

Function

```
void UTOP_config(
    UTOP_Config *config
);
```

Arguments

config Pointer to an initialized configuration structure

Return Value

none

Description

Sets up the UTOPIA using the configuration structure. The values of the structure are written to the UTOP associated register. See also `UTOP_configArgs()` and `UTOP_Config`.

Example

```
UTOP_Config MyConfig = {
  0x00010000, /* ucr */
  0x00FF00FF, /* cdr */
};
…
UTOP_config(&MyConfig);
```

UTOP_configArgs

Sets up UTOP mode using register value passed in

Function

```
Void UTOP_configArgs(
    Uint32 ucr,
    Uint32 cdr
 );
```

Arguments

ucr Control register value

cdr Clock Detect value

Return Value

none

Description

Sets up the UTOP mode using the register value passed in. The register value is written to the associated registers. See also `UTOP_config()`.

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

Example

```
UTOP_configArgs(
  0x00010000, /* ucr */
  0x00FF00FF, /* cdr */
);
```

UTOP_enableRcv — *Enables UTOPIA receiver interface*

Function void UTOP_enableRcv();

Arguments none

Return Value none

Description This function enables the UTOPIA receiver port.

Example

```
/* Configures UTOPIA */
UTOP_configArgs(
    0x00040004, /*ucr*/
    0x00FF00FF  /*cdr*/
);
/* Enables Receiver port */
UTOP_enableRcv();
```

UTOP_enableXmt — *Enables the UTOPIA transmitter interface*

Function void UTOP_enableXmt();

Arguments none

Return Value none

Description This function enables the UTOPIA transmitter port.

Example

```
/* Configures UTOPIA*/
UTOP_configArgs(
    0x00040004, /*ucr*/
    0x00FF00FF  /*cdr*/|
);
/* Enables Transmitter port */
UTOP_enableXmt();
```

UTOP_errClear — *Clears error condition bit*

Function

```
void UTOP_errClear(
    Uint32 errNum
);
```

Arguments errNum — Error condition ID from the following constant list:

- ❏ UTOP_ERR_RQS
- ❏ UTOP_ERR_RCF
- ❏ UTOP_ERR_RCP
- ❏ UTOP_ERR_XQS
- ❏ UTOP_ERR_XCF
- ❏ UTOP_ERR_XCP

Return Value none

Description This function clears the bit of given error condition ID of EIPR.

Example

```
/* Clears bit error  condition*/
UTOP_errClear(UTOP_ERR_RCF);
```

UTOP_errDisable *Disables the error interrupt bit*

Function

```
void UTOP_errDisable(
    Uint32 errNum
);
```

Arguments errNum Error condition ID from the following constant list:

- ❏ UTOP_ERR_RQS
- ❏ UTOP_ERR_RCF
- ❏ UTOP_ERR_RCP
- ❏ UTOP_ERR_XQS
- ❏ UTOP_ERR_XCF
- ❏ UTOP_ERR_XCP

Return Value none

Description This function disables the error interrupt event

Example

```
/* Disables  error condition interrupt*/
UTOP_errDisable(UTOP_ERR_RCF);
```

UTOP_errEnable *Enables the error interrupt bit*

Function

```
void UTOP_errEnable(
    Uint32 errNum
);
```

Arguments errNum Error condition ID from the following constant list:

- ❏ UTOP_ERR_RQS
- ❏ UTOP_ERR_RCF
- ❏ UTOP_ERR_RCP
- ❏ UTOP_ERR_XQS
- ❏ UTOP_ERR_XCF
- ❏ UTOP_ERR_XCP

Return Value none

Description This function enables the error interrupt event.

Example
```
/* Enables error condition interrupt */
UTOP_errEnable(UTOP_ERR_RCF);
```

UTOP_errReset *Resets the error interrupt event bit*

Function
```
void UTOP_errReset(
    Uint32 errNum
);
```

Arguments

errNum — Error condition ID from the following constant list:
- ❏ UTOP_ERR_RQS
- ❏ UTOP_ERR_RCF
- ❏ UTOP_ERR_RCP
- ❏ UTOP_ERR_XQS
- ❏ UTOP_ERR_XCF
- ❏ UTOP_ERR_XCP

Return Value none

Description This function resets the error interrupt event by disabling and clearing the error interrupt bit associated to the error condition.

Example
```
/* Resets error condition interrupt */
UTOP_errReset(UTOP_ERR_RCF);
```

UTOP_errTest *Tests the error interrupt event bit*

Function
```
Uint32 UTOP_errTest(
    Uint32 errNum
);
```

Arguments

errNum — Error condition ID from the following constant list:
- ❏ UTOP_ERR_RQS
- ❏ UTOP_ERR_RCF
- ❏ UTOP_ERR_RCP
- ❏ UTOP_ERR_XQS
- ❏ UTOP_ERR_XCF
- ❏ UTOP_ERR_XCP

Return Value val — Equals to 1 if Error event has occurred and 0 otherwise

Description This function tests the error interrupt event by returning the bit status.

Example

```
/* Enables error condition interrupt */
Uint32 errDetect;
UTOP_errEnable(UTOP_ERR_RCF);
errDetect = UTOP_errTest(UTOP_ERR_RCF)
```

UTOP_getConfig *Reads the current UTOP configuration structure*

Function

Uint32 UTOP_getConfig(
    UTOP_Config *Config
);

Arguments

Config Pointer to a configuration structure.

Return Value

none

Description

Get UTOP current configuration value. See also UTOP_config() and UTOP_configArgs() functions.

Example

```
UTOP_config UTOPCfg;

UTOP_getConfig(&UTOPCfg);
```

UTOP_getEventId *Returns the UTOPIA interrupt Event ID*

Function

Uint32 UTOP_getEventId();

Arguments

none

Return Value

val UTOPIA Event ID

Description

This function returns the event ID associated to the UTOPIA CPU-interrupt. See also IRQ_EVT_NNNN (IRQ Chapter 13)

Example

```
Uint32 UtopEventId;
UtopEventId = UTOP_getEventId();
```

UTOP_getRcvAddr *Returns the Receiver Queue address*

Function

Uint32 UTOP_getRcvAddr();

Arguments

none

Return Value

val UTOPIA Event ID

Description This function returns the address of the Receiver Queue. This address is needed when you read from the Receiver Port.

Example
```
Uint32 UtopRcvAddr;
UtopRcvAddr = UTOP_getRcvAddr();
```

UTOP_getXmtAddr *Returns the Transmit Queue address*

Function Uint32 UTOP_getXmtAddr();

Arguments none

Return Value val UTOPIA Event ID

Description This function returns the address of the Transmit Queue. This address is needed when you write to the Transmit Port.

Example
```
Uint32 UtopXmtAddr;
UtopXmtAddr = UTOP_getXmtAddr();
```

UTOP_intClear *Clears the interrupt bit related to Receive and Transmit Queues*

Function
```
void UTOP_intClear(
Uint32 intNum
);
```

Arguments intNum The interrupt ID from the following list:
- ❏ UTOP_INT_XQ
- ❏ UTOP_INT_RQ

Return Value none

Description Clears the associated bit to the interrupt ID of the utopia interrupt pending register (`UIPR`).

Example
```
/* Clears the flag of the receive event */
UTOP_intClear(UTOP_INT_RQ);
```

UTOP_intDisable *Disables the interrupt to the CPU*

Function
```
void UTOP_intDisable(
Uint32 intNum
);
```

Arguments intNum The interrupt ID from the following list:
- ❏ UTOP_INT_XQ
- ❏ UTOP_INT_RQ

Return Value none

Description Disables the interrupt bit to the CPU. No interrupts are sent if the corresponding event occurs.

Example

```
/* Disables the interrupt of the receive event */
UTOP_intDisable(UTOP_INT_RQ);
```

UTOP_intEnable *Enables the interrupt to the CPU*

Function

```
void UTOP_intEnable(
Uint32 intNum
);
```

Arguments intNum The interrupt ID from the following list:

- ❑ UTOP_INT_XQ
- ❑ UTOP_INT_RQ

Return Value none

Description Enables the interrupt to the CPU by setting the bit to 1. The interrupt event is sent to the CPU selector. The CPU interrupt is generated only if the relevant bit is set under UIER register.

Example

```
/* Enables the interrupt of the receive event */
UTOP_intEnable(UTOP_INT_RQ);
IRQ_enable(IRQ_EVT_UINT);
```

UTOP_intReset *Resets the interrupt to the CPU*

Function

```
void UTOP_intReset(
Uint32 intNum
);
```

Arguments intNum The interrupt ID from the following list:

- ❑ UTOP_INT_XQ
- ❑ UTOP_INT_RQ

Return Value none

Description Resets the interrupt to the CPU by disabling the interrupt bit uner UIER and clearing the pending bit of UIPR.

Example

```
/* Resets the interrupt of the receive event */
UTOP_intReset(UTOP_INT_RQ);
```

UTOP_intTest — *Tests the interrupt event*

Function

Uint32 UTOP_intReset(
Uint32 intNum
);

Arguments

intNum — The interrupt ID from the following list:
- ❑ UTOP_INT_XQ
- ❑ UTOP_INT_RQ

Return Value

val — Equal to 1 if the event has occurred and 0 otherwise

Description

Tests the interrupt to the CPU has occurred by reading the corresponding flag of UIPR register.

Example

```
Uint32 UtopEvent;
/* Tests the interrupt of the receive event */
UtopEvent = UTOP_intTest(UTOP_INT_RQ);
```

UTOP_read — *Reads the UTOPIA receive queue*

Function

Uint32 UTOP_read();

Arguments

none

Return Value

val — Value from the receive queue

Description

Reads data from the receive queue.

Example

```
Uint32 UtopData;
/* Reads data from the receive queue */
UtopData = UTOP_read();
```

UTOP_SUPPORT — *Compile-time constant*

Constant

UTOP_SUPPORT

Description

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

Note: The UTOP module is not supported on devices that do not have the UTOP peripheral.

Example

```
#if (UTOP_SUPPORT)
  /* user UTOP configuration /
#endif
```

UTOP_write

Writes to the UTOPIA transmit queue

Function

```
void UTOP_write(
    Uint32 val
);
```

Arguments

val Value to be written into transmit queue

Return Value

none

Description

Writes data into the transmit queue.

Example

```
Uint32 UtopData = 0x1111FFFF;
/* Writes data into the transmit queue */
UTOP_write(UtopData);
```

Chapter 22

VCP Module

This chapter describes the VCP module, lists the API functions and macros within the module, discusses how to use the VCP, and provides a VCP API reference section.

Topic **Page**

22.1 Overview

Currently, there is one TMS320C6000™ device with a Viterbi Co-processor (VCP): the TMS320C6416. The VCP should be serviced using the EDMA for most accesses, but the CPU must first configure the VCP control values. There are also a number of functions available to the CPU to monitor the VCP status and access decision and output parameter data.

Table 22–1 lists the configuration structures for use with the VCP functions. Table 22–2 lists the functions and constants available in the CSL VCP module.

Table 22–1. VCP Configuration Structures

Syntax	Type	Description	See page ...
VCP_BaseParams	S	Structure used to set basic VCP Parameters	22-8
VCP_ConfigIc	S	Structure containing the IC register values	22-9
VCP_Params	S	Structure containing all channel characteristics	22-10

Table 22–2. VCP APIs

Syntax	Type	Description	See page ...
VCP_calcSubBlocksSA	F	Calculates the sub-blocks within a frame for standalone mode	22-13
VCP_calcSubBlocksSP	F	Calculates the sub-frames and -blocks within a frame for shared processing mode	22-13
VCP_calcCountsSA	F	Calculates the number of elements for each data buffer to be transmitted to/from the VCP using the EDMA for standalone mode	22-14
VCP_calcCountsSP	F	Calculates the number of elements for each data buffer to be transmitted to/from the VCP using the EDMA for shared processing mode	22-14
VCP_calculateHd	F	Calculates hard decisions for shared processing mode	22-14
VCP_ceil	F	Ceiling function	22-15
VCP_DECISION_HARD	C	Value indicating hard decisions output	22-15

Note: F = Function; C = Constant

Table 22–2. VCP APIs (Continued)

Syntax	Type	Description	See page ...
VCP_DECISION_SOFT	C	Value indicating soft decisions output	22-15
VCP_deinterleaveExt	F	De-interleaves extrinsics data for shared processing mode	22-15
VCP_demuxInput	F	Demultiplexes input into two working data sets for shared processing mode	22-16
VCP_END_NATIVE	C	Value indicating native data format	22-17
VCP_END_PACKED32	C	Value indicating packed data format	22-17
VCP_errTest	F	Returns the error code	22-17
VCP_FLEN_MAX	F	Maximum frame length	22-17
VCP_genIc	F	Generates the `VCP_ConfigIc` struct based on the VCP parameters provided by the `VCP_Params` struct	22-17
VCP_genParams	F	Function used to set basic VCP Parameters	22-18
VCP_getBmEndian	F	Returns branch metrics data endian configuration	22-19
VCP_getIcConfig	F	Returns the IC values already programmed into the VCP	22-19
VCP_getMaxSm	F	Returns the final maximum state metric	22-20
VCP_getMinSm	F	Returns the final minimum state metric	22-20
VCP_getNumInFifo	F	Returns the number of symbols in the input FIFO	22-20
VCP_getNumOutFifo	F	Returns the number of symbols in the output FIFO	22-21
VCP_getSdEndian	F	Returns the soft decisions data configuration	22-21
VCP_getStateIndex	F	Returns the index of the final maximum state metric	22-21
VCP_getYamBit	F	Returns the Yamamoto bit result	22-22
VCP_icConfig	F	Stores the IC values into the VCP	22-22
VCP_icConfigArgs	F	Stores the IC values into the VCP using arguments	22-23

Note: F = Function; C = Constant

Table 22–2. VCP APIs (Continued)

Syntax	Type	Description	See page ...
VCP_interleaveExt	F	Interleaves extrinsics data for shared processing mode	22-24
VCP_normalCeil	F	Normalized ceiling function	22-24
VCP_pause	F	Pauses the VCP	22-25
VCP_RATE_1_2	C	Value indicating a rate of 1/2	22-25
VCP_RATE_1_3	C	Value indicating a rate of 1/3	22-25
VCP_RATE_1_4	C	Value indicating a rate of 1/4	22-25
VCP_reset	F	Resets the VCP	22-25
VCP_RLEN_MAX	C	Maximum reliability length	22-26
VCP_setBmEndian	F	Sets the branch metrics data endian configuration	22-26
VCP_setNativeEndian	F	Sets all data formats to be native (not packed data)	22-26
VCP_setPacked32Endian	F	Sets all data formats to be packed data	22-27
VCP_setSdEndian	F	Sets the soft decisions data configuration	22-27
VCP_start	F	Starts the VCP	22-27
VCP_statError	F	Returns the error status	22-28
VCP_statInFifo	F	Returns the input FIFO status	22-28
VCP_statOutFifo	F	Returns the output FIFO status	22-28
VCP_statPause	F	Returns the pause status	22-29
VCP_statRun	F	Returns the run status	22-29
VCP_statSymProc	F	Returns the Number of Symbols processed status bit	22-29
VCP_statWaitIc	F	Returns the input control status	22-30
VCP_stop	F	Stops the VCP	22-30
VCP_TRACEBACK_CONVERGENT	C	Value indicating convergent traceback mode	22-30
VCP_TRACEBACK_MIXED	C	Value indicating mixed traceback mode	22-30

Note: F = Function; C = Constant

Table 22–2. VCP APIs (Continued)

Syntax	Type	Description	See page ...
VCP_TRACEBACK_TAILED	C	Value indicating tailed traceback mode	22-30
VCP_unpause	F	Unpauses the VCP	22-31

Note: F = Function; C = Constant

22.1.1 Using the VCP

To use the VCP, you must first configure the control values, or IC values, that will be sent via the EDMA to program its operation. To do this, the `VCP_Params` structure is passed to `VCP_icConfig()`. `VCP_Params` contains all of the channel characteristics required to configure the VCP. This configuration function returns a pointer to the IC values that are to be sent using the EDMA. If desired, the configuration function can be bypassed and the user can generate each IC value independently using several *VCP_RMK* (make) macros that construct register values based on field values. In addition, the symbol constants may be used for the field values.

When operating in big endian mode the CPU must configure the format of all the data to be transferred to and from the VCP. This is accomplished by programming the VCP Endian register (`VCP_END`). Typically, the data will all be of the same format, either following the native element size (either 8-bit or 16-bit) or packed into a 32-bit word. This being the case, the values can be set using a single function call to either `VCP_nativeEndianSet()` or `VCP_packed32EndianSet()`. Alternatively, the data format of individual data types can be programmed with independent functions.

The user can monitor the status of the VCP during operation and also monitor error flags if there is a problem.

22.2 Macros

There are two types of VCP macros: those that access registers and fields, and those that construct register and field values. These are not required as all VCP configuring and monitoring can be done through the provided functions. These VCP functions make use of a number of macros.

Table 22–3 lists the VCP macros that access registers and fields, and Table 22–4 lists the VCP macros that construct register and field values. The macros themselves are found in Chapter 24, *Using the HAL Macros*.

The VCP module includes handle-based macros.

Table 22–3. VCP Macros that Access Registers and Fields

Macro	Description/Purpose	See page ...
VCP_ADDR(<REG>)	Register address	24-12
VCP_RGET(<REG>)	Returns the value in the peripheral register	24-18
VCP_RSET(<REG>,x)	Register set	24-20
VCP_FGET(<REG>,<FIELD>)	Returns the value of the specified field in the peripheral register	24-13
VCP_FSET(<REG>,<FIELD>,fieldval)	Writes *fieldval* to the specified field in the peripheral register	24-15
VCP_FSETS(<REG>,<FIELD>,<SYM>)	Writes the symbol value to the specified field in the peripheral	24-17
VCP_RGETA(addr,<REG>)	Gets register for a given address	24-19
VCP_RSETA(addr,<REG>,x)	Sets register for a given address	24-20
VCP_FGETA(addr,<REG>,<FIELD>)	Gets field for a given address	24-13
VCP_FSETA(addr,<REG>,<FIELD>, fieldval)	Sets field for a given address	24-16
VCP_FSETSA(addr,<REG>,<FIELD>, <SYM>)	Sets field symbolically for a given address	24-17

Table 22–4. VCP Macros that Construct Register and Field Values

Macro	Description/Purpose	See page ...
VCP_<REG>_DEFAULT	Register default value	24-21
VCP_<REG>_RMK()	Register make	24-23
VCP_<REG>_OF()	Register value of ...	24-22
VCP_<REG>_<FIELD>_DEFAULT	Field default value	24-24
VCP_FMK()	Field make	24-14
VCP_FMKS()	Field make symbolically	24-15
VCP_<REG>_<FIELD>_OF()	Field value of ...	24-24
VCP_<REG>_<FIELD>_<SYM>	Field symbolic value	24-24

22.3 Configuration Structure

VCP_BaseParams *Structure used to set basic TCP Parameters*

Structure VCP_BaseParams

Members

VCP_Rate	rate;	Code rate
Uint8	constLen;	Code rate
Uint16	frameLen;	Frame Length
Uint16	yamTh;	Yamamoto Threshold
Uint8	stateNum;	State Index
Uint8	decision;	Hard/soft Decision
Uint8	readFlag;	Output Parameter Read flag

Description This is the VCP base parameters structure used to set up the VCP programmable parameters. You create the object and pass it to the VCP_genParams() function which returns the VCP_Params structure. See the VCP_genParams() function.

Example

```
VCP_BaseParams vcpBaseParam0 = {
     3,    /* Rate  */
     9,   /*Constraint Length (K=5,6,7,8, OR 9)*/
    81,   /*Frame Length (FL)  */
     0,   /*Yamamoto Threshold (YAMT)*/
     0,   /*Stat Index to set to IMAXS (IMAXI) */
     0,   /*Output Hard Decision Type   */
     0    /*Output Parameters Read Flag   */
};
  …
VCP_genParams(&vcpBaseParam0, &vcpParam0);
```

VCP_ConfigIc

Structure containing the IC register values

Structure

```
typedef struct {
    Uint32 ic0;
    Uint32 ic1;
    Uint32 ic2;
    Uint32 ic3;
    Uint32 ic4;
    Uint32 ic5;
} VCP_ConfigIc;
```

Members

ic0 Input Configuration word 0 value

ic1 Input Configuration word 1 value

ic2 Input Configuration word 2 value

ic3 Input Configuration word 3 value

ic4 Input Configuration word 4 value

ic5 Input Configuration word 5 value

Description

This is the VCP input configuration structure that holds all of the configuration values that are to be transferred to the VCP via the EDMA. Though using the EDMA is highly recommended, the values can be written to the VCP using the CPU with the `VCP_icConfig()` function.

Example

```
extern VCP_Params *params;
VCP_ConfigIc *config;
...
VCP_genIc(params, config);
```

VCP_Params

Structure containing all channel characteristics

Structure

```
typedef struct {
    VCP_Rate    rate;
    Uint32      constLen;
    Uint32      poly0;
    Uint32      poly1;
    Uint32      poly2;
    Uint32      poly3;
    Uint32      yamTh;
    Uint32      frameLen;
    Uint32      relLen;
    Uint32      convDist;
    Uint32      maxSm;
    Uint32      minSm;
    Uint32      stateNum;
    Uint32      bmBuffLen;
    Uint32      decBuffLen;
    Uint32      traceBack;
    Uint32      readFlag;
    Uint32      decision;
    Uint32      numBranchMetrics;
    Uint32      numDecisions;
} VCP_Params;
```

Members

rate	The rate: 1/2, 1/3, 1/4 The available constants are: ❏ VCP_RATE_1_2 ❏ VCP_RATE_1_3 ❏ VCP_RATE_1_4
constLen	Constraint length
poly0	Polynomial 0
poly1	Polynomial 1
poly2	Polynomial 2
poly3	Polynomial 3
yamTh	Yamamoto Threshold value
frameLen	The number of symbols in a frame
relLen	Reliability length
convDist	Convergence distance
maxSm	Maximum initial state metric
minSm	Minimum initial state metric
stateNum	State index set to the maximum initial state metric
bmBuffLen	Branch metrics buffer length in input FIFO
decBuffLen	Decisions buffer length in output FIFO
traceBack	Traceback mode The available constants are: ❏ VCP_TRACEBACK_NONE ❏ VCP_TRACEBACK_TAILED ❏ VCP_TRACEBACK_MIXED ❏ VCP_TRACEBACK_CONVERGENT
readFlag	Output parameters read flag
decision	Decision selection: hard or soft The following constants are available: ❏ VCP_DECISION_HARD ❏ VCP_DECISION_SOFT
numBranchMetrics	Number of branch metrics per event
numDecisions	Number of decisions words per event

Description

This is the VCP parameters structure that holds all of the information concerning the user channel. These values are used to generate the appropriate input configuration values for the VCP and to program the EDMA.

Example

```
extern VCP_Params *params;
VCP_ConfigIc *config;
...
VCP_genIc(params, config);
...
```

22.4 Functions

VCP_calcSubBlockSA *Calculates sub-blocks for standalone processing*

Function void VCP_calcSubBlocksSA(VCP_Params *configParms);

Arguments configParms Configuration parameters

Return Value none

Description This function divides the data frames into sub-blocks for standalone processing mode.

Example
```
VCP_calcSubBlocksSA(configParms);
```

VCP_calcSubBlocksSP *Calculates sub-blocks for shared processing*

Function Uint32 VCP_calcSubBlocksSP(VCP_Params *configParms;

Arguments configParms Configuration parameters

Return Value numSubFrames Number of sub-frames

Description This function divides the data frames into sub-frames and sub-blocks for shared processing mode. The number of subframes into which the data frame was divided is returned.

Example
```
Uint32 numSubFrames;
  NumSubFrames = VCP_calcSubBlocksSP(configParms);
```

VCP_calcCountsSA *Calculates the count values for standalone processing*

Function Void VCP_calcCountsSA(VCP_Params *configParms);

Arguments configParms Configuration parameters

Return Value none

Description This function calculates all of the count values required to transfer all data to/from the VCP using the EDMA. This function is for standalone processing mode.

Example
```
VCP_calcCountsSA(configParms);
```

VCP_calcCountsSP *Calculates the count values for shared processing*

Function Void VCP_calcCountsSP(VCP_Params *configParms);

Arguments

configParms	Configuration parameters

Return Value none

Description This function calculates all of the count values required to transfer all data to/from the VCP using the EDMA. This function is for shared processing mode.

Example

```
VCP_calcCountsSP(configParms);
```

VCP_calculateHd *Calculate hard decisions*

Function

```
void VCP_calculateHd(
    const VCP_ExtrinsicData *restrict extrinsicsMap1,
    const VCP_ExtrinsicData *restrict apriori,
    const VCP_UserData *restrict channel_data,
    Uint32 *restrict hardDecisions,
    Uint16 numExt,
    Uint8  rate);
```

Arguments

extrinsicsMap1	Extrinsics data following MAP1 decode
apriori	Apriori data following MAP2 decode
channel_data	Input channel data
hardDecisions	Hard decisions
numExt	Number of extrinsics
rate	Channel rate

Return Value none

Description This function calculates the hard decisions following multiple MAP decodings in shared processing mode.

Example

```
<...Iterate through MAP1 and MAP2 decodes...>
void VCP_calculateHd(extrinsicsMap1, apriori,
      channel_data, hardDecisions, numExt, rate);
```

VCP_ceil *Ceiling function*

Function Uint32 VCP_ceil(Uint32 val, Uint32 pwr2);

Arguments

val	Value to be augmented
pwr2	The power of two by which val must be divisible

Return Value

ceilVal	The smallest number which when multiplied by 2^pwr2 is greater than val.

Description This function calculates the ceiling for a given value and a power of 2. The arguments follow the formula: ceilVal * 2^pwr2 = ceiling(val, pwr2).

Example

```
numSysPar = VCP_ceil((frameLen * rate), 4);
```

VCP_DECISION_HARD *Value indicating hard decisions output*

Constant VCP_DECISION_HARD

Description This constant allows selection of hard decisions output from the VCP.

VCP_DECISION_SOFT *Value indicating soft decisions output*

Constant VCP_DECISION_SOFT

Description This constant allows selection of soft decisions output from the VCP.

VCP_deinterleaveExt *De-interleave extrinsics data*

Function

```
Void VCP_deinterleaveExt(
    VCP_ExtrinsicData *restrict aprioriMap1,
    const VCP_ExtrinsicData *restrict extrinsicsMap2,
    const Uint16 *restrict interleaverTable,
    Uint32  numExt);
```

Arguments

aprioriMap1	Apriori data for MAP1 decode
extrinsicsMap2	Extrinsics data following MAP2 decode
interleaverTable	Interleaver data table
numExt	Number of Extrinsics

Return Value none

Description This function de-interleaves the MAP2 extrinsics data to generate apriori data for the MAP1 decode. This function is for use in performing shared processing.

Example

```
<...MAP 2 decode...>
VCP_deinterleaveExt(aprioriMap2, extrinsicsMap1,
                    interleaverTable, numExt);
<...MAP 1 decode...>
```

VCP_demuxInput *De-multiplexes the input data*

Function

```
Void VCP_demuxInput(Uint32 rate,
    Uint32 frameLen,
    const VCP_UserData *restrict input,
    const Uint16 *restrict interleaver,
    VCP_ExtrinsicData *restrict nonInterleaved,
    VCP_ExtrinsicData *restrict interleaved);
```

Arguments

rate	Channel rate
frameLen	Frame length
input	Input channel data
interleaver	Interleaver data table
nonInterleaved	Non-interleaved input data
interleaved	Interleaved input data

Return Value none

Description This function splits the input data into two working sets. One set contains the non-interleaved input data and is used with the MAP 1 decoding. The other contains the interleaved input data and is used with the MAP2 decoding. This function is used in shared processing mode.

Example

```
VCP_demuxInput(rate, frameLen, input,
      interleaver, nonInterleaved, interleaved);
```

VCP_END_NATIVE *Value indicating native endian format*

Constant VCP_END_NATIVE

Description This constant allows selection of the native format for all data transferred to and from the VCP. That is to say that all data is contiguous in memory with incrementing addresses.

VCP_END_PACKED32 *Value indicating little endian format within packed 32-bit words*

Constant VCP_END_PACKED32

Description This constant allows selection of the packed 32-bit format for data transferred to and from the VCP. That is to say that all data is packed into 32-bit words in little endian format and these words are contiguous in memory.

VCP_errTest *Returns the error code*

Function Uint32 VCP_errTest();

Arguments none

Return Value Error code Code error value

Description This function returns an ERR bit indicating what VCP error has occurred.

Example

```
/* check whether an error has occurred */
if (VCP_errTest()){
} /* end if */
```

VCP_FLEN_MAX *Maximum frame length*

Constant VCP_FLEN_MAX

Description This constant equals the maximum frame length programmable into the VCP.

VCP_genIc *Generates the VCP_ConfigIc struct*

Function

```
void VCP_genIc(
    VCP_Params *restrict configParms,
    VCP_ConfigIc *restrict configIc
)
```

Arguments

configParms	Pointer to Channel parameters structure
configIc	Pointer to Input Configuration structure

Return Value none

Description This function generates the required input configuration values needed to program the VCP based on the parameters provided by configParms.

Example

```
extern VCP_Params *params;
VCP_ConfigIc *config;
...
VCP_genIc(params, config);

...
```

VCP_genParams *Sets basic VCP Parameters*

Function

```
void VCP_genParams(
    VCP_BaseParams *configBase,
    VCP_Params *configParams
)
```

Arguments

configBase Pointer to VCP_BaseParams structure

configParams Output VCP_Params structure pointer

Return Value none

Description This function calculates the TCP parameters based on the input VCP_BaseParams object values and set the values to the output VCP_Params parameters structure.

The calculated parameters are:

Polynomial constants:

- G0-G3 (POLY1-POLY3)
- Traceback (TB)
- Convergence Distance (CD)
- Reliability Length (R)
- Decision Buffer Length (SYMR +1)
- Branch Metric Buffer Length (SYMX +1)
- Max Initial Metric State (IMAXS)
- Min Initial Metric State (IMINS)

Example

```
VCP_Params vcpParam0;
VCP_genParams(&vcpBaseParam0, &vcpParam0);
```

VCP_getBmEndian *Returns branch metrics data endian configuration*

Function Uint32 VCP_getBmEndian();

Arguments none

Return Value Endian Endian setting for branch metrics data

Description This function returns the value programmed into the END register for the branch metrics data indicating whether the data is in its native 8-bit format (‘1’) or consists of values packed in little endian format into 32-bit words (‘0’). This should always be ‘0’ for little endian operation.

See also `VCP_setBmEndian, VCP_setNativeEndian, VCP_setPacked32Endian, VCP_getSdEndian, VCP_setSdEndian.`

Example

```
If (VCP_getBmEndian()){
  ...
} /* end if */
```

VCP_getIcConfig *Returns the IC values already programmed into the VCP*

Function void VCP_getIcConfig(VCP_ConfigIc *config)

Arguments config Pointer to Input Configuration structure

Return Value none

Description This function reads the input configuration values currently programmed into the VCP.

Example

```
VCP_ConfigIc *config;
...
VCP_getIcConfig(config);

...
```

VCP_getMaxSm *Returns the final maximum state metric*

Function Uint32 VCP_getMaxSm();

Arguments none

Return Value State Metric Final maximum state metric

Description This function returns the final maximum state metric after the VCP has completed its decoding.

See also `VCP_getMinSm`.

Example

```
Uint32 maxSm;
MaxSm = VCP_getMaxSm();
```

VCP_getMinSm *Returns the final minimum state metric*

Function Uint32 VCP_getMinSm();

Arguments none

Return Value State Metric Final minimum state metric

Description This function returns the final minimum state metric after the VCP has completed its decoding.

See also `VCP_getMaxSm`.

Example

```
Uint32 minSm;
MinSm = VCP_getMinSm();
```

VCP_getNumInFifo *Returns the number of symbols in the input FIFO*

Function Uint32 VCP_getNumInFifo();

Arguments none

Return Value count The number of symbols currently in the input FIFO

Description this function returns the number of symbols currently in the input FIFO.

Example

```
numSym = VCP_getNumInFifo();
```

VCP_getNumOutFifo *Returns the number of symbols in the output FIFO*

Function Uint32 VCP_getNumOutFifo();

Arguments none

Return Value count The number of symbols currently in the output FIFO

Description this function returns the number of symbols currently in the output FIFO.

Example
```
numSym = VCP_getNumOutFifo();
```

VCP_getSdEndian *Returns soft decision data endian configuration*

Function Uint32 VCP_getSdEndian();

Arguments none

Return Value Endian Endian setting for soft decision data

Description This function returns the value programmed into the VCP_END register for the soft decision data indicating whether the data is in its native 16-bit format (‘1’) or consists of values packed in little endian format into 32-bit words (‘0’). This should always be ‘0’ for little endian operation.

See also `VCP_setSdEndian, VCP_setNativeEndian, VCP_setPacked32Endian.`

Example
```
If (VCP_getBmEndian()){
  ...
} /* end if */
```

VCP_getStateIndex *Returns index of final maximum state metric*

Function Uint32 VCP_getStateIndex();

Arguments none

Return Value Index Index of final max state metric

Description This function returns an index for the final maximum state metric.

Example
```
Uint32 index;
  ...
index = VCP_getStateIndex();
```

VCP_getYamBit — *Returns the Yamamoto bit result*

Function Uint32 VCP_getYamBit();

Arguments none

Return Value bit Yamamoto bit result

Description Returns the value of the Yamamoto bit after the VCP decoding.

Example

```
Uint32 yamBit;
YamBit = VCP_getYamBit();
```

VCP_icConfig — *Stores the IC values into the VCP*

Function void VCP_icConfig(VCP_ConfigIc *config)

Arguments Config Pointer to Input Configuration structure

Return Value none

Description This function stores the input configuration values currently programmed into the VCP. This is not the recommended means by which to program the VCP, as it is more efficient to transfer the IC values using the EDMA, but can be used in test code.

Example

```
extern VCP_Params *params;
VCP_ConfigIc *config;
...
VCP_genIc(params, config);
VCP_icConfig(config);
...
```

VCP_icConfigArgs *Stores the IC values into the VCP using arguments*

Function

```
void VCP_icConfigArgs(
    Uint32 ic0, Uint32 ic1, Uint32 ic2,
    Uint32 ic3, Uint32 ic4, Uint32 ic5
)
```

Arguments

ic0	Input Configuration word 0 value
ic1	Input Configuration word 1 value
ic2	Input Configuration word 2 value
ic3	Input Configuration word 3 value
ic4	Input Configuration word 4 value
ic5	Input Configuration word 5 value

Return Value

none

Description

This function stores the input configuration values currently programmed into the VCP. This is not the recommended means by which to program the VCP, as it is more efficient to transfer the IC values using the EDMA, but can be used in test code.

Example

```
VCP_icConfigArgs(
  0x00283200           /* IC0  */
  0x00270000           /* IC1  */
  0x00080118           /* IC2  */
  0x001E0014           /* IC3  */
  0x00000000           /* IC4  */
  0x00000002           /* IC5  */
);
```

VCP_interleaveExt *Interleaves extrinsics data*

Function

Void VCP_interleaveExt(
VCP_ExtrinsicData *restrict aprioriMap2,
const VCP_ExtrinsicData *restrict extrinsicsMap1,
const Uint16 *restrict interleaverTable,
Uint32 numExt);

Arguments

aprioriMap2	Apriori data for MAP2 decode
extrinsicsMap1	Extrinsics data following MAP1 decode
interleaverTable	Interleaver data table
numExt	Number of Extrinsics

Return Value

none

Description

This function interleaves the MAP1 extrinsics data to generate apriori data for the MAP2 decode. This function is used in performing shared processing.

Example

```
<...MAP 1 decode...>
VCP_interleaveExt(aprioriMap2, extrinsicsMap1,
                  InterleaverTable, numExt);
<...MAP 2 decode...>
```

VCP_normalCeil *Normalized ceiling function*

Function

Uint32 VCP_normalCeil(Uint32 val1, Uint32 val2) ;

Arguments

val1	Value to be augmented
val2	Value by which val1 must be divisible

Return Value

ceilVal	The smallest number greater than or equal to val1 that is divisible by val2.

Description

This function returns the smallest number greater than or equal to val1 that is divisible by val2.

Example

```
winSize = VCP_normalCeil(winSize, numSlidingWindow);
```

VCP_pause

Pauses VCP by writing a pause command in VCP_EXE

Function void VCP_pause();

Arguments none

Return Value none

Description This function pauses the VCP by writing a pause command in the VCP_EXE register. See also `VCP_start()`, `VCP_unpause()`, and `VCP_stop()`.

Example

```
VCP_pause();
```

VCP_RATE_1_2

Value indicating a rate of 1/2

Constant VCP_RATE_1_2

Description This constant allows selection of a rate of 1/2.

VCP_RATE_1_3

Value indicating a rate of 1/3

Constant VCP_RATE_1_3

Description This constant allows selection of a rate of 1/3.

VCP_RATE_1_4

Value indicating a rate of 1/4

Constant VCP_RATE_1_4

Description This constant allows selection of a rate of 1/4.

VCP_reset

Resets VCP registers to default values

Function Uint32 VCP_reset();

Arguments none

Return Value none

Description This function sets all of the VCP control registers to their default values.

Example

```
VCP_reset();
```

VCP_RLEN_MAX *Maximum reliability length*

Constant VCP_RLEN_MAX

Description This constant equals the maximum reliability length programmable into the VCP.

VCP_setBmEndian *Sets the branch metrics data endian configuration*

Function

```
Void VCP_setBmEndian(
Uint32 bmEnd
);
```

Arguments

bmEnd — Endian setting for branch metrics data
The following constants can be used:
- ❑ VCP_END_NATIVE
- ❑ VCP_END_PACKED32

Return Value none

Description This function programs the VCP to view the format of the branch metrics data as either native 8-bit format (‘1’) or values packed into 32-bit words in little endian format (‘0’). This should always be ‘0’ for little endian operation.

See also `VCP_getBmEndian, VCP_setNativeEndian, VCP_setPacked32Endian.`

Example

```
VCP_setBmEndian(VCP_END_PACKED32);
```

VCP_setNativeEndian *Sets all data formats to native (not packed data)*

Function void VCP_setNativeEndian();

Arguments none

Return Value none

Description This function programs the VCP to view the format of all data as native 8-/16-bit format. This should only be used when running in big endian mode.

See also `VCP_setPacked32Endian, VCP_getBmEndian, VCP_getSdEndian, VCP_setBmEndian, VCP_setSdEndian.`

Example

```
VCP_setNativeEndian();
```

VCP_setPacked32Endian *Sets all data formats to packed data*

Function void VCP_setPacked32Endian();

Arguments none

Return Value none

Description This function programs the VCP to view the format of all data as packed data in 32-bit words. This should always be used when running in little endian mode and should be used in big endian mode only if the CPU is formatting the data.

See also `VCP_setNativeEndian, VCP_getBmEndian, VCP_getSdEndian, VCP_setBmEndian, VCP_setSdEndian.`

Example

```
VCP_setPacked32Endian();
```

VCP_setSdEndian *Sets soft decision data endian configuration*

Function

```
Void VCP_setSdEndian
    (Uint32 sdEnd
);
```

Arguments

SdEnd — Endian setting for soft decision data
The following constants can be used:
- ❏ VCP_END_NATIVE
- ❏ VCP_END_PACKED32

Return Value none

Description This function programs the VCP to view the format of the soft decision data as either native 8-bit format (‘1’) or values packed into 32-bit words in little endian format (‘0’). This should always be ‘0’ for little endian operation.

See also `VCP_getSdEndian, VCP_setNativeEndian, VCP_setPacked32Endian.`

Example

```
VCP_setSdEndian(VCP_END_PACKED32);
```

VCP_start *Starts VCP by writing a start command in VCP_EXE*

Function void VCP_start();

Arguments none

Return Value none

Description This function starts the VCP by writing a start command to the VCP_EXE register. See also `VCP_pause()`, `VCP_unpause()`, and `VCP_stop()`.

Example

```
VCP_start();
```

VCP_statError

Returns the error status

Function Uint32 VCP_statError();

Arguments none

Return Value Error status Boolean indication of any error

Description This function returns a Boolean value indicating whether any VCP error has occurred.

Example

```
/* check whether an error has occurred */
if (VCP_statError()){
  error = VCP_errTest();
} /* end if */
```

VCP_statInFifo

Returns the input FIFO status

Function Uint32 VCP_statInFifo();

Arguments none

Return Value Empty Flag Flag indicating FIFO empty

Description This function returns the input FIFO's empty status flag. A '1' indicates that the input FIFO is empty and a '0' indicates it is not empty.

Example

```
If (VCP_statInFifo()){
 ...
} /* end if */
```

VCP_statOutFifo

Returns the output FIFO status

Function Uint32 VCP_statOutFifo();

Arguments none

Return Value Empty Flag Flag indicating FIFO full

Description This function returns the output FIFO's full status flag. A '1' indicates that the output FIFO is full and a '1' indicates it is not full.

Example

```
If (VCP_statOutFifo()){
  ...
} /* end if */
```

VCP_statPause *Returns pause status*

Function Uint32 VCP_statPause();

Arguments none

Return Value Status Boolean status

Description This function returns the PAUSE bit status indicating whether the VCP is paused or not.

Example

```
/* pause the VCP */
VCP_pause();
/* wait for pause to take place */
while (!VCP_statPause());
```

VCP_statRun *Returns the run status*

Function Uint32 VCP_statRun();

Arguments none

Return Value Status Boolean status

Description This function returns the RUN bit status indicating whether the VCP is running.

Example

```
/* start the VCP */
VCP_start();
/* check that the VCP is running */
while (!VCP_statRun());
```

VCP_statSymProc *Returns number of symbols processed*

Function Uint32 VCP_statSymProc();

Arguments none

Return Value count The number of symbols processed

Description This function returns the NSYMPROC status bit of the VCP.

Example

```
numSym = VCP_statSymProc();
```

VCP_statWaitIc — *Returns input control status*

Function Uint32 VCP_statWaitIc();

Arguments none

Return Value Status Boolean status

Description This function returns the WIC bit status indicating whether the VCP is waiting to receive new IC values.

Example

```
If (statWaitIc()){
...
} /* end if */
```

VCP_stop — *Stops the VCP by writing a stop command in VCP_EXE*

Function void VCP_stop();

Arguments none

Return Value none

Description This function stops the VCP by writing a stop command to the VCP_EXE register. See also `VCP_pause()`, `VCP_unpause()`, and `VCP_start()`.

Example

```
VCP_stop();
```

VCP_TRACEBACK_CONVERGENT — *Value indicating convergent tracebackmode*

Constant VCP_TRACEBACK_CONVERGENT

Description This constant allows selection of convergent traceback mode.

VCP_TRACEBACK_MIXED — *Value indicating mixed traceback mode*

Constant VCP_TRACEBACK_MIXED

Description This constant allows selection of mixed traceback mode.

VCP_TRACEBACK_TAILED — *Value indicating tailed traceback mode*

Constant VCP_TRACEBACK_TAILED

Description This constant allows selection of tailed traceback mode.

VCP_unpause

Un-pauses the VCP by writing an unpause command in VCP_EXE

Function void VCP_unpause();

Arguments none

Return Value none

Description This function un-pauses the VCP by writing an un-pause command to the VCP_EXE register. See also `VCP_pause()`, `VCP_start()`, and `VCP_stop()`.

Example

```
VCP_unpause();
```

Chapter 23

XBUS Module

This chapter describes the XBUS module, lists the API functions and macros within the module, discusses how to use the XBUS device, and provides an XBUS API reference section.

Topic **Page**

23.1 Overview

This module has a simple API for configuring the XBUS registers. The XBUS may be configured by passing an `XBUS_CONFIG` structure to `XBUS_Config()` or by passing register values to the `XBUS_ConfigArgs()` function.

Table 23–1 lists the configuration structure for use with the XBUS functions. Table 23–2 lists the functions and constants available in the CSL DMA module.

Table 23–1. XBUS Configuration Structure

Syntax	Type	Description	See page ...
XBUS_Config	S	XBUS configuration structure	23-4

Table 23–2. XBUS APIs

Syntax	Type	Description	See page ...
XBUS_config	F	Configures entry for XBUS configuration structure	23-5
XBUS_configArgs	F	Configures entry for XBUS registers	23-5
XBUS_getConfig	F	Returns the current XBUS configuration structure	23-7
XBUS_SUPPORT	C	Compile time constant	23-7

Note: F = Function; C = Constant

23.2 Macros

There are two types of XBUS macros: those that access registers and fields, and those that construct register and field values.

Table 23–3 lists the XBUS macros that access registers and fields, and Table 23–4 lists the XBUS macros that construct register and field values. The macros themselves are found in Chapter 24, *Using the HAL Macros*.

XBUS macros are not handle-based.

Table 23–3. XBUS Macros that Access Registers and Fields

Macro	Description/Purpose	See page ...
XBUS_ADDR(<REG>)	Register address	24-12
XBUS_RGET(<REG>)	Returns the value in the peripheral register	24-18
XBUS_RSET(<REG>,x)	Register set	24-20
XBUS_FGET(<REG>,<FIELD>)	Returns the value of the specified field in the peripheral register	24-13
XBUS_FSET(<REG>,<FIELD>,fieldval)	Writes *fieldval* to the specified field in the peripheral register	24-15
XBUS_FSETS(<REG>,<FIELD>,<SYM>)	Writes the symbol value to the specified field in the peripheral	24-17
XBUS_RGETA(addr,<REG>)	Gets register for a given address	24-19
XBUS_RSETA(addr,<REG>,x)	Sets register for a given address	24-20
XBUS_FGETA(addr,<REG>,<FIELD>)	Gets field for a given address	24-13
XBUS_FSETA(addr,<REG>,<FIELD>, fieldval)	Sets field for a given address	24-16
XBUS_FSETSA(addr,<REG>,<FIELD>, <SYM>)	Sets field symbolically for a given address	24-17

Table 23–4. XBUS Macros that Construct Register and Field Values

Macro	Description/Purpose	See page ...
XBUS_<REG>_DEFAULT	Register default value	24-21
XBUS_<REG>_RMK()	Register make	24-23
XBUS_<REG>_OF()	Register value of ...	24-22
XBUS_<REG>_<FIELD>_DEFAULT	Field default value	24-24
XBUS_FMK()	Field make	24-14
XBUS_FMKS()	Field make symbolically	24-15
XBUS_<REG>_<FIELD>_OF()	Field value of ...	24-24
XBUS_<REG>_<FIELD>_<SYM>	Field symbolic value	24-24

23.3 Configuration Structure

XBUS_Config *XBUS configuration structure*

Structure XBUS_Config

Members

Uint32 xbgc	Expansion Bus global control register value
Uint32 xce0ctl	XCE0 space control register value
Uint32 xce1ctl	XCE1 space control register value
Uint32 xce2ctl	XCE2 space control register value
Uint32 xce3ctl	XCE3 space control register value
Uint32 xbhc	Expansion Bus host port interface control register value
Uint32 xbima	Expansion Bus internal master address register value
Uint32 xbea	Expansion Bus external address register value

Description This is the XBUS configuration structure used to set up an XBUS configuration. You create and initialize this structure then pass its address to the `XBUS_config()` function.

Example

```
XBUS_Config xbusCfg = {
    0x00000000,  /*  Global Control Register(XBGC)  */
    0xFFFF3F23,  /*  XCE0 Space Control Register(XCE0CTL)   */
    0xFFFF3F23,  /*  XCE1 Space Control Register(XCE1CTL)   */
    0xFFFF3F23,  /*  XCE2 Space Control Register(XCE2CTL)   */
    0xFFFF3F23,  /*  XCE3 Space Control Register(XCE3CTL)   */
    0x00000000,  /*  XBUS HPI Control Register(XBHC)  */
    0x00000000,  /*  XBUS Internal Master Address
                      Register(XBIMA) */
    0x00000000   /*  XBUS External Address Register(XBEA)  */
};
.
.
XBUS_config(&xbusCfg);
```

23.4 Functions

XBUS_config — *Establishes XBUS configuration structure*

Function

```
void XBUS_config(
    XBUS_Config *config
);
```

Arguments

config Pointer to an initialized configuration structure

Return Value

none

Description

Sets up the XBUS using the configuration structure. The values of the structure are written to the XBUS registers.

Example

```
XBUS_Config xbusCfg = {
    0x00000000,  /*  Global Control Register(XBGC)  */
    0xFFFF3F23,  /*  XCE0 Space Control Register(XCE0CTL)   */
    0xFFFF3F23,  /*  XCE1 Space Control Register(XCE1CTL)   */
    0xFFFF3F23,  /*  XCE2 Space Control Register(XCE2CTL)   */
    0xFFFF3F23,  /*  XCE3 Space Control Register(XCE3CTL)   */
    0x00000000,  /*  XBUS HPI Control Register(XBHC)  */
    0x00000000,  /*  XBUS Internal Master Address
                      Register(XBIMA)  */
    0x00000000   /*  XBUS External Address Register(XBEA)  */
};
XBUS_config(&xbusCfg);
```

XBUS_configArgs — *Establishes XBUS register value*

Function

```
void XBUS_configArgs(
    Uint32 xbgc,
    Uint32 xce0ctl,
    Uint32 xce1ctl,
    Uint32 xce2ctl,
    Uint32 xce3ctl,
    Uint32 xbhc,
    Uint32 xbima,
    Uint32 xbea
);
```

Arguments

xbgc Expansion Bus global control register value

xce0ctl	XCE0 space control register value
xce1ctl	XCE1 space control register value
xce2ctl	XCE2 space control register value
xce3ctl	XCE3 space control register value
xbhc	Expansion Bus host port interface control register value
xbima	Expansion Bus internal master address register value
xbea	Expansion Bus external address register value

Return Value none

Description Sets up the XBUS using the register values passed in. The register values are written to the XBUS registers.

Example

```
xbgc = 0x00000000;
xce0ctl = 0xFFFF3F23;
xce1ctl = 0xFFFF3F23;
xce2ctl = 0xFFFF3F23;
xce3ctl = 0xFFFF3F23;
xbhc = 0x00000000;
xbima = 0x00000000;
xbea = 0x00000000;
XBUS_configArgs(
    xbgc,
    xce0ctl,
    xce1ctl,
    xce2ctl,
    xce3ctl,
    xbhc,
    xbima,
    xbea
);
```

XBUS_getConfig *Gets XBUS current configuration value*

Function

```
void XBUS_getConfig(
    XBUS_Config *config
);
```

Arguments

config Pointer to a configuration structure

Return Value

none

Description

Get XBUS current configuration value.

Example

```
XBUS_config xbusCfg;
XBUS_getConfig(&xbusCfg);
```

XBUS_SUPPORT *Compile time constant*

Constant

XBUS_SUPPORT

Description

The compile time constant has a value of 1 if the device supports the XBUS module, and 0 otherwise. You are not required to use this constant.

Example

```
#if (XBUS_SUPPORT)
  /* user XBUS configuration */
#endif
```

Chapter 24

Using the HAL Macros

This chapter describes the hardware abstraction layer (HAL), gives a summary of the HAL macros, discusses RMK macros and macro token pasting, and provides a HAL macro reference section.

Topic **Page**

24.1 Introduction

The chip support library (CSL) has two layers: the service layer and the hardware abstraction layer, or HAL for short. The service layer contains the API functions, data types, and constants as defined in the various chapters of this reference guide. The HAL is made up of a set of header files that define an orthogonal set of C macros and constants which abstract registers and bit-fields into symbols.

24.1.1 HAL Macro Symbols

These HAL macro symbols define memory-mapped register addresses, register bit-field mask and shift values, symbolic names for bit-field values, and access macros for reading/writing registers and individual bit-fields. In other high-level OS environments, HAL usually refers to a set of functions that completely abstract hardware. In the context of the CSL, the abstraction is limited to processor-dependent changes of register/bit-field definitions. For example, if a bit-field changes width from one chip to another, this is reflected in the HAL macros. If a memory-mapped register is specific to a chip, the register is described in the HAL file with a condition. For example, the memory-mapped register SDEXT (EMIF register) is supported only by 6211 and 6711 devices, and the register description is set for these devices with a condition access. Devices other than 6211/6711 cannot access the abstract macros related to the SDEXT register.

Prior to the HAL definition, almost all application programmers found themselves defining a HAL in one form or another. Users would go through and add many symbols (*#define*s) to map registers, and they would define bit-field positions and values. Consequently, that process generated a large development time for programmers, all with their own HAL macros and no standardization. With the development of the CSL, TI has generated a set of HAL macros and made it available to all users in order to make peripheral configuration easy. The HAL macros add a level of compatibility and standardization and, more importantly, reduce development time.

24.1.2 HAL Header Files

The HAL macros are defined in the HAL header file (e.g., csl_dmahal.h, csl_mcbsphal.h, etc.) The user does not directly include these files; instead, the service layer header file is included, which indirectly includes the HAL file. For example, if the DMA HAL file is needed (csl_dmahal.h), include csl_dma.h, which will indirectly include csl_dmahal.h.

The HAL is nothing more than a large set of C macros; there is no compiled C code involved. In an application where only the HAL gets used, the result will be that zero CSL library code gets linked in.

24.1.3 HAL Macro Summary

TMS320C6000™ CSL macros can be divided into two functionality groups:

- **Macros that access registers and fields** (set, get). These macros are implemented at the beginning of the HAL files and include:
  - Macro for reading a register
  - Macro for writing to a register
  - Macro that returns the address of a memory-mapped register
  - Macro for inserting a field value into a register
  - Macro for extracting a field value from a register
  - Macro for inserting a field value into a register using a symbolic name
  - Variations of the above for handle-based registers
  - Variations of the above for given register addresses
- **Macros that construct register and field values**. These macros are register-specific and implemented for each value. They include:
  - Macro constant for the default value of a register
  - Macro that constructs register values based on field values
  - Macro constant for the default value of a register field
  - Macro that constructs a field value
  - Macro that constructs a field value given a symbolic constant

24.2 Generic Macro Notation and Table of Macros

Table 24–1 lists the macros defined in the HAL using the following generic notation:

- <PER> = placeholder for peripheral (module) name: DMA, MCBSP, IRQ, etc.
- <REG> = placeholder for a register name: PRICTL, SPCR, AUXCTL, etc.
- <FIELD> = placeholder for a field name: PRI, STATUS, XEMPTY, etc.
- <SYM> = placeholder for a value name: ENABLE, YES, HIGH, etc.

<PER> represents a placeholder for the peripheral (module) name; i.e., DMA, MCBSP, etc. When the table lists something like <PER>_ADDR, it actually represents a whole set of macros defined in the different modules: DMA_ADDR(…), MCBSP_ADDR(…), etc. Likewise, whenever <REG> is used, it is a placeholder for a register name. For example, <PER>_<REG>_DEFAULT represents a set of macros including DMA_AUXCTL_DEFAULT, MCBSP_SPCR_DEFAULT, TIMER_CTL_DEFAULT, etc. There are also field name place holders such as in the macro <PER>_<REG>_<FIELD>_DEFAULT. In this case it represents a set of macros including: DMA_PRICTL_PRI_DEFAULT, MCBSP_SPCR_GRST_DEFAULT, etc.

Table 24–1. CSL HAL Macros

HAL Macro Type	Purpose	See page ...
<PER>_ADDR	Register Address	24-12
<PER>_ADDRH	Register Address For Given Handle	24-12
<PER>_CRGET	Gets the Value of CPU Register	24-12
<PER>_CRSET	Sets the Value of CPU Register	24-13
<PER>_FGET	Field Get	24-13
<PER>_FGETA	Field Get Given Address	24-13
<PER>_FGETH	Field Get For Given Handle	24-14
<PER>_FMK	Field Make	24-14
<PER>_FMKS	Field Make Symbolically	24-15
<PER>_FSET	Field Set	24-15
<PER>_FSETA	Field Set Given Address	24-16
<PER>_FSETH	Field Set For Given Handle	24-16
<PER>_FSETS	Field Set Symbolically	24-17
<PER>_FSETSA	Field Set Symbolically For Given Address	24-17
<PER>_FSETSH	Field Set Symbolically For Given Handle	24-18
<PER>_RGET	Register Get	24-18
<PER>_RGETA	Register Get Given Address	24-19
<PER>_RGETH	Register Get For Given Handle	24-19
<PER>_RSET	Register Set	24-20
<PER>_RSETA	Register Set Given Address	24-20
<PER>_RSETH	Register Set For Given Handle	24-21
<PER>_<REG>_DEFAULT	Register Default Value	24-21
<PER>_<REG>_OF	Register Value Of …	24-22
<PER>_<REG>_RMK	Register Make	24-23
<PER>_<REG>_<FIELD>_DEFAULT	Field Default Value	24-24
<PER>_<REG>_<FIELD>_OF	Field Value Of …	24-24
<PER>_<REG>_<FIELD>_<SYM>	Field Symbolic Value	24-24

24.3 General Comments Regarding HAL Macros

This section contains some general comments of interest regarding the HAL macros.

24.3.1 Right-Justified Fields

Whenever field values are referenced, they are always right-justified. This makes it easier to deal with them and it also adds some processor independence. To illustrate, consider the following:

Assume that you have a register (MYREG) in a peripheral named MYPER with a field that spans bits 17 to 21 – a 5-bit field named (MYFIELD). Also assume that this field can take on three valid values, 00000b = V1, 01011b = V2, and 11111b = V3. It will look like this:

MYREG:

If you wanted to extract this field, you would first mask the register value with 0x003E0000 then right-shift it by 17 bits. This would give the right-justified field value.

If you start with the right justified field value and want to create the in-place field value, you would first left-shift it by 17 bits then mask it with 0x003E0000.

If we had HAL macros for this hypothetical register, then we would have a MYPER_FGET(MYREG, MYFIELD) macro that would return the MYFIELD value right-justified. We would also have the MYPER_FSET(MYREG, MYFIELD, x) macro that accepts a right-justified field value and inserts it into the register.

All of the FGET type of macros return the right-justified field value and all of the FSET type of macros take a right-justified field value as an argument. The FMK and RMK macros also deal with right-justified field values.

24.3.2 _OF Macros

The HAL defines a set of ***value-of*** macros for registers and fields:

```
<PER>_<REG>_OF(x)
<PER>_<REG>_<FIELD>_OF(x)
```

These macros serve the following two purposes:

- ❏ They typecast the argument
- ❏ They make code readable

Typecasting the Argument

The macros do nothing more than return the original argument but with a typecast.

```
#define <PER>_<REG>_OF(x)     ((Uint32)(x))
```

So, you could pass just about anything as an argument and it will get typecasted into a Uint32.

Making Code More Readable

The second purpose of these macros is to make code more readable. When you are assigning a value to a register or field, it may not be clear what you are assigning to. However, if you enclose the value with an _OF() macro, then it becomes perfectly clear what the value is.

Consider the following example where a DMA configuration structure is being statically initialized with hard-coded values. You can see from the example that it is not very clear what the values mean.

```
/* create a config structure using hard coded values */
DMA_Config cfg = {
  0x10002050,
  0x00000080,
  (Uint32)buffa,
  (Uint32)buffb,
  0x00010008
);
```

However, using the _OF() macros, the code now becomes clear. The above code and the below code both do the same thing. Also notice that the _OF() macros help out by eliminating the need to do manual typecasts.

```
/* create a config structure using the _OF() macros */
DMA_Config cfg = {
  DMA_PRICTL_OF(0x10002050),
  DMA_SECCTL_OF(0x00000080),
  DMA_SRC_OF(buffa),
  DMA_DST_OF(buffb),
  DMA_XFRCNT_OF(0x00010008)
);
```

Every register has an _OF() macro:

- DMA_PRICTL_OF(x)
- DMA_AUXCTL_OF(x)
- MCBSP_SPCR_OF(x)
- TIMER_PRD_OF(x)
- etc…

The same principle applies for field values. Every field has an _OF() macro defined for it:

- DMA_PRICTL_ESIZE_OF(x)
- DMA_PRICTL_PRI_OF(x)
- DMA_AUXCTL_CHPRI_OF(x)
- MCBSP_SPCR_DLB_OF(x)
- etc…

The field _OF() macros are generally used with the RMK macros. However, they are also useful when a field is very wide and it is not practical to #define a symbol for every value the field could take on.

24.3.3 RMK Macros

This set of macros allows you to create or make a value suitable for a particular register by specifying its individual field values. It basically shifts and masks all of the field values then ORs them together bit-wise to form a value. No writes are actually performed, only a value is returned.

The RMK macros take an argument for each writable field and they are passed in the order of most significant field first down to the least-significant field.

```
<PER>_<REG>_RMK(field_ms,...,field_ls)
```

For illustrative purposes, we will pick a register that does not have too many fields, such as the MCBSP multichannel control register, or MCR. Here is the MCR register comment header pulled directly from the MCBSP HAL file:

```
/*******************************************************\
* _____________________
* |                   |
* |  M C R            |
* |___________________|
*
* MCR0  – serial port 0 multichannel control register
* MCR1  – serial port 1 multichannel control register
* MCR2  – serial port 2 multichannel control register (1)
*
* (1) only supported on devices with three serial ports
*
* FIELDS (msb –> lsb)
* (rw) XPBBLK
* (rw) XPABLK
* (r)  XCBLK
* (rw) XMCM
* (rw) RPBBLK
* (rw) RPABLK
* (r)  RCBLK
* (rw) RMCM
*
\*******************************************************/
```

Out of the eight fields, only six are writable; hence, the RMK macro takes six arguments.

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

This macro will take each of the field values xpbblk to rmcm and form a 32-bit value. There are several ways you could use this macro each with a differing level of readability.

You could just hardcode the field values

```
x = MCBSP_MCR_RMK(1,0,0,1,0,1);
```

Or you could use the field value symbols

```
x = MCBSP_MCR_RMK(
  MCBSP_MCR_XPBBLK_SF1,
  MCBSP_MCR_XPABLK_SF0,
  MCBSP_MCR_XMCM_DISXP,
  MCBSP_MCR_RPBBLK_SF3,
  MCBSP_MCR_RPABLK_SF0,
  MCBSP_MCR_RMCM_CHENABLE
);
```

As you can see, the second method is much easier to understand and, in the long run, will be much easier to maintain.

Another consideration is when you use a variable for one of the field value arguments. Let's say that the XMCM argument is based on a variable in your program.

Just like before, but with the variable

```
x = MCBSP_MCR_RMK(
  MCBSP_MCR_XPBBLK_SF1,
  MCBSP_MCR_XPABLK_SF0,
  myVar,
  MCBSP_MCR_RPBBLK_SF3,
  MCBSP_MCR_RPABLK_SF0,
  MCBSP_MCR_RMCM_CHENABLE
);
```

Now use the field _OF() macro

```
x = MCBSP_MCR_RMK(
  MCBSP_MCR_XPBBLK_SF1,
  MCBSP_MCR_XPABLK_SF0,
  MCBSP_MCR_XMCM_OF(myVar),
  MCBSP_MCR_RPBBLK_SF3,
  MCBSP_MCR_RPABLK_SF0,
  MCBSP_MCR_RMCM_CHENABLE
);
```

In the first method, it's a little unclear what *myVar* is; however, in the second method, it's very clear because of the _OF() macro.

One thing that needs to be re-emphasized is the fact that the RMK macros do not write to anything; they simply return a value. As a matter of fact, if you used all symbolic constants for the field values, then the whole macro resolves down to a single integer from the compilers standpoint. The RMK macros may be used anywhere in your code, in static initializers, function arguments, arguments to other macros, etc.

24.3.4 Macro Token Pasting

The HAL macros rely heavily on token pasting, a feature of the C language. Basically, the argument of a macro is used to form identifiers.

For example, consider:

```
#define MYMAC(ARG)     MYMAC##ARG##()
```

If you call MYMAC(0), then it resolves into MYMAC0() which can be a totally different macro definition.

The HAL uses this in many instances.

```
#define DMA_RGET(REG)   _PER_RGET(_DMA_##REG##_ADDR, DMA,
REG)
```

Where,

```
#define _PER_RGET(addr,PER,REG) (*(volatileUint32*)(addr))
```

Because of this token pasting, there is the possibility of side effects if you define macros that match the token names.

24.3.5 Peripheral Register Data Sheet

It is beyond the scope of this document to list every register name and every bit-field name. Instead, it is anticipated that you will have all applicable supplemental documentation available when working with this user's guide.

One option is to look inside the HAL header files where it is fairly easy to determine the register names, field names, and field values.

24.4 HAL Macro Reference

<PER>_ADDR *Register Address*

Macro <PER>_ADDR(<REG>)

Arguments

<REG>	Register name

Return Value

Uint32	Address

Description Returns the address of a memory mapped register. <PER> is a placeholder for the peripheral name: DMA, MCBSP, TIMER, etc.

Example

```
Uint32 regAddr;
   regAddr = DMA_ADDR(PRICTL0);
   regAddr = DMA_ADDR(AUXCTL);
   regAddr = MCBSP_ADDR(SPCR0);
```

<PER>_ADDRH *Register Address for a Given Handle*

Macro <PER>_ADDRH(h,<REG>)

Arguments

h	Peripheral handle
<REG>	Register name

Return Value

Uint32	Address

Description Returns the address of the memory-mapped register given a handle. Only registers covered by the handle structure are valid, if any. <PER> is a placeholder for the peripheral name: DMA, MCBSP, TIMER, etc.

Example

```
DMA_Handle hDma;
   Uint32 regAddr;
   hDma = DMA_open(DMA_CHA2, 0);
   regAddr = DMA_ADDRH(hDma, PRICTL);
   regAddr = DMA_ADDRH(hDma, SRC);
```

<PER>_CRGET *Gets the Value of CPU Register*

Macro <PER>_CRGET(<REG>)

Arguments

<REG>	CPU register name (i.e. CSL, IER, ISR...)

Return Value

Uint32	Register value

Description Returns the current value of a CPU register. The value returned is right-justified. <PER> is a placeholder for the peripheral name (applies to CHIP module only).

Example

```
Uint32 valReg;
   valReg = CHIP_CRGET(CSR);
```

<PER>_CRSET — *Sets the Value of CPU Register*

Macro <PER>_CRSET(<REG>)

Arguments

<REG>	CPU register name (i.e. CSL, IER, ISR...)
x	Uint 32 value to set the register

Return Value none

Description Writes the value x into the CPU <REG> register. x may be any valid C expression. <PER> is a placeholder for the peripheral name (applies to CHIP module only).

Example

```
/*  set the IER register   */
   CHIP_CRSET(IER,0x00010001);
```

<PER>_FGET — *Gets a Field Value From a Register*

Macro <PER>_FGET(<REG>,<FIELD>)

Arguments

<REG>	Register name
<FIELD>	Field name

Return Value

Uint32	Field value, right-justified

Description Returns a field value from a register. The value returned is right-justified. <PER> is a placeholder for the peripheral name: DMA, MCBSP, TIMER, etc.

Example

```
Uint32 fieldVal;
   fieldVal = DMA_FGET(PRICTL0, INDEX);
   fieldVal = DMA_FGET(AUXCTL, CHPRI);
   fieldVal = MCBSP_FGET(SPCR0, XRDY);
```

<PER>_FGETA — *Gets Field for a Given Address*

Macro <PER>_FGETA(addr,<REG>,<FIELD>)

Arguments

addr	Uint32 address
<REG>	Register name
<FIELD>	Field name

Return Value

Uint32	Field value, right-justified

Description Returns the value of the field given the address of the memory mapped register. The return value is right-justified. This macro is useful in those situations where an arbitrary memory location is treated like a register such as in a configuration structure. <PER> is a placeholder for the peripheral name: DMA, MCBSP, TIMER, etc.

Example

```
Uint32 fieldVal;
   Uint32 regAddr = 0x01840000;
   DMA_Config cfg;
   fieldVal = DMA_FGETA(0x01840000, PRICTL, INDEX);
   fieldVal = DMA_FGETA(regAddr, PRICTL, PRI);
   fieldVal = DMA_FGETA(0x01840070, AUXCTL, CHPRI);
   fieldVal = DMA_FGETA(&(cfg.prictl), PRICTL, EMOD);
```

<PER>_FGETH *Gets Field for a Given Handle*

Macro <PER>_FGETH(h,<REG>,<FIELD>)

Arguments

h	Peripheral handle
<REG>	Register name
<FIELD>	Field name

Return Value Uint32 Field value, right-justified

Description Returns the value of the field given a handle. Only registers covered by handles per peripheral are valid, if any. <PER> is a placeholder for the peripheral name: DMA, MCBSP, TIMER, etc.

Example

```
DMA_Handle hDma;
   Uint32 fieldVal;
   hDma = DMA_open(DMA_CHA1, 0);
   fieldVal = DMA_FGETH(hDma, PRICTL , ESIZE);
```

<PER>_FMK *Field Make*

Macro <PER>_FMK(<REG>,<FIELD>,x)

Arguments

<REG>	Register name
<FIELD>	Field name
x	Field value, right-justified

Return Value Uint32 In-place and masked-field value

Description This macro takes the right-justified field value then shifts it over and masks it to form the in-place field value. It can be bit-wise OR'ed with other FMK or FMKS macros to form a register value as an alternative to the RMK macro.

Example

```
Uint32 x;
   x = DMA_FMK(AUXCTL, CHPRI, 0)
     | DMA_FMK(AUXCTL, AUXPRI, 0);
```

<PER>_FMKS *Field Make Symbolically*

Macro <PER>_FMKS(<REG>,<FIELD>,<SYM>)

Arguments

<REG>	Register name
<FIELD>	Field name
<SYM>	Symbolic field value

Return Value Uint32 In-place and masked-field value

Description This macro takes the symbolic field value then shifts it over and masks it to form the in-place field value. It can be bit-wise OR'ed with other FMK or FMKS macros to form a register value as an alternative to the RMK macro.

Example

```
Uint32 x;
   x = DMA_FMKS(AUXCTL, CHPRI, HIGHEST)
     | DMA_FMKS(AUXCTL, AUXPRI, CPU);
```

<PER>_FSET *Field Set*

Macro <PER>_FSET(<REG>,<FIELD>,x)

Arguments

<REG>	Register name
<FIELD>	Field name
x	Uint32 right-justified field value

Return Value none

Description Sets a field of a register to the specified value. The value is right-justified. <PER> is a placeholder for the peripheral name: DMA, MCBSP, TIMER, etc.

Example

```
Uint32 fieldVal = 0;
   DMA_FSET(PRICTL0, INDEX, 0);
   DMA_FSET(PRICTL0, INDEX, fieldVal);
   DMA_FSET(AUXCTL, CHPRI, 1);
   TIMER_FSET(CTL0, GO, 0);
```

<PER>_FSETA — *Sets Field for a Given Address*

Macro <PER>_FSETA(addr,<REG>,<FIELD>,x)

Arguments

addr	Uint32 address
<REG>	Register name
<FIELD>	Field name
x	Uint32 field value, right-justified

Return Value none

Description Sets the field value to *x* where *x* is right-justified. This macro is useful in those situations where an arbitrary memory location is treated like a register such as in a configuration structure. <PER> is a placeholder for the peripheral name: DMA, MCBSP, TIMER, etc.

Example

```
Uint32 fieldVal = 0;
   Uint32 regAddr = 0x01840000;
   DMA_FSETA(0x01840000, PRICTL ,INDEX, 0);
   DMA_FSETA(regAddr, PRICTL, INDEX, fieldVal);
   DMA_FSETA(0x01840000, PRICTL, INDEX, fieldVal);
   MCBSP_FSETA(0x018C0008, SPCR, DLB, 0);
   Uint32 dummyReg = DMA_PRICTL_DEFAULT;
   DMA_FSETA(&dummyReg, PRICTL, EMOD, 1);
```

<PER>_FSETH — *Sets Field for a Given Handle*

Macro <PER>_FSETH(h,<REG>,<FIELD>,x)

Arguments

h	Peripheral handle
<REG>	Register name
<FIELD>	Field name
x	Uint32 field value, right-justified

Return Value none

Description Sets the field value to *x* given a handle. Only registers covered by handles per peripheral are valid, if any. <PER> is a placeholder for the peripheral name: DMA, MCBSP, TIMER, etc.

Example

```
DMA_Handle hDma;
   hDma = DMA_open(DMA_CHA2, DMA_OPEN_RESET);
   DMA_FSETH(hDma, PRICTL, ESIZE, 3);
```

<PER>_FSETS — *Sets a Field Symbolically*

Macro

<PER>_FSETS(<REG>,<FIELD>,<SYM>)

Arguments

<REG>	Register name
<FIELD>	Field name
<SYM>	Symbolic value name

Return Value

none

Description

Sets a register field to the specified symbol value. The value *MUST* be one of the predefined symbolic names for that field for that register. <PER> is a placeholder for the peripheral name: DMA, MCBSP, TIMER, etc.

Example

```
DMA_FSETS(PRICTL0, INDEX, A);
   DMA_FSETS(AUXCTL, CHPRI, HIGHEST);
   MCBSP_FSETS(SPCR0, DLB, OFF);
   MCBSP_FSETS(SPCR1, DLB, DEFAULT);
```

<PER>_FSETSA — *Sets Field Symbolically for a Given Address*

Macro

<PER>_FSETSA(addr,<REG>,<FIELD>,<SYM>)

Arguments

addr	Uint32 address
<REG>	Register name
<FIELD>	Field name
<SYM>	Field value name

Return Value

none

Description

Sets a register field to the specified value. The value *MUST* be one of the predefined symbolic names for that field in that register. This macro is useful in those situations where an arbitrary memory location is treated like a register such as in a configuration structure. <PER> is a placeholder for the peripheral name: DMA, MCBSP, TIMER, etc.

Example

```
Uint32 regAddr = 0x01840000;
   DMA_FSETSA(0x01840000, PRICTL ,INDEX, A);
   DMA_FSETSA(regAddr, PRICTL, INDEX, B);
   DMA_FSETSA(0x01840000, PRICTL, INDEX, A);
   MCBSP_FSETSA(0x018C0008, SPCR, DLB, OFF);
   Uint32 dummyReg = DMA_PRICTL_DEFAULT;
   DMA_FSETSA(&dummyReg, PRICTL, EMOD, HALT);
```

<PER>_FSETSH — *Sets Field Symbolically for a Given Handle*

Macro <PER>_FSETSH(h,<REG>,<FIELD>,<SYM>)

Arguments

h	Peripheral handle
<REG>	Register name
<FIELD>	Field name
<SYM>	Symbolic field value

Return Value none

Description Sets a register field to the specified value given a handle. The value *MUST* be one of the predefined symbolic names for that field in that register. Only registers covered by handles per peripheral are valid, if any. <PER> is a placeholder for the peripheral name: DMA, MCBSP, TIMER, etc.

Example

```
DMA_Handle hDma;
   hDma = DMA_open(DMA_CHA0, DMA_OPEN_RESET);
   DMA_FSETSH(hDma, PRICTL, ESIZE, 32BIT);
```

<PER>_RGET — *Gets Register Current Value*

Macro <PER>_RGET(<REG>)

Arguments

<REG>	Register name

Return Value

Uint32	Register value

Description Returns the current value of a register. <PER> is a placeholder for the peripheral name: DMA, MCBSP, TIMER, etc.

Example

```
Uint32 regVal;
   regVal = DMA_RGET(PRICTL0);
   regVal = MCBSP_RGET(SPCR0);
   regVal = DMA_RGET(AUXCTL);
   regVal = TIMER_RGET(CTL0);
```

<PER>_RGETA *Gets Register Address*

Macro <PER>_RGETA(addr,<REG>)

Arguments

addr	Uint32 address
<REG>	Register name

Return Value Uint32 Register value

Description Returns the current value of a register given the address of the register. This macro is useful in those situations where an arbitrary memory location is treated like a register such as in a configuration structure. <PER> is a placeholder for the peripheral name: DMA, MCBSP, TIMER, etc.

Example

```
Uint32 regVal;
   DMA_Config cfg;
   regVal = DMA_RGETA(0x01840000, PRICTL);
   regVal = DMA_RGETA(0x01840070, AUXCTL);
   regVal = DMA_RGETA(&(cfg.prictl), PRICTL);
```

<PER>_RGETH *Gets Register for a Given Handle*

Macro <PER>_RGETH(h,<REG>)

Arguments

h	Peripheral handle
<REG>	Register name

Return Value Uint32 Uint32 register value

Description Returns the value of a register given a handle. Only registers covered by the handle structure are valid, if any. <PER> is a placeholder for the peripheral name: DMA, MCBSP, TIMER, etc.

Example

```
DMA_Handle hDma;
   Uint32 regVal;
   hDma = DMA_open(DMA_CHA2, DMA_OPEN_RESET);
   regVal = DMA_RGETH(hDma, PRICTL);
```

<PER>_RSET *Register Set*

Macro <PER>_RSET(<REG>,x)

Arguments

<REG>	Register name
x	Uint32 value to set register to

Return Value none

Description Write the value *x* to the register. *x* may be any valid C expression. <PER> is a placeholder for the peripheral name: DMA, MCBSP, TIMER, etc.

Example

```
Uint32 regVal = 0x09000101;
   DMA_RSET(PRICTL0, 0x09000101);
   DMA_RSET(PRICTL0, regVal);
   DMA_RSET(AUXCTL, DMA_AUXCTL_DEFAULT);
   MCBSP_RSET(SPCR0, 0x00000000);
   DMA_RSET(PRICTL0, DMA_RGET(PRICTL0)&0xFFFFFFFC);
```

<PER>_RSETA *Sets Register for a Given Address*

Macro <PER>_RSETA(addr,<REG>,x)

Arguments

addr	Uint32 address
<REG>	Register name
x	Uint32 register value

Return Value none

Description Sets the value of a register given its address. This macro is useful in those situations where an arbitrary memory location is treated like a register such as in a configuration structure. <PER> is a placeholder for the peripheral name: DMA, MCBSP, TIMER, etc.

Example

```
Uint32 regVal = 0x09000101;
   DMA_Config cfg;
   DMA_RSETA(0x01840000, PRICTL, 0x09000101);
   DMA_RSETA(0x01840000, PRICTL, regVal);
   DMA_RSETA(0x01840070, AUXCTL, 0);
   DMA_RSETA(&(cfg.secctl), SECCTL, 0);
```

<PER>_RSETH *Sets Register for a Given Handle*

Macro <PER>_RSETH(h,<REG>,x)

Arguments

h	Peripheral handle
<REG>	Register name
x	Uint32 register value

Return Value none

Description Sets the register value to *x* given a handle. Only registers covered by handles per peripheral are valid, if any. <PER> is a placeholder for the peripheral name: DMA, MCBSP, TIMER, etc.

Example

```
DMA_Handle hDma;
   hDma = DMA_open(DMA_CHA0, DMA_OPEN_RESET);
   DMA_RSETH(hDma, PRICTL, 0x09000101);
   DMA_RSETH(hDma, SECCTL, 0x00000080);
```

<PER>_<REG>_DEFAULT *Register Default Value*

Macro <PER>_<REG>_DEFAULT

Arguments none

Return Value Uint32 Register default value

Description Returns the default (power-on) value for the specified register.

Example

```
Uint32 defRegVal;;
   defRegVal = DMA_AUXCTL_DEFAULT;
   defRegVal = DMA_PRICTL_DEFAULT;
   defRegVal = EMIF_GBLCTL_DEFAULT;
```

<PER>_<REG>_OF *Returns Value Of*

Macro <PER>_<REG>_OF(x)

Arguments

x		Register value

Return Value

Uint32	Returns *x* casted into a Uint32

Description This macro simply takes the argument *x* and returns it. It is type-casted into a Uint32. The intent is to make code more readable. The examples illustrate this: Example 1 does not use these macros and Example 2 does. Notice how Example 2 is easier to follow. You are not required to use these macros; however, they can be helpful.

Example 1

```
/* create a config structure using hard coded values */
   DMA_Config cfg = {
     0x10002050,
     0x00000080,
     (Uint32)buffa,
     (Uint32)buffb,
     0x00010008
   );
```

Example 2

```
/* create a config structure using the OF macros */
   DMA_Config cfg = {
     DMA_PRICTL_OF(0x10002050),
     DMA_SECCTL_OF(0x00000080),
     DMA_SRC_OF(buffa),
     DMA_DST_OF(buffb),
     DMA_XFRCNT_OF(0x00010008)
   );
```

<PER>_<REG>_RMK *Register Make*

Macro <PER>_<REG>_RMK(field_ms,…,field_ls)

Arguments

field_ms	Most-significant field value, right-justified
…	Intermediate-field values, right-justified
field_ls	Least-significant field value, right-justified

Return Value

Uint32	Value suitable for this register

Description This macro constructs (makes) a register value based on individual field values. It does not do any writes; it just returns a value suitable for this register. Use this macro to make your code readable.

Only writable fields are specified and they are ordered from most significant to least significant. Also, note that this macro may vary from one device to another for the same register.

Example

```
Uint32 prictl;
   prictl = DMA_PRICTL_RMK(
     DMA_PRICTL_DSTRLD_NONE,
     DMA_PRICTL_SRCRLD_NONE,
     DMA_PRICTL_EMOD_HALT,
     DMA_PRICTL_FS_DISABLE,
     DMA_PRICTL_TCINT_DISABLE,
     DMA_PRICTL_PRI_CPU,
     DMA_PRICTL_WSYNC_NONE,
     DMA_PRICTL_RSYNC_NONE,
     DMA_PRICTL_INDEX_DEFAULT,
     DMA_PRICTL_CNTRLD_DEFAULT,
     DMA_PRICTL_SPLIT_DISABLE,
     DMA_PRICTL_ESIZE_32BIT,
     DMA_PRICTL_DSTDIR_INC,
     DMA_PRICTL_SRCDIR_INC,
     DMA_PRICTL_START_NORMAL
   );
```

<PER>_<REG>_<FIELD>_DEFAULT *Field Default Value*

Macro <PER>_<REG>_<FIELD>_DEFAULT

Arguments none

Return Value Uint32 Register default value

Description Returns the default (power-on) value for the specified field.

Example

```
Uint32 defFieldVal;;
   defRegVal = DMA_AUXCTLCHPRI_DEFAULT;
   defRegVal = DMA_PRICTLESIZE_DEFAULT;
```

<PER>_<REG>_<FIELD>_OF *Field Value Of*

Macro <PER>_<REG>_<FIELD>_OF(x)

Arguments x Field value, right-justified

Return Value Uint32 Returns *x* casted into a Uint32

Description This macro simply takes the argument *x* and returns it. It is type-casted into a Uint32. The intent is to make code more readable. It serves a similar purpose to the <PER>_<REG>_OF() macros. Generally, these macros are used in conjunction with the <PER>_RMK(…) macros. You are not required to use these macros; however, they can be helpful.

Example

```
Uint32 idx;
   idx = DMA_PRICTL_INDEX_OF(1);
```

<PER>_<REG>_<FIELD>_<SYM> *Field Symbolic Value*

Macro <PER>_<REG>_<FIELD>_<SYM>

Arguments none

Return Value Uint32 field value

Description Sets the specified value to the bit field

Example

```
MCBSP_SRGR_RMK(
   MCBSP_SRGR_GSYNC_FREE,
   MCBSP_SRGR_CLKSP_RISING,
   MCBSP_SRGR_CLKSM_INTERNAL,
   MCBSP_SRGR_FSGM_DXR2XSR,
   MCBSP_SRGR_FPER_OF(63),
   MCBSP_SRGR_FWID_OF(31),
   MCBSP_SRGR_CLKGDV_OF(15)
),
```

Appendix A

Using CSL APIs Without DSP/BIOS ConfigTool

You are not required to use the DSP/BIOS Configuration Tool when working with the CSL library. As GUI–based configuration of the CSL is getting deprecated, it is recommended to avoid its usage.

Note:

You can continue to use the CDB file in your application but only for configuring CSL peripherals; you can choose not to use CSL GUI in the CDB file.

For 6713 and DA610 CSL, the GUI configuration supports only the peripherals already supported in GUI for 6711.

This appendix provides an example of using CSL independently of the DSP/BIOS kernel.

Topic **Page**

A.1 Using CSL APIs

Example A–1 illustrates the use of CSL to initialize DMA channel 0 and copy table BuffA to another table BuffB of 1024 bytes (1024/4 elements).

A.1.1 Using DMA_config()

Example A–1 uses the DMA_config() function to initialize the registers.

Example A–1. Initializing a DMA Channel with DMA_config()

```
// Step 1:   Include the generic csl.h include file and
//           the header file of the module/peripheral you
//           will use. The different header files are shown
//           in Table 1–1, CSL Modules and Include Files, on page 1-4.
//           The order of inclusion does not matter.

#include <csl.h>
#include <csl_dma.h>

// Example–specific initialization
#define BUFFSZ 1024

#define Uint32 BuffA[BUFFSZ/sizeof(Uint32)]

#define Uint32 BuffB[BUFFSZ/sizeof(Uint32)]









//Step 2:    Define and initialize the DMA channel
//           configuration structure.
```

```
DMA_Config  myconfig = {
    /* DMA_PRICTL */
    DMA_PRICTL_RMK(
   DMA_PRICTL_DSTRLD_NONE,
   DMA_PRICTL_SRCRLD_NONE,
   DMA_PRICTL_EMOD_NOHALT,
   DMA_PRICTL_FS_DISABLE,
   DMA_PRICTL_TCINT_DISABLE,
   DMA_PRICTL_PRI_DMA,
   DMA_PRICTL_WSYNC_NONE,
   DMA_PRICTL_RSYNC_NONE,
   DMA_PRICTL_INDEX_NA,
   DMA_PRICTL_CNTRLD_NA,
   DMA_PRICTL_SPLIT_DISABLE,
   DMA_PRICTL_ESIZE_32BIT,
   DMA_PRICTL_DSTDIR_INC,
   DMA_PRICTL_SRCDIR_INC,
   DMA_PRICTL_STATUS_STOPPED,
   DMA_PRICTL_START_NORMAL,
     ),

     /* DMA_SECCTL */
     DMA_SECCTL_RMK(
   DMA_SECCTL_WSPOL_NA,
   DMA_SECCTL_RSPOL_NA,
   DMA_SECCTL_FSIG_NA,
   DMA_SECCTL_DMACEN_LOW,
   DMA_SECCTL_WSYNCCLR_NOTHING,
   DMA_SECCTL_WSYNCSTAT_CLEAR,
   DMA_SECCTL_RSYNCCLR_NOTHING,
   DMA_SECCTL_RSYNCSTAT_CLEAR,
   DMA_SECCTL_WDROPIE_DISABLE,
   DMA_SECCTL_WDROPCOND_CLEAR,
   DMA_SECCTL_RDROPIE_DISABLE,
   DMA_SECCTL_RDROPCOND_CLEAR,
   DMA_SECCTL_BLOCKIE_ENABLE,
   DMA_SECCTL_BLOCKCOND_CLEAR,
   DMA_SECCTL_LASTIE_DISABLE,
   DMA_SECCTL_LASTCOND_CLEAR,
   DMA_SECCTL_FRAMEIE_DISABLE,
   DMA_SECCTL_FRAMECOND_CLEAR,
   DMA_SECCTL_SXIE_DISABLE,
   DMA_SECCTL_SXCOND_CLEAR,
     ),

     /* src */
     (Uint32) BuffA,

     /* dst */
     (Uint32) BuffB,

     /* xfrcnt */
     BUFFSZ/sizeof(Uint32)
   };
```

Example A–1. Initializing a DMA Channel with DMA_config() (Continued)

```
//Step 3:    Define a DMA_Handle pointer. DMA_open returns
//           a pointer to a DMA_Handle when a DMA channel is
//           opened.

DMA_Handle myhDma;

void main(void) {

// Initialize Buffer tables

  for (x=0;x<BUFFSZ/sizeof(Uint32);x++) {

       BuffA[x]=x;

       BuffB[x]=0;

   }

//Step 4:    One-time only initialization of the CSL library
//           and of the CSL module to be used. Must be done
//           before calling any CSL module API.

   CSL_init();                      /* Init CSL      */

//Step 5: Open, configure and start the DMA channel.
//        To configure the channel you can use the
//        DMA_config or DMA_configArgs functions.

   myhDma = DMA_open(DMA_CHA0,0);/*Open Channel(Optional) */
   DMA_config(myhDma, &myconfig);   /* Configure Channel */
   DMA_start(myhDma);            /* Begin Transfer    */

//Step 6: (Optional)
//        Use CSL DMA APIs to track DMA channel status.

   while(DMA_getStatus(myhDma));   /* Wait for complete */

//Step 7: Close DMA channel after using.

   DMA_close(myhDma);         /* Close channel (Optional) */
}
```

Note:

The usage of the RMK macro for configuring registers is recommended as shown above in Step 2. This is because it is using symbolic constants provided by CSL for setting fields of the register.

Refer to Table 1–5 for further help on the RMK macro. Also refer to Appendix B for the symbolic constants provided by CSL for registers and bit–fields of any peripherals.

A.1.2 Using DMA_configArgs()

Example A–2 performs the same task as Example A–1 but uses DMA_configArgs() to initialize the registers.

Example A–2. Initializing a DMA Channel with DMA_configArgs()

```
// Step 1:   Include the generic csl.h include file and
//           the header file of the module/peripheral you
//           will use. The different header files are shown
//           in  Table 1–1, CSL Modules and Include Files, on page 1-4.
//           The order of inclusion does not matter.

#include <csl.h>
#include <csl_dma.h>

// Example–specific initialization
#define BUFFSZ 1024

#define Uint32 BuffA[BUFFSZ/sizeof(Uint32)]

#define Uint32 BuffB[BUFFSZ/sizeof(Uint32)]




//Step 2:    Define a DMA_Handle pointer. DMA_open returns
//           a pointer to a DMA_Handle when a DMA channel is
//           opened.

DMA_Handle myhDma;

void main(void) {

// Initialize Buffer tables

  for (x=0;x<BUFFSZ/sizeof(Uint32);x++) {

       BuffA[x]=x;

       BuffB[x]=0;

   }

//Step 3:    One-time only initialization of the CSL library
//           and of the CSL module to be used. Must be done
//           before calling any CSL module API.

   CSL_init();                      /* Init CSL      */

//Step 4: Open, configure, and start the DMA channel.
//        To configure the channel you can use the
//        DMA_config() or DMA_configArgs() functions.
```

Example A–2. Initializing a DMA Channel with DMA_configArgs() (Continued)

```
    myhDma = DMA_open(DMA_CHA0,0);/*Open Channel(Optional) */
    DMA_configArgs(myhDma,
      0x01000051,                      /* prictl */
       0x00000080,                      /* secctl */
       (Uint32) BuffA,                  /* src    */
       (Uint32) BuffB,                  /* dst    */
       BUFFSZ/sizeof(Uint32)            /* xfrcnt */
};
    );
    DMA_start(myhDma);            /* Begin Transfer    */

//Step 6: (Optional)
//        Use CSL DMA APIs to track DMA channel status.

    while(DMA_getStatus(myhDma));   /* Wait for complete */

//Step 7: Close DMA channel after using.

    DMA_close(myhDma);         /* Close channel (Optional) */
}
```

A.2 Compiling and Linking With CSL Using Code Composer Studio IDE

A.2.1 CSL Directory Structure

Table A–1 lists the locations of the CSL components. Use this information when you set up the compiler and linker search paths.

Table A–1. CSL Directory Structure

This CSL component...	is located in this directory...
Libraries	c:\ti\c6000\bios\lib
Source library	c:\ti\c6000\bios\src
Include files	c:\ti\c6000\bios\include
Examples	c:\ti\examples\dsk6211\csl c:\ti\examples\evm6201\csl
Documentation	c:\ti\docs

A.2.2 Using the Code Composer Studio Project Environment

To configure the CCS project environment to work with CSL, follow these steps to specify the target device you are using:

1) Select Project→Options... from the menu bar.

2) Select the Compiler tab in the Build Options dialog box (Figure A–1), and highlight Symbols in the category list box.

3) In the Define Symbols field, enter one and only one of the compiler symbols in Table 1–10, *CSL Device Support Library Name and Symbol Conventions*, on page 1-19.

   For example, if you are using the 6201™ device, enter CHIP_6201.

4) Click OK.

Figure A–1. Defining the Target Device in the Build Options Dialog Box

Appendix B

TMS320C6000 CSL Registers

This appendix shows the registers associated with current TMS320C6000 digital signal processor (DSP) devices.

Topic **Page**

B.1 Cache Registers

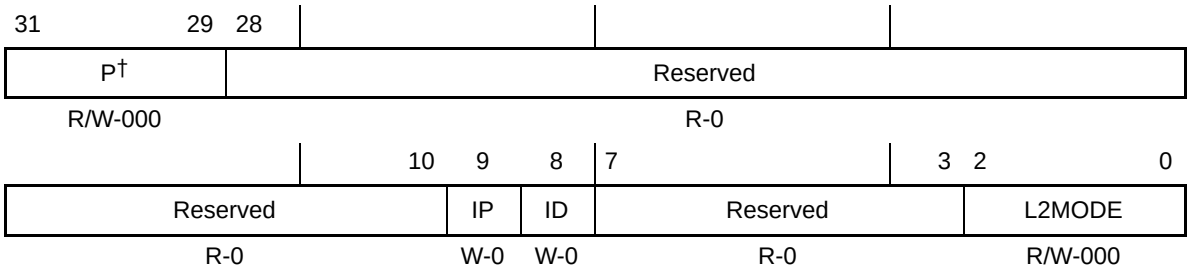
Table B–1. Cache Registers

Acronym	Register Name	Section
CCFG	Cache configuration register	B.1.1
L2WBAR	L2 writeback base address register	B.1.2
L2WWC	L2 writeback word count	B.1.3
L2WIBAR	L2 writeback– invalidate base address register	B.1.4
L2WIWC	L2 writeback– invalidate word count	B.1.5
L2IBAR‡	L2 invalidate base address register	B.1.6
L2IWC‡	L2 invalidate word count	B.1.7
Q1CNT–Q4CNT‡	L2 allocation priority queue registers	B.1.8
L1PIBAR	L1P invalidate base address register	B.1.9
L1PIWC	L1P invalidate word count	B.1.10
L1DWIBAR	L1D writeback– invalidate base address register	B.1.11
L1DWIWC	L1D writeback– invalidate word count	B.1.12
L1DIBAR‡	L1D invalidate base address register	B.1.13
L1DIWC‡	L1D invalidate word count	B.1.14
L2WB	L2 writeback all	B.1.15
L2WBINV	L2 writeback– invalidate all	B.1.16
MAR0–15†	L2 memory attribute registers	B.1.17
MAR96–111‡	L2 memory attribute registers for EMIFA only	B.1.18
MAR128–191‡	L2 memory attribute registers for EMIFB only	B.1.19

Notes: 1) The names of the registers have been changed, Appendix D contains a comparison table of old versus new names.
† For C621x/C671x only.
‡ For C64x only.

B.1.1 Cache Configuration Register (CCFG)

Figure B–1. Cache Configuration Register (CCFG)



† Applicable on C64 only. On C621x/C671x, bit field P is Reserved, R-000b.

Legend: R/W-x = Read/Write-Reset value

Table B–2. Cache Configuration Register (CCFG) Field Values (CACHE_CCFG_field_symval)

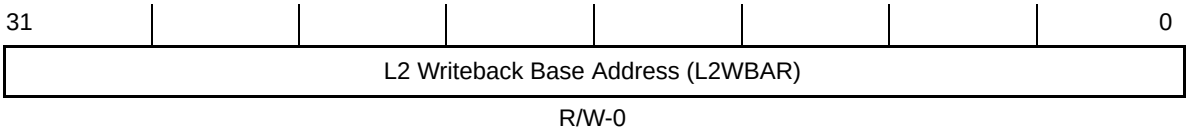
Bit	*field*	*symval*	Value	Description
31–29	P			L2 Requestor Priority (for C64x only, reserved for C621x/C671x)
		URGENT	000	L2 controller requests are placed on urgent priority level
		HIGH	001	L2 controller requests are placed on high priority level
		MEDIUM	010	L2 controller requests are placed on medium priority level
		LOW	011	L2 controller requests are placed on low priority level
28–10	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
9	IP			L1P operation
		NORMAL	0	Normal L1P operation
		INVALIDATE	1	All L1P lines invalidated
8	ID			Invalidate L1D
		NORMAL	0	Normal L1D operation
		INVALIDATE	1	All L1D lines invalidated Invalidate LIP
7–3	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.

Table B–2. Cache Configuration Register (CCFG) Field Values (CACHE_CCFG_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
2–0	L2MODE			L2 Operation Mode
		0KC	000	No L2 Cache (All SRAM mode)
		16KC	001	1-way cache (16K L2 cache) C621x/C671x only
		32KC	001	4-way cache (32K L2 cache) C64x only
		32KC	010	2-way cache (32K L2 cache) C621x/C671x only
		32KC	010	4-way cache (64K L2 cache) C64x only
		48KC	011	3-way cache (48K L2 cache) C621x/C671x only
		48KC	011	4-way cache (128K L2 cache) C64x only
			100–110	Reserved
		64KC	111	4-way cache (64K L2 cache) C621x/C671x only
		64KC	111	4-way cache (256K L2 cache) C64x only

B.1.2 L2 Writeback Base Address Register (L2WBAR)

Figure B–2. L2 Writeback Base Address Register (L2WBAR)



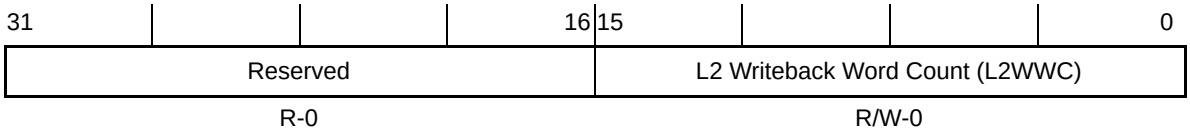
Legend: R/W-x = Read/Write-Reset value

Table B–3. L2 Writeback Base Address Register (L2WBAR) Field Values (CACHE_L2WBAR_field_symval)

Bit	*field*	*symval*	Value	Description
31–0	LWFBAR	OF(*value*)	0–FFFFFFFFh	

B.1.3 L2 Writeback Word Count Register (L2WWC)

Figure B–3. L2 Writeback Word Count Register (L2WWC)



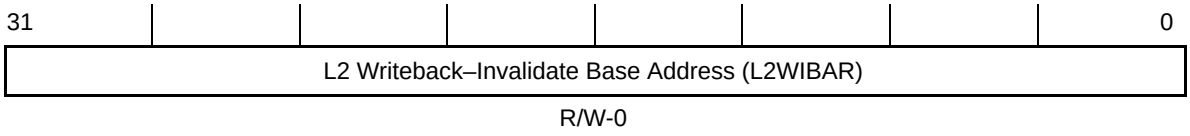
Legend: R/W-x = Read/Write-Reset value

Table B–4. L2 Writeback Word Count Register (L2WWC) Field Values (CACHE_L2WWC_field_symval)

Bit	*field*	*symval*	Value	Description
31–16	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
15–0	L2WWC	OF(*value*)	0–FFFFh	

B.1.4 L2 Writeback–Invalidate Base Address Register (L2WIBAR)

Figure B–4. L2 Writeback–Invalidate Base Address Register (L2WIBAR)



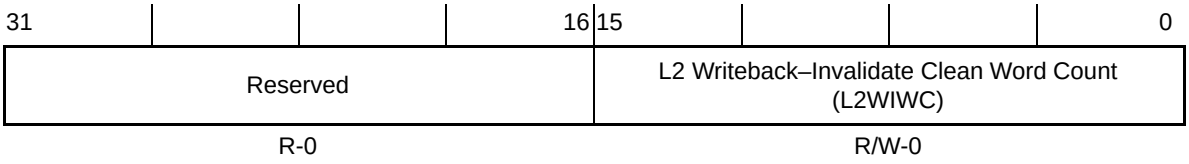
Legend: R/W-x = Read/Write-Reset value

Table B–5. L2 Writeback–Invalidate Base Address Register (L2WIBAR) Field Values (CACHE_L2WIBAR_field_symval)

Bit	*field*	*symval*	Value	Description
31–0	L2WIBAR	OF(*value*)	0–FFFFFFFFh	

B.1.5 L2 Writeback–Invalidate Count Register (L2WIWC)

Figure B–5. L2 Writeback–Invalidate Word Count Register (L2WIWC)



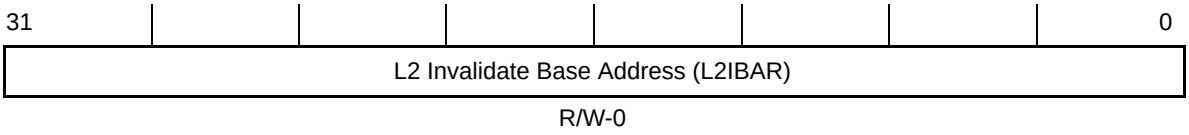
Legend: R/W-x = Read/Write-Reset value

Table B–6. L2 Writeback–Invalidate Word Count Register (L2WIWC) Field Values (CACHE_L2WIWC_field_symval)

Bit	*field*	*symval*	Value	Description
31–16	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
15–0	L2WIWC	OF(*value*)	0–FFFFh	

B.1.6 L2 Invalidate Base Address Register (L2IBAR) (C64x only)

Figure B–6. L2 Invalidate Base Address Register (L2IBAR)



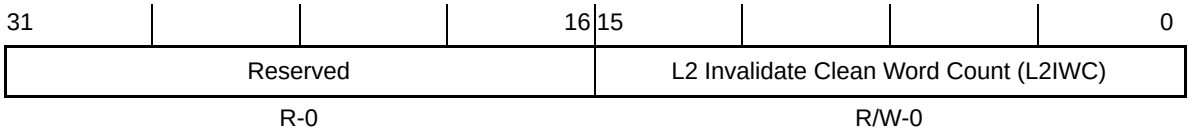
Legend: R/W-x = Read/Write-Reset value

Table B–7. L2 Invalidate Base Address Register (L2IBAR) Field Values (CACHE_L2IBAR_field_symval)

Bit	*field*	*symval*	Value	Description
31–0	L2IBAR	OF(*value*)	0–FFFFFFFFh	

B.1.7 L2 Invalidate Count Register (L2IWC) (C64x only)

Figure B–7. L2 Writeback–Invalidate Word Count Register (L2IWC)



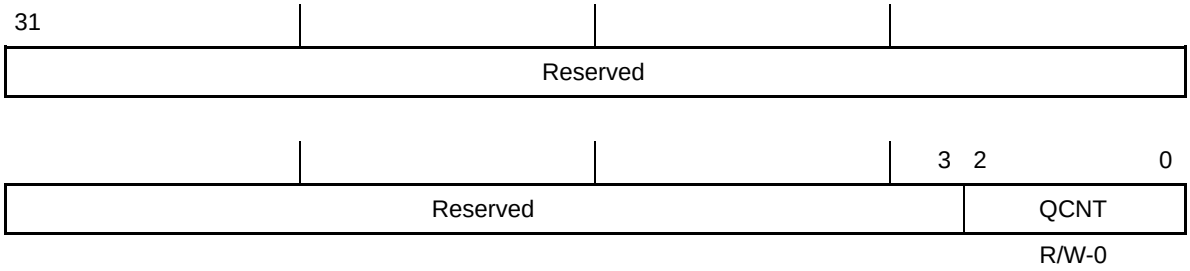
Legend: R/W-x = Read/Write-Reset value

Table B–8. L2 Invalidate Word Count Register (L2IWC) Field Values (CACHE_L2IWC_field_symval)

Bit	*field*	*symval*	Value	Description
31–16	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
15–0	L2IWC	OF(*value*)	0–FFFFh	

B.1.8 L2 Allocation Priority Queue Registers (Q1CNT–Q4CNT) (C64x)

Figure B–8. L2 Allocation Registers (Q1CNT–Q4CNT)



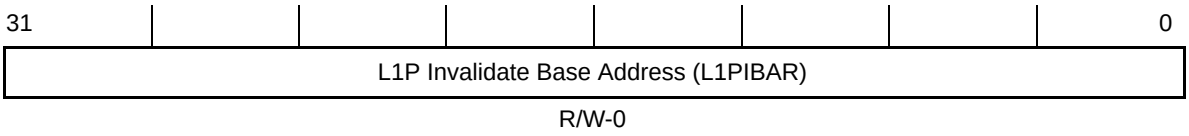
Legend: R/W-x = Read/Write-Reset value

Table B–9. L2 Allocation Registers (Q1CNT–Q4CNT) Field Value (CACHE_QCNT_field_symval)

Bit	*field*	*symval*	Value	Description
31–3	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
2–0	QCNT			

B.1.9 L1P Invalidate Base Address Register (L1PIBAR)

Figure B–9. L1P Invalidate Base Address Register (L1PIBAR)



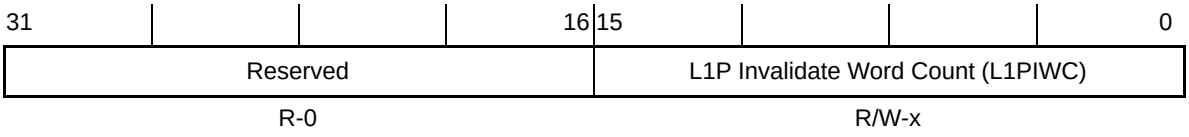
Legend: R/W-x = Read/Write-Reset value

Table B–10. L1P Invalidate Base Address Register (L1PIBAR) Field Values (CACHE_L1PIBAR_field_symval)

Bit	*field*	*symval*	Value	Description
31–0	L1PIBAR	OF(*value*)	0–FFFFFFFFh	

B.1.10 L1P Invalidate Word Count Register (L1PIWC)

Figure B–10. L1P Invalidate Word Count Register (L1PIWC)



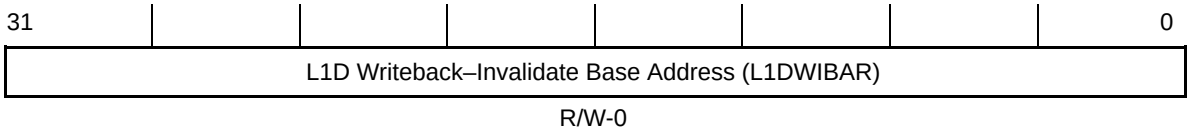
Legend: R/W-x = Read/Write-Reset value

Table B–11. L1P Invalidate Word Count Register (L1PIWC) Field Values (CACHE_L1PIWC_field_symval)

Bit	*field*	*symval*	Value	Description
31–16	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
15–0	L1PIWC	OF(*value*)	0–FFFFh	

B.1.11 L1D Writeback–Invalidate Base Address Register (L1DWIBAR)

Figure B–11. L1D Writeback–Invalidate Base Address Register (L1DWIBAR)



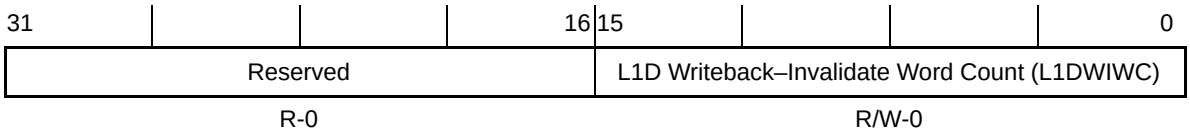
Legend: R/W-x = Read/Write-Reset value

Table B–12. L1D Writeback–Invalidate Base Address Register (L1DWIBAR) Field Values (CACHE_L1DWIBAR_field_symval)

Bit	*field*	*symval*	Value	Description
31–0	L1DWIBAR	OF(*value*)	0–FFFFFFFFh	

B.1.12 L1D Writeback–Invalidate Word Count Register (L1DWIWC)

Figure B–12. L1D Writeback–Invalidate Word Count Register (L1DWIWC)



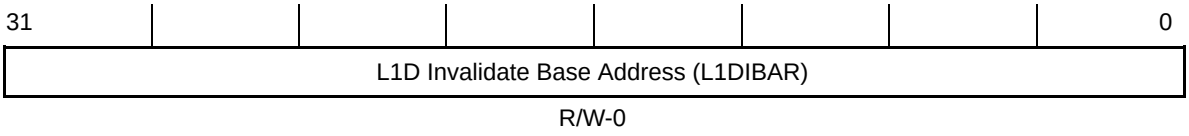
Legend: R/W-x = Read/Write-Reset value

Table B–13. L1D Writeback–Invalidate Word Count Register (L1DWIWC) Field Values (CACHE_L1DWIWC_field_symval)

Bit	*field*	*symval*	Value	Description
31–16	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
15–0	L1DWIWC	OF(*value*)	0–FFFFh	

B.1.13 L1D Invalidate Base Address Register (L1DIBAR) (C64x only)

Figure B–13. L1P Invalidate Base Address Register (L1PIBAR)



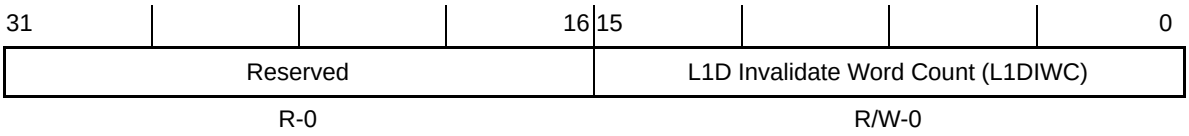
Legend: R/W-x = Read/Write-Reset value

Table B–14. L1D Invalidate Base Address Register (L1DIBAR) Field Values (CACHE_L1DIBAR_field_symval)

Bit	*field*	*symval*	Value	Description
31–0	L1DIBAR	OF(*value*)	0–FFFFFFFFh	

B.1.14 L1D Invalidate Word Count Register (L1DIWC) (C64x only)

Figure B–14. L1D Invalidate Word Count Register (L1DIWC)



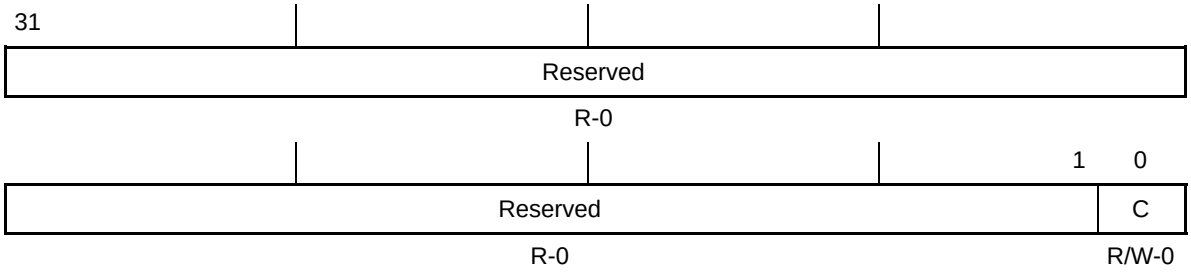
Legend: R/W-x = Read/Write-Reset value

Table B–15. L1D Invalidate Word Count Register (L1DIWC) Field Values (CACHE_L1DIWC_field_symval)

Bit	*field*	*symval*	Value	Description
31–16	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
15–0	L1DIWC	OF(*value*)	0–FFFFh	

B.1.15 L2 Writeback All Register (L2WB)

Figure B–15. L2 Writeback All Register (L2WB)



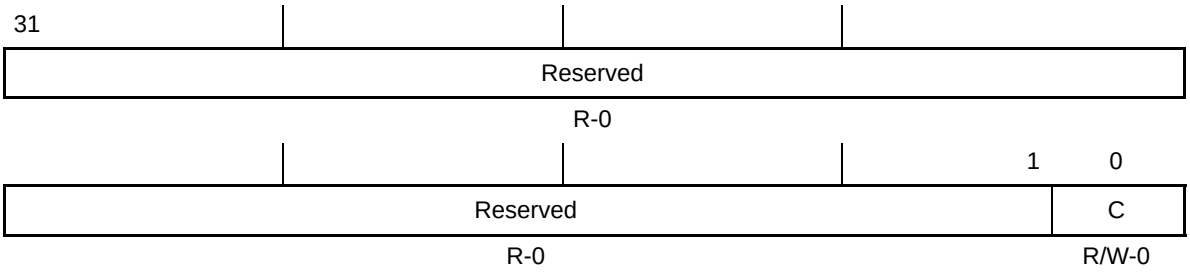
Legend: R/W-x = Read/Write-Reset value

Table B–16. L2 Writeback All Register (L2WB) Field Values (CACHE_L2WB_field_symval)

Bit	*field*	*symval*	Value	Description
31–1	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
0	C			Writeback All L2
		NORMAL	0	Normal L2 operation
		FLUSH	1	All L2 lines writeback all

B.1.16 L2 Writeback–Invalidate All Register (L2WBINV)

Figure B–16. L2 Writeback–Invalidate All Register (L2WBINV)



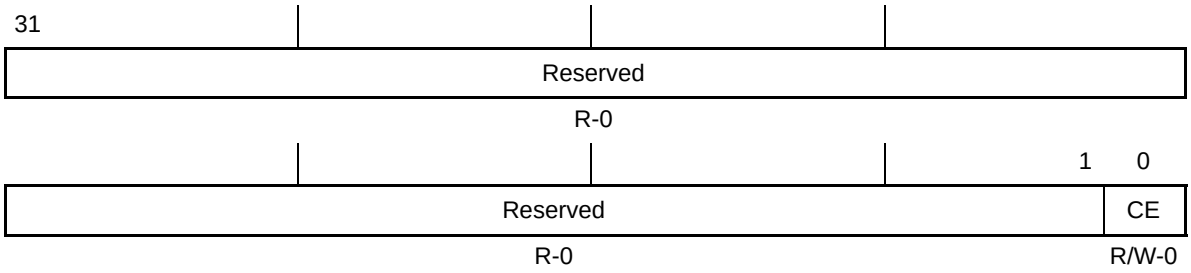
Legend: R/W-x = Read/Write-Reset value

Table B–17. L2 Writeback–Invalidate All Register (L2WBINV) Field Values (CACHE_L2WBINV_field_symval)

Bit	*field*	*symval*	Value	Description
31–1	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
0	C			Clean L2
		NORMAL	0	Normal L2 operation
		CLEAN	1	All L2 lines writeback–invalidate all

B.1.17 L2 Memory Attribute Registers (MAR0–MAR15)

Figure B–17. L2 Memory Attribute Registers (MAR0–MAR15)



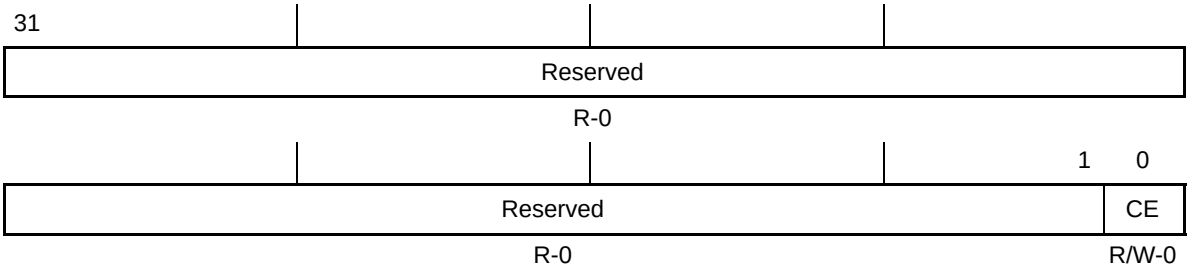
Legend: R/W-x = Read/Write-Reset value

Table B–18. L2 Memory Attribute Registers (MAR0–MAR15) Field Value (CACHE_MAR_field_symval)

Bit	*field*	*symval*	Value	Description
31–1	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
0	CE			
		DISABLE	0	
		ENABLE	1	

B.1.18 L2 Memory Attribute Registers for EMIFA Only (MAR96–MAR111)

Figure B–18. L2 Memory Attribute Registers for EMIFA Only (MAR96–MAR111)



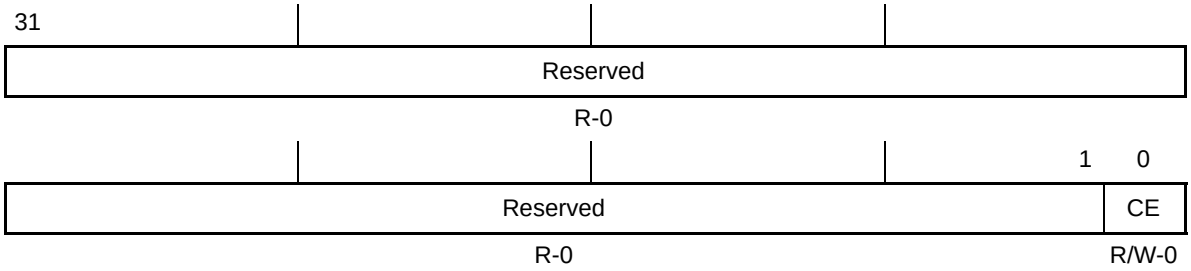
Legend: R/W-x = Read/Write-Reset value

Table B–19. L2 Memory Attribute Registers for EMIFA Only (MAR96–MAR111) Field Value (CACHE_MAR_field_symval)

Bit	*field*	*symval*	Value	Description
31–1	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
0	CE			
		DISABLE	0	
		ENABLE	1	

B.1.19 L2 Memory Attribute Registers for EMIFB Only (MAR128–MAR191)

Figure B–19. L2 Memory Attribute Registers for EMIFB Only (MAR128–MAR191)



Legend: R/W-x = Read/Write-Reset value

Table B–20. L2 Memory Attribute Registers for EMIFB Only (MAR128–MAR191) Field Value (CACHE_MAR_field_symval)

Bit	*field*	*symval*	Value	Description
31–1	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
0	CE			
		DISABLE	0	
		ENABLE	1	

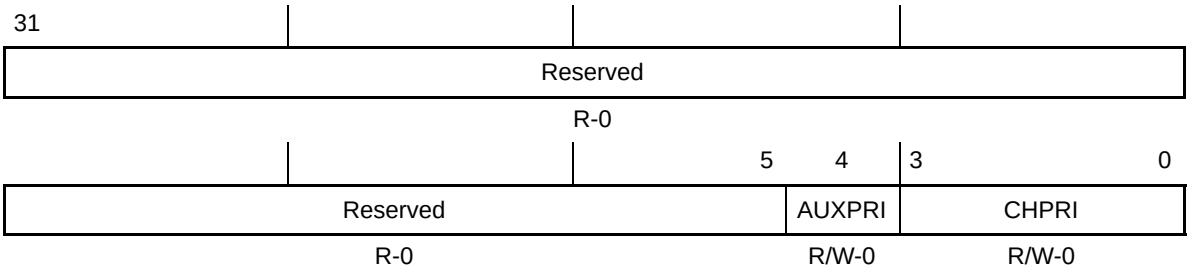
B.2 Direct Memory Access (DMA) Registers

Table B–21. DMA Registers

Acronym	Register Name	Section
AUXCTL	DMA auxiliary control register	B.2.1
PRICTL	DMA channel primary control register	B.2.2
SECCTL	DMA channel secondary control register	B.2.3
SRC	DMA channel source address register	B.2.4
DST	DMA channel destination address register	B.2.5
XFRCNT	DMA channel transfer counter register	B.2.6
GBLCNT	DMA global count reload register	B.2.7
GBLIDX	DMA global index register	B.2.8
GBLADDR	DMA global address reload register	B.2.9

B.2.1 DMA Auxiliary Control Register (AUXCTL)

Figure B–20. DMA Auxiliary Control Register (AUXCTL)



Legend: R/W-x = Read/Write-Reset value

Table B–22. DMA Auxiliary Control Register (AUXCTL) Field Values (DMA_AUXCTL_field_symval)

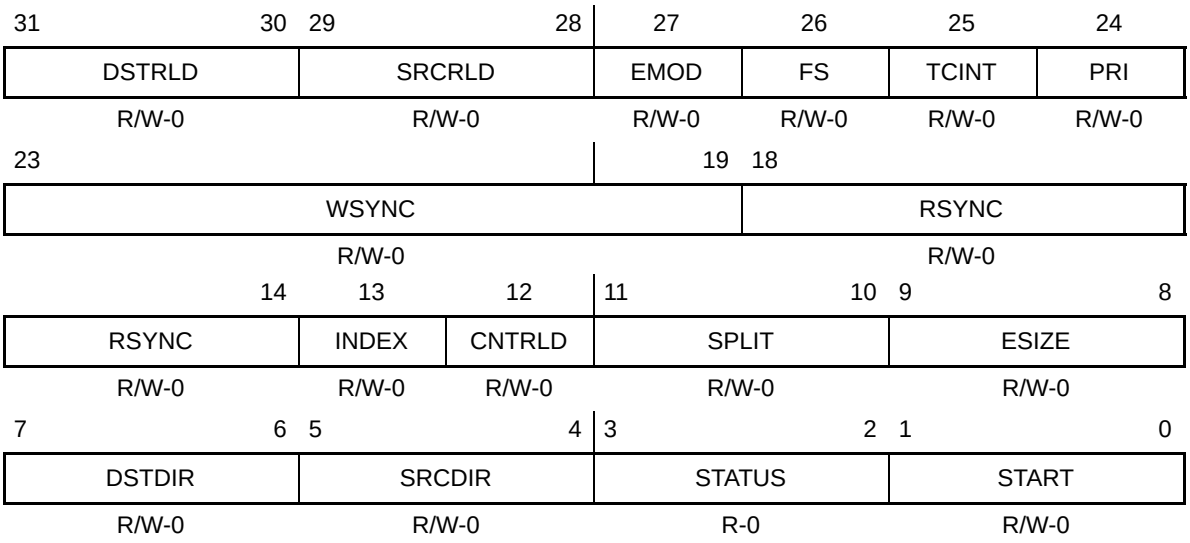
Bit	*field*	*symval*	Value	Description
31–5	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
4	AUXPRI			Auxiliary channel priority mode
		CPU	0	CPU priority
		DMA	1	DMA priority

Table B–22. DMA Auxiliary Control Register (AUXCTL) Field Values (DMA_AUXCTL_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
3–0	CHPRI			DMA channel priority. In the case when the auxiliary channel is used to service the expansion bus host port operation, CHPRI must be 0000b (auxiliary channel highest priority).
		HIGHEST	0000	Fixed channel priority mode auxiliary channel highest priority
		2ND	0001	Fixed channel priority mode auxiliary channel 2nd-highest priority
		3RD	0010	Fixed channel priority mode auxiliary channel 3rd-highest priority
		4TH	0011	Fixed channel priority mode auxiliary channel 4th-highest priority
		LOWEST	0100	Fixed channel priority mode auxiliary channel lowest priority
			0101–1111	Reserved

B.2.2 DMA Channel Primary Control Register (PRICTL)

Figure B–21. DMA Channel Primary Control Register (PRICTL)



Legend: R/W-x = Read/Write-Reset value

Table B–23. DMA Channel Primary Control Register (PRICTL) Field Values (DMA_PRICTL_field_symval)

Bit	*field*	*symval*	Value	Description
31–30	DSTRLD			Destination address reload for autoinitialization
		NONE	00	Do not reload during autoinitialization
		B	01	Use DMA global address register B as reload
		C	10	Use DMA global address register C as reload
		D	11	Use DMA global address register D as reload
29–28	SRCRLD			Source address reload for autoinitialization
		NONE	00	Do not reload during autoinitialization
		B	01	Use DMA global address register B as reload
		C	10	Use DMA global address register C as reload
		D	11	Use DMA global address register D as reload

Table B–23. DMA Channel Primary Control Register (PRICTL) Field Values (DMA_PRICTL_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
27	EMOD			Emulation mode
		NOHALT	0	DMA channel keeps running during an emulation halt
		HALT	1	DMA channel pauses during an emulation halt
26	FS			Frame synchronization
		DISABLE	0	Disable
		RSYNC	1	RSYNC event used to synchronize entire frame
25	TCINT			Transfer controller interrupt
		DISABLE	0	Interrupt disabled
		ENABLE	1	Interrupt enabled
24	PRI			Priority mode: DMA versus CPU
		CPU	0	CPU priority
		DMA	1	DMA priority
23–19	WSYNC			Write transfer synchronization
		NONE	00000	No synchronization
		TINT0	00001	Timer interrupt event 0
		TINT1	00010	Timer interrupt event 1
		SDINT	00011	
		EXTINT4	00100	External interrupt event 4
		EXTINT5	00101	External interrupt event 5
		EXTINT6	00110	External interrupt event 6
		EXTINT7	00111	External interrupt event 7
		DMAINT0	01000	DMA interrupt event 0
		DMAINT1	01001	DMA interrupt event 1
		DMAINT2	01010	DMA interrupt event 2
		DMAINT3	01011	DMA interrupt event 3

Table B–23. DMA Channel Primary Control Register (PRICTL) Field Values (DMA_PRICTL_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
		XEVT0	01100	McBSP 0 transmit event 0
		REVT0	01101	McBSP 0 receive event
		XEVT1	01110	McBSP 1 transmit event
		REVT1	01111	McBSP 1 receive event
		DSPINT	10000	DSP interrupt event
		XEVT2	10001	McBSP 2 transmit event
		REVT2	10010	McBSP 2 receive event
		1–18	10011–11111	Reserved
18–14	RSYNC			Read synchronization
		NONE	00000	No synchronization
		TINT0	00001	Timer interrupt event 0
		TINT1	00010	Timer interrupt event 1
		SDINT	00011	
		EXTINT4	00100	External interrupt event 4
		EXTINT5	00101	External interrupt event 5
		EXTINT6	00110	External interrupt event 6
		EXTINT7	00111	External interrupt event 7
		DMAINT0	01000	DMA interrupt event 0
		DMAINT1	01001	DMA interrupt event 1
		DMAINT2	01010	DMA interrupt event 2
		DMAINT3	01011	DMA interrupt event 3
		XEVT0	01100	McBSP 0 transmit event 0
		REVT0	01101	McBSP 0 receive event
		XEVT1	01110	McBSP 1 transmit event
		REVT1	01111	McBSP 1 receive event

Table B–23. DMA Channel Primary Control Register (PRICTL) Field Values (DMA_PRICTL_field_symval) (Continued)

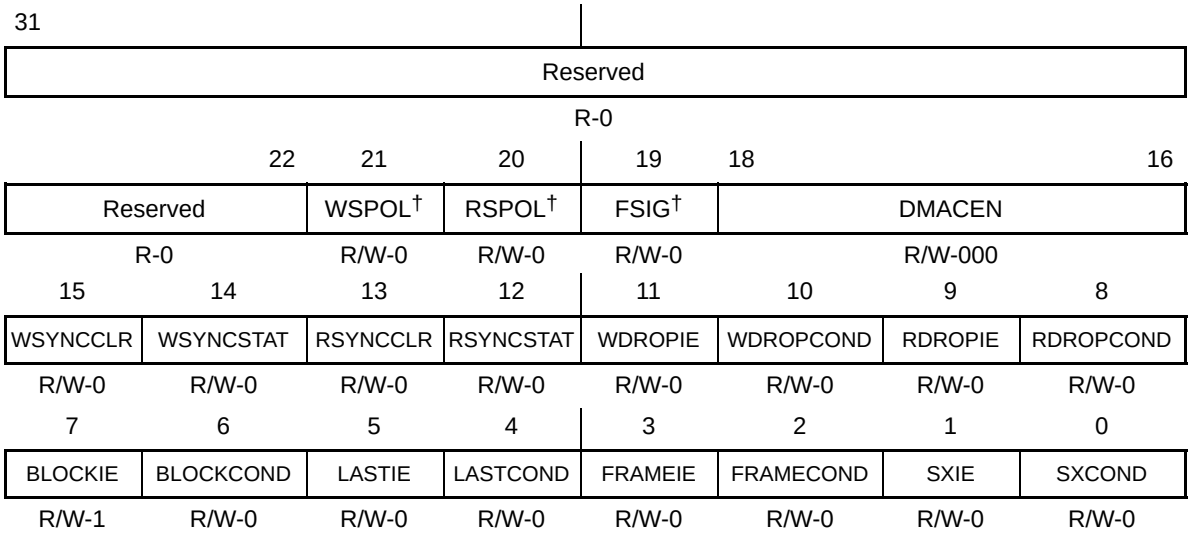
Bit	*field*	*symval*	Value	Description
		DSPINT	10000	DSP interrupt event
		XEVT2	10001	McBSP 2 transmit event
		REVT2	10010	McBSP 2 receive event
		1–18	10011–11111	Reserved
13	INDEX			Selects the DMA global data register to use as a programmable index
		A	0	Use DMA global index register A
		B	1	Use DMA global index register B
12	CNTRLD			Transfer counter reload for autoinitialization and multiframe transfers
		A	0	Reload with DMA global count reload register A
		B	1	Reload with DMA global count reload register B
11–10	SPLIT			Split channel mode
		DISABLE	00	Split-channel mode disabled
		A	01	Split-channel mode enabled; use DMA global address register A as split address
		B	10	Split-channel mode enabled; use DMA global address register B as split address
		C	11	Split-channel mode enabled; use DMA global address register C as split address
9–8	ESIZE			Element size
		32 BIT	00	32-bit
		16 BIT	01	16-bit
		8 BIT	10	8-bit
			11	Reserved

Table B–23. DMA Channel Primary Control Register (PRICTL) Field Values (DMA_PRICTL_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
7–6	DSTDIR			Destination address modification after element transfers
		NONE	00	No modification
		INC	01	Increment by element size in bytes
		DEC	10	Decrement by element size in bytes
		IDX	11	Adjust using DMA global index register selected by INDEX
5–4	SRCDIR			Source address modification after element transfers
		NONE	00	No modification
		INC	01	Increment by element size in bytes
		DEC	10	Decrement by element size in bytes
		IDX	11	Adjust using DMA global index register selected by INDEX
3–2	STATUS			
		STOPPED	00	Stopped
		RUNNING	01	Running without autoinitialization
		PAUSED	10	Paused
		AUTORUNNING	11	Running with autoinitialization
1–0	START			
		STOP	00	Stopped
		NORMAL	01	Running without autoinitialization
		PAUSE	10	Paused
		AUTOINIT	11	Running with autoinitialization

B.2.3 DMA Channel Secondary Control Register (SECCTL)

Figure B–22. DMA Channel Secondary Control Register (SECCTL)



† These bits are not available on the C6201 and C6701 devices. These bits are R+0 on the C6201 and C6701 devices.

Legend: R/W-x = Read/Write-Reset value

Table B–24. DMA Channel Secondary Control Register (SECCTL) Field Values (DMA_SECCTL_field_symval)

Bit	*field*	*symval*	Value	Description
31–22	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
21	WSPOL			Write synchronization event polarity (not applicable for C6201 and C6701 devices).
		ACTIVEHIGH	0	Active high
		ACTIVELOW	1	Active low
				This field is valid only if EXT_INTx is selected.
20	RSPOL			Read and frame synchronization event polarity (not applicable for C6201 and C6701 devices).
		ACTIVEHIGH	0	Active high
		ACTIVELOW	1	Active low
				This field is valid only if EXT_INTx is selected.

Table B–24. DMA Channel Secondary Control Register (SECCTL) Field Values (DMA_SECCTL_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
19	FSIG			Level/edge detect mode selection. FSIG must be cleared to 0 for non-frame-synchronized transfers (not applicable for C6201 and C6701 devices).
		NORMAL	0	Edge detect mode (FS = 1 or FS = 0).
		IGNORE	1	Level detect mode (valid only when FS = 1).
				In level detect mode, synchronization inputs received during a frame transfer are ignored unless still set after the frame transfer completes.
18–16	DMACEN			DMA action complete pins reflect status and condition.
		LOW	000	DMAC pin is held low.
		HIGH	001	DMAC pin is held high.
		RSYNCSTAT	010	DMAC reflects RSYNCSTAT.
		WSYNCSTAT	011	DMAC reflects WSYNCSTAT.
		FRAMECOND	100	DMAC reflects FRAMECOND.
		BLOCKCOND	101	DMAC reflects BLOCKCOND.
			110–111	Reserved
15	WSYNCCLR			Write synchronization status clear bit.
		NOTHING	0	
		CLEAR	1	Clear write synchronization status.
14	WSYNCSTAT			Write synchronization status.
		CLEAR	0	Synchronization is not received.
		SET	1	Synchronization is received.
13	RSYNCCLR			Read synchronization status clear bit.
		NOTHING	0	
		CLEAR	1	Clear read synchronization status.

Table B–24. DMA Channel Secondary Control Register (SECCTL) Field Values (DMA_SECCTL_field_symval) (Continued)

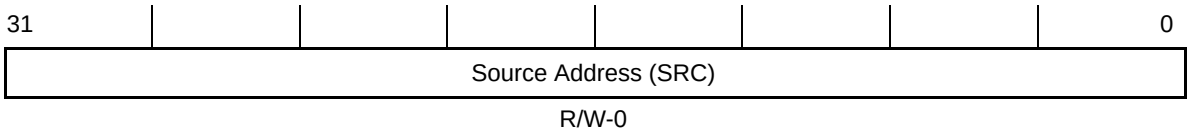
Bit	*field*	*symval*	Value	Description
12	RSYNCSTAT			Read synchronization status.
		CLEAR	0	Synchronization is not received.
		SET	1	Synchronization is received.
11	WDROPIE			Write synchronization dropped interrupt enable.
		DISABLE	0	WDROP condition does not enable DMA channel interrupt
		ENABLE	1	WDROP condition enables DMA channel interrupt.
10	WDROPCOND			Write drop condition.
		CLEAR	0	WDROP condition is not detected.
		SET	1	WDROP condition is detected.
9	RDROPIE			Read synchronization dropped interrupt enable.
		DISABLE	0	RDROP condition does not enable DMA channel interrupt
		ENABLE	1	RDROP condition enables DMA channel interrupt.
8	RDROPCOND			Read drop condition.
		CLEAR	0	RDROP condition is not detected.
		SET	1	RDROP condition is detected.
7	BLOCKIE			Block transfer finished interrupt enable.
		DISABLE	0	BLOCK condition does not enable DMA channel interrupt
		ENABLE	1	BLOCK condition enables DMA channel interrupt
6	BLOCKCOND			Block transfer finished condition.
		CLEAR	0	BLOCK condition is not detected.
		SET	1	BLOCK condition is detected.

Table B–24. DMA Channel Secondary Control Register (SECCTL) Field Values (DMA_SECCTL_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
5	LASTIE			Last frame finished interrupt enable.
		DISABLE	0	LAST condition does not enable DMA channel interrupt
		ENABLE	1	LAST condition enables DMA channel interrupt
4	LASTCOND			Last frame finished condition.
		CLEAR	0	LAST condition is not detected.
		SET	1	LAST condition is detected.
3	FRAMEIE			Frame complete interrupt enable.
		DISABLE	0	FRAME condition does not enable DMA channel interrupt
		ENABLE	1	FRAME condition enables DMA channel interrupt
2	FRAMECOND			Frame complete condition.
		CLEAR	0	FRAME condition is not detected.
		SET	1	FRAME condition is detected.
1	SXIE			Split transmit overrun receive interrupt enable.
		DISABLE	0	SX condition does not enable DMA channel interrupt
		ENABLE	1	SX condition enables DMA channel interrupt
0	SXCOND			Split transmit condition.
		CLEAR	0	SX condition is not detected.
		SET	1	SX condition is detected.

B.2.4 DMA Channel Source Address Register (SRC)

Figure B–23. DMA Channel Source Address Register (SRC)



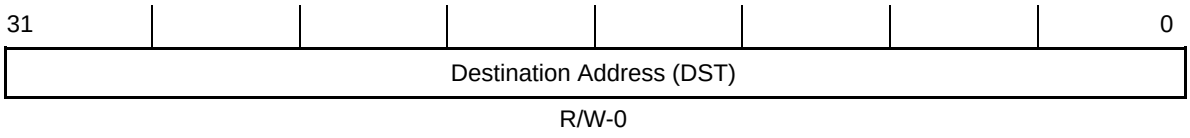
Legend: R/W-x = Read/Write-Reset value

Table B–25. DMA Channel Source Address Register (SRC) Field Values (DMA_SRC_field_symval)

Bit	*field*	*symval*	Value	Description
31–0	SRC	OF(*value*)	0–FFFFFFFFh	

B.2.5 DMA Channel Destination Address Register (DST)

Figure B–24. DMA Channel Destination Address Register (DST)



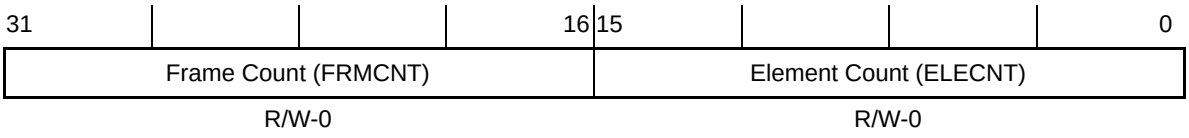
Legend: R/W-x = Read/Write-Reset value

Table B–26. DMA Channel Destination Address Register (DST) Field Values (DMA_DST_field_symval)

Bit	*field*	*symval*	Value	Description
31–0	DST	OF(*value*)	0–FFFFFFFFh	

B.2.6 DMA Channel Transfer Counter Register (XFRCNT)

Figure B–25. DMA Channel Transfer Counter Register (XFRCNT)



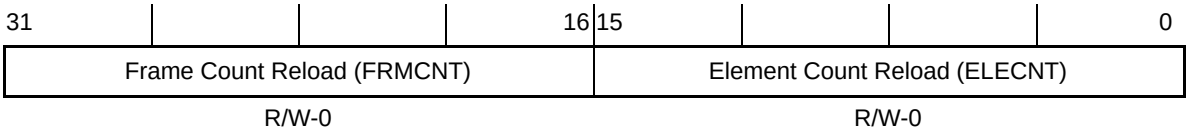
Legend: R/W-x = Read/Write-Reset value

Table B–27. DMA Channel Transfer Counter Register (XFRCNT) Field Values (DMA_XFRCNT_field_symval)

Bit	*field*	*symval*	Value	Description
31–16	FRMCNT	OF(value)	0–FFFFh	
15–0	ELECNT	OF(value)	0–FFFFh	

B.2.7 DMA Global Count Reload Register (GBLCNT)

Figure B–26. DMA Global Count Reload Register (GBLCNT)



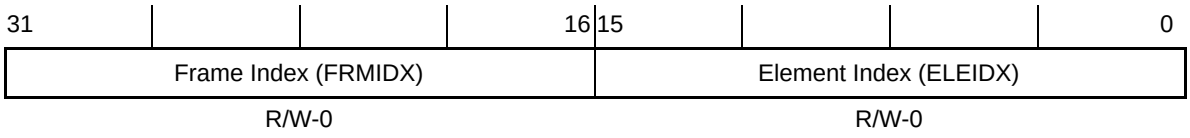
Legend: R/W-x = Read/Write-Reset value

Table B–28. DMA Global Count Reload Register (GBLCNT) Field Values (DMA_GBLCNT_field_symval)

Bit	*field*	*symval*	Value	Description
31–16	FRMCNT	OF(value)	0–FFFFh	This 16-bit value reloads FRMCNT bits in XFRCNT.
15–0	ELECNT	OF(value)	0–FFFFh	This 16-bit value reloads ELECNT bits in XFRCNT.

B.2.8 DMA Global Index Register (GBLIDX)

Figure B–27. DMA Global Index Register (GBLIDX)



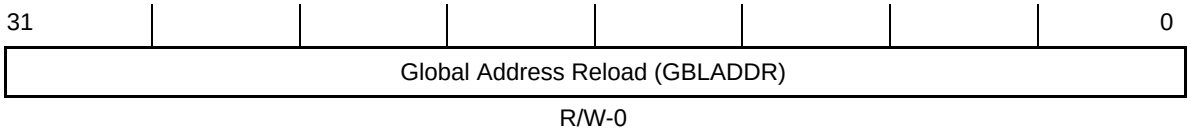
Legend: R/W-x = Read/Write-Reset value

Table B–29. DMA Global Index Register (GBLIDX) Field Values (DMA_GBLIDX_field_symval)

Bit	*field*	*symval*	Value	Description
31–16	FRMIDX	OF(*value*)	0–FFFFh	
15–0	ELEIDX	OF(*value*)	0–FFFFh	

B.2.9 DMA Global Address Reload Register (GBLADDR)

Figure B–28. DMA Global Address Reload Register (GBLADDR)



Legend: R/W-x = Read/Write-Reset value

Table B–30. DMA Global Address Reload Register (GBLADDR) Field Values (DMA_GBLADDR_field_symval)

Bit	*field*	*symval*	Value	Description
31–0	GBLADDR	OF(*value*)	0–FFFFFFFFh	

B.3 Enhanced DMA (EDMA) Registers

Table B–31. EDMA Registers

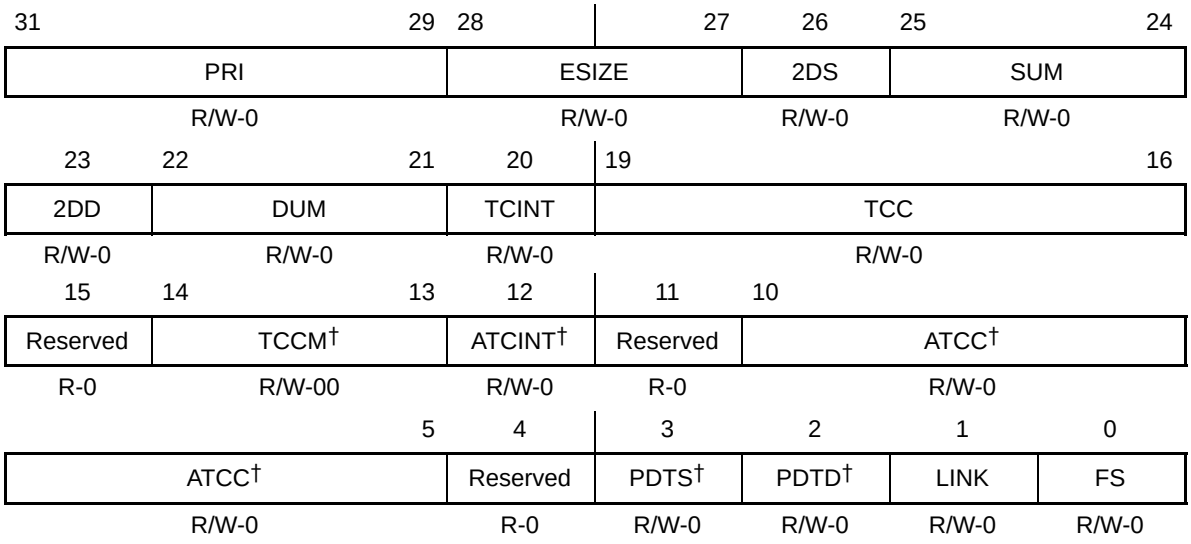
Acronym	Register Name	Section
OPT	EDMA channel options register	B.3.1
SRC	EDMA channel source address register	B.3.2
CNT	EDMA channel transfer count register	B.3.3
PQSR	Priority queue status register (C621x/C671x)	B.3.4
PQSR	Priority queue status register (C64x)	B.3.5
DST	EDMA channel destination address register	B.3.6
CIPR	EDMA channel interrupt pending register (C621x/C671x)	B.3.7
CIPRL	EDMA channel interrupt pending low register (C64x)	B.3.8
CIPRH	EDMA channel interrupt pending high register (C64x)	B.3.9
IDX	EDMA channel index register	B.3.10
RLD	EDMA channel count reload/link register	B.3.11
CIER	EDMA channel interrupt enable register (C621x/C671x)	B.3.12
CIERL	EDMA channel interrupt enable low register (C64x)	B.3.13
CIERH	EDMA channel interrupt enable high register (C64x)	B.3.14
CCER	EDMA channel chain enable register (C621x/C671x)	B.3.15
CCERL	EDMA channel chain enable low register (C64x)	B.3.16
CCERH	EDMA channel chain enable high register (C64x)	B.3.17
ER	EDMA event register (C621x/C671x)	B.3.18
ERL	EDMA event low register (C64x)	B.3.19
ERH	EDMA event high register (C64x)	B.3.20
EER	EDMA event enable register (C621x/C671x)	B.3.21
EERL	EDMA event enable low register (C64x)	B.3.22
EERH	EDMA event enable high register (C64x)	B.3.23
ECR	EDMA event clear register (C621x/C671x)	B.3.24
ECRL	EDMA event clear low register (C64x)	B.3.25

Table B–31. EDMA Registers (Continued)

Acronym	Register Name	Section
ECRH	EDMA event clear high register (C64x)	B.3.26
ESR	EDMA event set register (C621x/C671x)	B.3.27
ESRL	EDMA event set low register (C64x)	B.3.28
ESRH	EDMA event set high register (C64x)	B.3.29

B.3.1 EDMA Channel Options Register (OPT)

Figure B–29. EDMA Channel Options Register (OPT)



† Applies to C64x only. On C621x/C671x, these bits are Reserved, R+0.

Legend: R/W-x = Read/Write-Reset value

Table B–32. EDMA Channel Options Register (OPT) Field Value (EDMA_OPT_field_symval)

Bit	*field*	*symval*	Value	Description
31–29	PRI			Priority levels for EDMA events
				For C64x:
		URGENT	000	Urgent priority.
		HIGH	001	This level is available for CPU and EDMA transfer requests.
		MEDIUM	010	Medium priority EDMA transfer

Table B–32. EDMA Channel Options Register (OPT) Field Value (EDMA_OPT_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
		LOW	011	Low priority EDMA transfer
				For C62x, C67x:
		HIGH	001	This level is Reserved only for L2 requests and not valid for EDMA transfer requests.
		LOW	010	Low priority EDMA transfer requests.
28–27	ESIZE			Element size
		32 BIT	00	32-bit word
		16 BIT	01	16-bit half-word
		8 BIT	10	8-bit byte
			11	Reserved
26	2DS			Source dimension
		NO	0	1-dimensional source.
		YES	1	2-dimensional source.
25–24	SUM			Source address update mode
		NONE	00	Fixed address mode. No source address modification
		INC	01	Source address increment depends on 2DS and FS bits
		DEC	10	Source address decrement depends on 2DS and FS bits
		IDX	11	Source address modified by the element index/frame index depending on 2DS and FS bits.
23	2DD			Destination dimension
		NO	0	1-dimensional destination
		YES	1	2-dimensional destination
22–21	DUM			Destination address update mode
		NONE	00	Fixed address mode. No destination address modification
		INC	01	Destination address increment depends on 2DD and FS bits
		DEC	10	Destination address decrement depends on 2DD and FS bits

Table B–32. EDMA Channel Options Register (OPT) Field Value (EDMA_OPT_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
		IDX	11	Destination address modified by the element index/frame index depending on 2DD and FS bits.
20	TCINT			Transfer complete interrupt
		NO	0	Transfer complete indication disabled. CIPR bits are not set upon completion of a transfer.
		YES	1	The relevant CIPR bit is set on channel transfer completion. The bit (position) set in the CIPR is the TCC value specified.
19–16	TCC	OF(*value*)	0–15	Transfer complete code 4-bit code is used to set the relevant bit in CIPR (i.e. CIPR[TCC] bit) provided. For C64x, the 4-bit TCC code is used in conjunction with bit field TCCM for a 6-bit transfer complete code.
15	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
14–13	TCCM	OF(*value*)	0–3	
12	ATCINT			
		NO	0	
		YES	1	
11	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
10–5	ATCC	OF(*value*)	0–63	
4	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
3	PDTS			
		DISABLE	0	
		ENABLE	1	
2	PDTD			
		DISABLE	0	
		ENABLE	1	

Table B–32. EDMA Channel Options Register (OPT) Field Value (EDMA_OPT_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
1	LINK			Link
		NO	0	Linking of event parameters disabled. Entry not reloaded.
		YES	1	Linking of event parameters enabled.
				After the current set is exhausted, the channel entry is reloaded with the parameter set specified by the link address. The link address must be on a 24-byte boundary and within the EDMA PaRAM. The link address is a 16-bit address offset from the PaRAM base address.
0	FS			Frame synchronization
		NO	0	Channel is element/array synchronized.
		YES	1	Channel is frame synchronized. The relevant event for a given EDMA channel is used to synchronize a frame.

B.3.2 EDMA Channel Source Address Register (SRC)

Figure B–30. EDMA Channel Source Address Register (SRC)

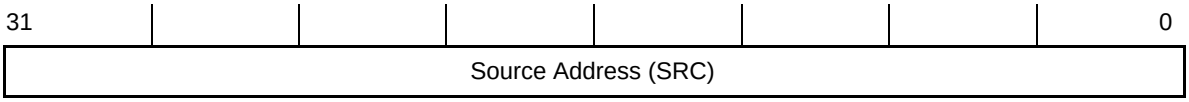


Table B–33. EDMA Channel Source Address Register (SRC) Field Values (EDMA_SRC_field_symval)

Bit	*field*	*symval*	Value	Description
31–0	SRC	OF(*value*)	0–FFFFFFFFh	

B.3.3 EDMA Channel Transfer Count Register (CNT)

Figure B–31. EDMA Channel Transfer Count Register (CNT)

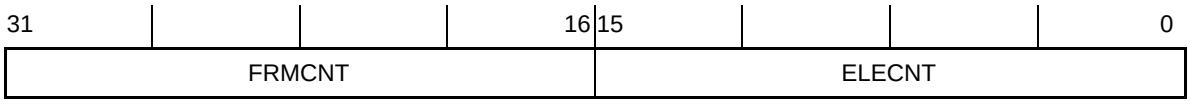
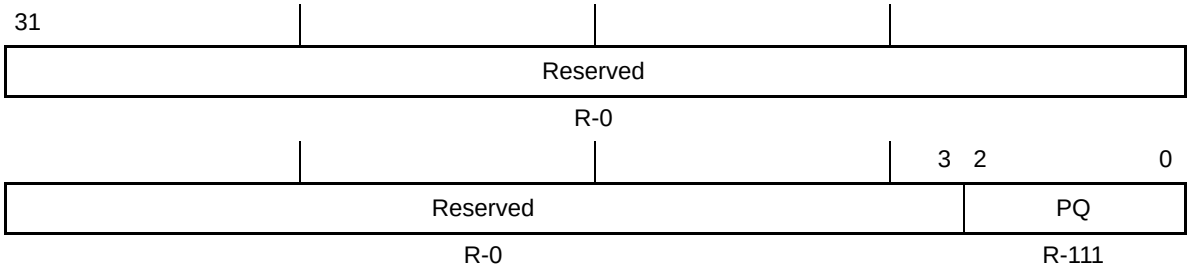


Table B–34. EDMA Channel Transfer Count Register (CNT) Field Values (EDMA_CNT_field_symval)

Bit	*field*	*symval*	Value	Description
31–16	FRMCNT	OF(*value*)	0–FFFFh	
15–0	ELECNT	OF(*value*)	0–FFFFh	

B.3.4 Priority Queue Status Register(PQSR) (C621x/C671x)

Figure B–32. Priority Queue Status Register (PQSR)



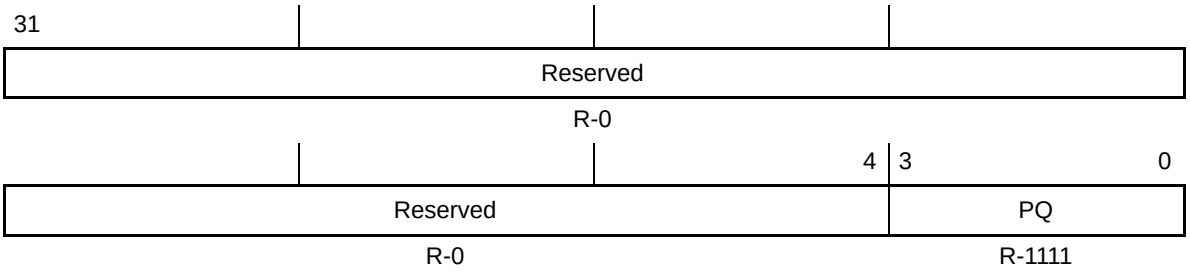
Legend: R/W-x = Read/Write-Reset value

Table B–35. Priority Queue Status Register (PQSR) Field Values (EDMA_PQSR_field_symval)

Bit	*field*	*symval*	Value	Description
31–3	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
2–0	PQ	OF(*value*)	0–7	

B.3.5 Priority Queue Status Register(PQSR) (C64x)

Figure B–33. Priority Queue Status Register (PQSR)



Legend: R/W-x = Read/Write-Reset value

Table B–36. Priority Queue Status Register (PQSR) Field Values (EDMA_PQSR_field_symval)

Bit	*field*	*symval*	Value	Description
31–4	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
3–0	PQ	OF(*value*)	0–15	

B.3.6 EDMA Channel Destination Address Register (DST)

Figure B–34. EDMA Channel Destination Address Register (DST)

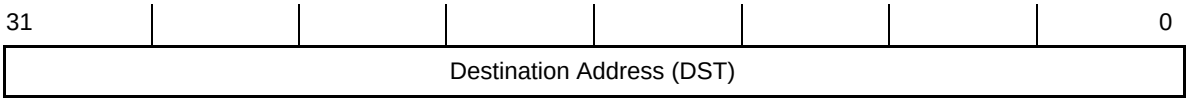
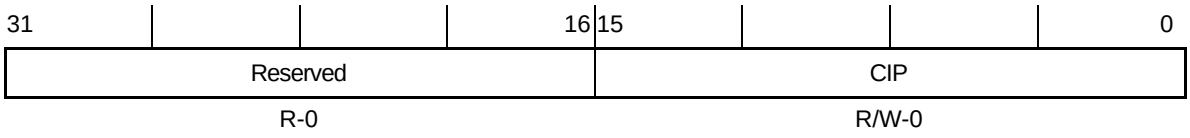


Table B–37. EDMA Channel Destination Address Register (DST) Field Values (EDMA_DST_field_symval)

Bit	*field*	*symval*	Value	Description
31–0	DST	OF(*value*)	0–FFFFFFFFh	

B.3.7 EDMA Channel Interrupt Pending Register (CIPR) (C621x/C671x)

Figure B–35. EDMA Channel Interrupt Pending Register (CIPR)



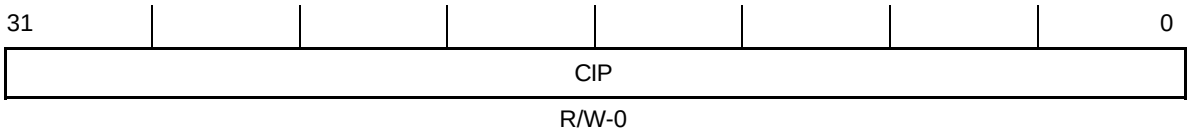
Legend: R/W-x = Read/Write-Reset value

Table B–38. EDMA Channel Interrupt Pending Register (CIPR) Field Values (EDMA_CIPR_field_symval)

Bit	*field*	*symval*	Value	Description
31–16	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
15–0	CIP	OF(*value*)	0–FFFFh	

B.3.8 EDMA Channel Interrupt Pending Low Register (CIPRL) (C64x)

Figure B–36. EDMA Channel Interrupt Pending Low Register (CIPRL)



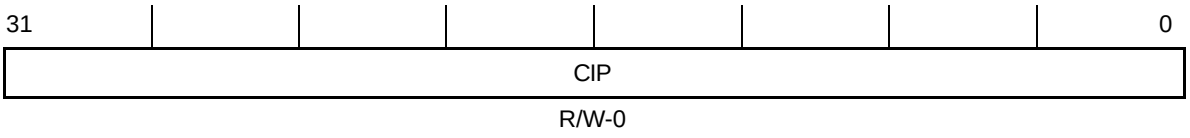
Legend: R/W-x = Read/Write-Reset value

Table B–39. EDMA Channel Interrupt Pending Low Register (CIPRL) Field Values (EDMA_CIPRL_field_symval)

Bit	*field*	*symval*	Value	Description
31–0	CIP	OF(*value*)	0–FFFFFFFFh	

B.3.9 EDMA Channel Interrupt Pending High Register (CIPRH) (C64x)

Figure B–37. EDMA Channel Interrupt Pending High Register (CIPRH)



Legend: R/W-x = Read/Write-Reset value

Table B–40. EDMA Channel Interrupt Pending High Register (CIPRH) Field Values (EDMA_CIPRH_field_symval)

Bit	*field*	*symval*	Value	Description
31–0	CIP	OF(*value*)	0–FFFFFFFFh	

B.3.10 EDMA Channel Index Register (IDX)

Figure B–38. EDMA Channel Index Register (IDX)

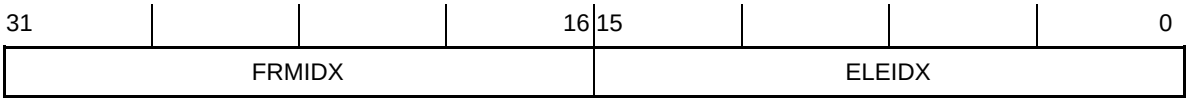


Table B–41. EDMA Channel Index Register (IDX) Field Values (EDMA_IDX_field_symval)

Bit	*field*	*symval*	Value	Description
31–16	FRMIDX	OF(*value*)	0–FFFFh	
15–0	ELEIDX	OF(*value*)	0–FFFFh	

B.3.11 EDMA Channel Count Reload/Link Register (RLD)

Figure B–39. EDMA Channel Count Reload/Link Register (RLD)

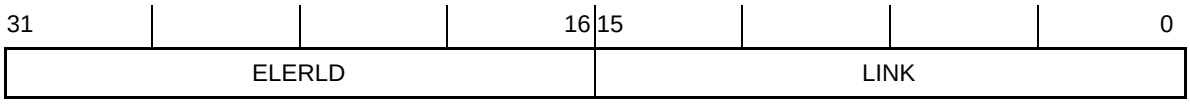
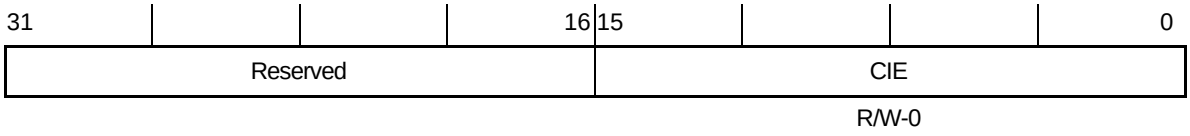


Table B–42. EDMA Channel Count Reload/Link Register (RLD) Field Values (EDMA_RLD_field_symval)

Bit	*field*	*symval*	Value	Description
31–16	ELERLD	OF(*value*)		
15–0	LINK	OF(*value*)		

B.3.12 EDMA Channel Interrupt Enable Register (CIER) (C621x/C671x)

Figure B–40. EDMA Channel Interrupt Enable Register (CIER)



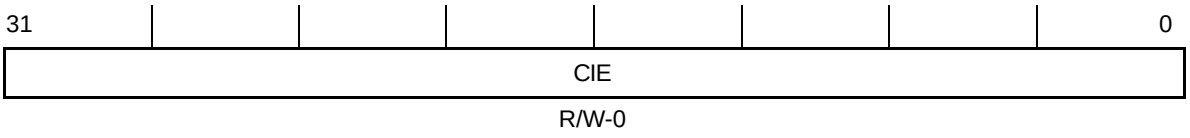
Legend: R/W-x = Read/Write-Reset value

Table B–43. C621x/C671x: Channel Interrupt Enable Register (CIER) Field Values (EDMA_CIER_field_symval)

Bit	*field*	*symval*	Value	Description
31–16	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
15–0	CIE	OF(*value*)	0–FFFFh	

B.3.13 EDMA Channel Interrupt Enable Low Register (CIERL) (C64x)

Figure B–41. EDMA Channel Interrupt Enable Low Register (CIERL)



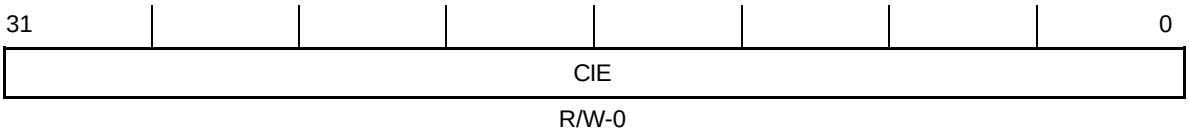
Legend: R/W-x = Read/Write-Reset value

Table B–44. EDMA Channel Interrupt Enable Low Register (CIERL) Field Values (EDMA_CIERL_field_symval)

Bit	*field*	*symval*	Value	Description
31–0	CIE	OF(*value*)	0–FFFFFFFFh	

B.3.14 EDMA Channel Interrupt Enable High Register (CIERH) (C64x)

Figure B–42. EDMA Channel Interrupt Enable High Register (CIERH)



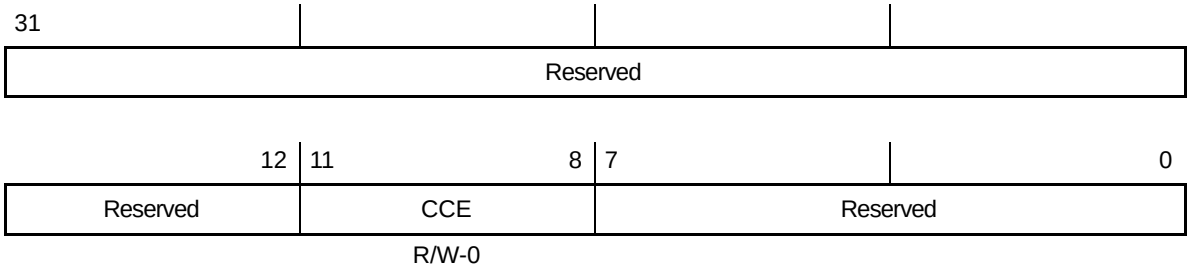
Legend: R/W-x = Read/Write-Reset value

Table B–45. EDMA Channel Interrupt Enable High Register (CIERH) Field Values (EDMA_CIERH_field_symval)

Bit	*field*	*symval*	Value	Description
31–0	CIE	OF(*value*)	0–FFFFFFFFh	

B.3.15 EDMA Channel Chain Enable Register (CCER) (C621x/C671x)

Figure B–43. EDMA Channel Chain Enable Register (CCER)



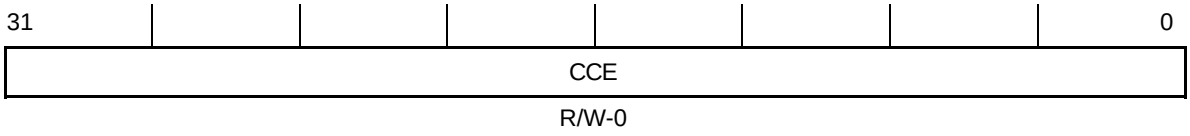
Legend: R/W-x = Read/Write-Reset value

Table B–46. EDMA Channel Chain Enable Register (CCER) Field Values (EDMA_CCER_field_symval)

Bit	*field*	*symval*	Value	Description
31–12	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
11–8	CCE	OF(*value*)	0–15	
7–0	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.

B.3.16 EDMA Channel Chain Enable Low Register (CCERL) (C64x)

Figure B–44. EDMA Channel Chain Enable Low Register (CCERL)



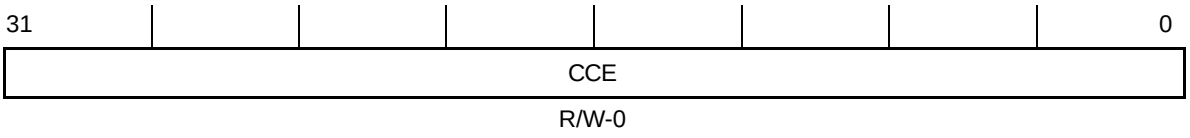
Legend: R/W-x = Read/Write-Reset value

Table B–47. EDMA Channel Chain Enable Low Register (CCERL) Field Values (EDMA_CCERL_field_symval)

Bit	*field*	*symval*	Value	Description
31–0	CCE	OF(*value*)	0–FFFFFFFFh	

B.3.17 EDMA Channel Chain Enable High Register (CCERH) (C64x)

Figure B–45. EDMA Channel Chain Enable High Register (CCERH)



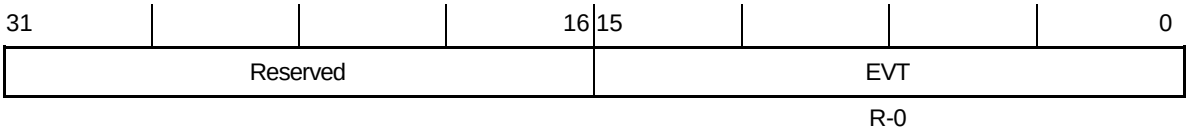
Legend: R/W-x = Read/Write-Reset value

Table B–48. EDMA Channel Chain Enable High Register (CCERH) Field Values (EDMA_CCERH_field_symval)

Bit	*field*	*symval*	Value	Description
31–0	CCE	OF(*value*)	0–FFFFFFFFh	

B.3.18 EDMA Event Register (ER) (C621x/C671x)

Figure B–46. EDMA Event Register (ER)



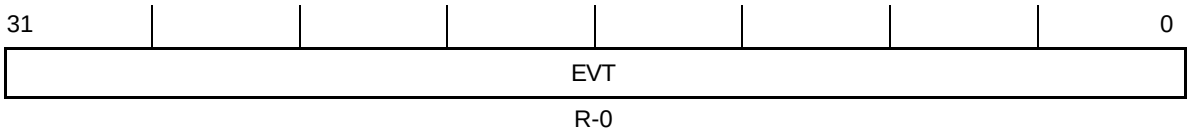
Legend: R/W-x = Read/Write-Reset value

Table B–49. EDMA Event Register (ER) Field Values (EDMA_ER_field_symval)

Bit	*field*	*symval*	Value	Description
31–16	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
15–0	EVT	OF(*value*)	0–FFFFh	

B.3.19 EDMA Event Low Register (ERL) (C64x)

Figure B–46. EDMA Event Low Register (ERL)



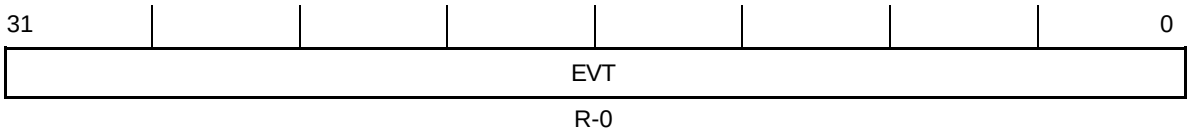
Legend: R/W-x = Read/Write-Reset value

Table B–50. EDMA Event Low Register (ERL) Field Values (EDMA_ERL_field_symval)

Bit	*field*	*symval*	Value	Description
31–0	EVT	OF(*value*)	0–FFFFFFFFh	

B.3.20 EDMA Event High Register (ERH) (C64x)

Figure B–47. 1EDMA Event High Register (ERH)



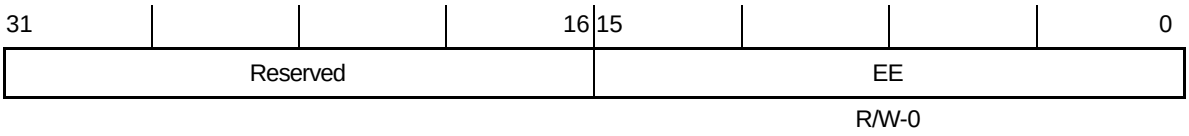
Legend: R/W-x = Read/Write-Reset value

Table B–51. EDMA Event High Register (ERH) Field Values (EDMA_ERH_field_symval)

Bit	*field*	*symval*	Value	Description
31–0	EVT	OF(*value*)	0–FFFFFFFFh	

B.3.21 EDMA Event Enable Register (EER) (C621x/C671x)

Figure B–48. EDMA Event Enable Register (EER)



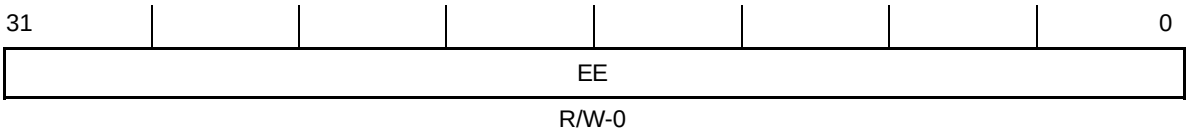
Legend: R/W-x = Read/Write-Reset value

Table B–52. EDMA Event Enable Register (EER) Field Values (EDMA_EER_field_symval)

Bit	*field*	*symval*	Value	Description
31–16	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
15–0	EE	OF(*value*)	0–FFFFh	

B.3.22 EDMA Event Enable Low Register (EERL) (C64x)

Figure B–49. EDMA Event Enable Low Register (EERL)



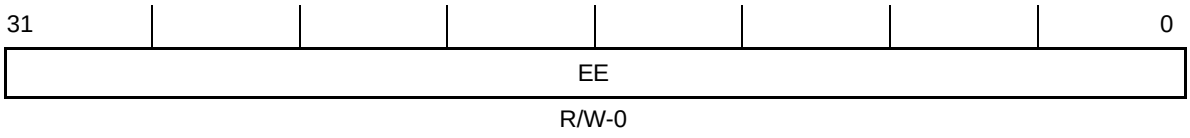
Legend: R/W-x = Read/Write-Reset value

Table B–53. EDMA Event Low Register (EERL) Field Values (EDMA_EERL_field_symval)

Bit	*field*	*symval*	Value	Description
31–0	EE	OF(*value*)	0–FFFFFFFFh	

B.3.23 EDMA Event Enable High Register (EERH) (C64x)

Figure B–50. EDMA Event Enable High Register (EERH)



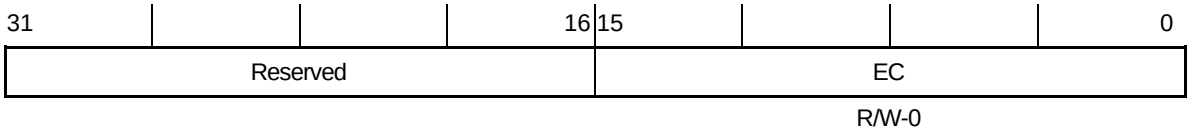
Legend: R/W-x = Read/Write-Reset value

Table B–54. EDMA Event Enable High Register (EERH) Field Values (EDMA_EERH_field_symval)

Bit	*field*	*symval*	Value	Description
31–0	EE	OF(*value*)	0–FFFFFFFFh	

B.3.24 EDMA Event Clear Register (ECR) (C621x/C671x)

Figure B–51. EDMA Event Clear Register (ECR)



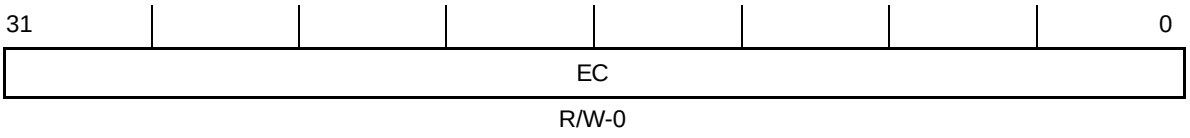
Legend: R/W-x = Read/Write-Reset value

Table B–55. EDMA Event Clear Register (ERC) Field Values (EDMA_ECR_field_symval)

Bit	*field*	*symval*	Value	Description
31–16	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
15–0	EC	OF(*value*)	0–FFFFh	

B.3.25 EDMA Event Clear Low Register (ECRL) (C64x)

Figure B–52. EDMA Event Clear Low Register (ECRL)



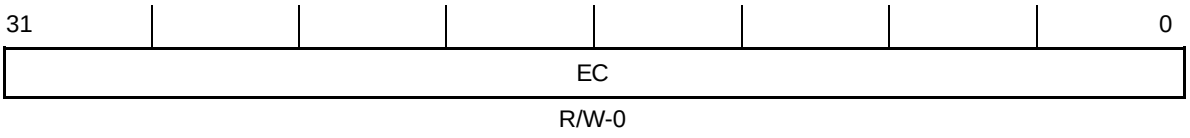
Legend: R/W-x = Read/Write-Reset value

Table B–56. EDMA Event Clear Low Register (ERCL) Field Values (EDMA_ECRL_field_symval)

Bit	*field*	*symval*	Value	Description
31–0	EC	OF(*value*)	0–FFFFFFFFh	

B.3.26 EDMA Event Clear High Register (ECRH) (C64x)

Figure B–53. EDMA Event Clear High Register (ECRH)



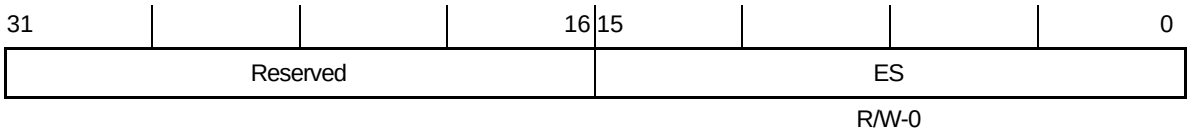
Legend: R/W-x = Read/Write-Reset value

Table B–57. EDMA Event Clear High Register (ECRH) Field Values (EDMA_ECRH_field_symval)

Bit	*field*	*symval*	Value	Description
31–0	EC	OF(*value*)	0–FFFFFFFFh	

B.3.27 EDMA Event Set Register (ESR) (C621x/C671x)

Figure B–54. EDMA Event Set Register (ESR)



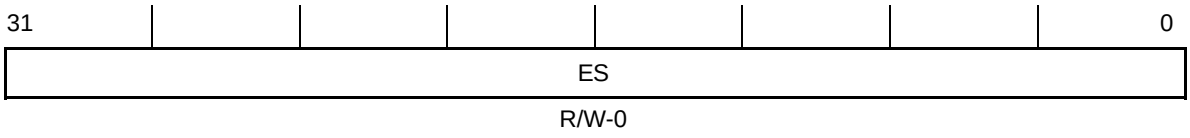
Legend: R/W-x = Read/Write-Reset value

Table B–58. EDMA Event Set Register (ESR) Field Values (EDMA_ESR_field_symval)

Bit	*field*	*symval*	Value	Description
31–16	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
15–0	ES	OF(*value*)	0–FFFFh	

B.3.28 EDMA Event Set Low Register (ESRL) (C64x)

Figure B–55. EDMA Event Set Low Register (ESRL)



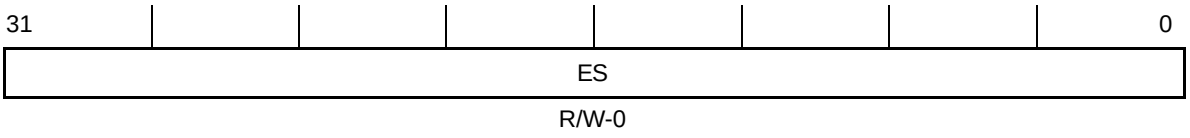
Legend: R/W-x = Read/Write-Reset value

Table B–59. EDMA Event Set Low Register (ESRL) Field Values (EDMA_ESRL_field_symval)

Bit	*field*	*symval*	Value	Description
31–0	ES	OF(*value*)	0–FFFFFFFFh	

B.3.29 EDMA Event Set High Register (ESRH) (C64x)

Figure B–56. EDMA Event Set High Register (ESRH)



Legend: R/W-x = Read/Write-Reset value

Table B–60. EDMA Event Set High Register (ESRH) Field Values (EDMA_ESRH_field_symval)

Bit	*field*	*symval*	Value	Description
31–0	ES	OF(*value*)	0–FFFFFFFFh	

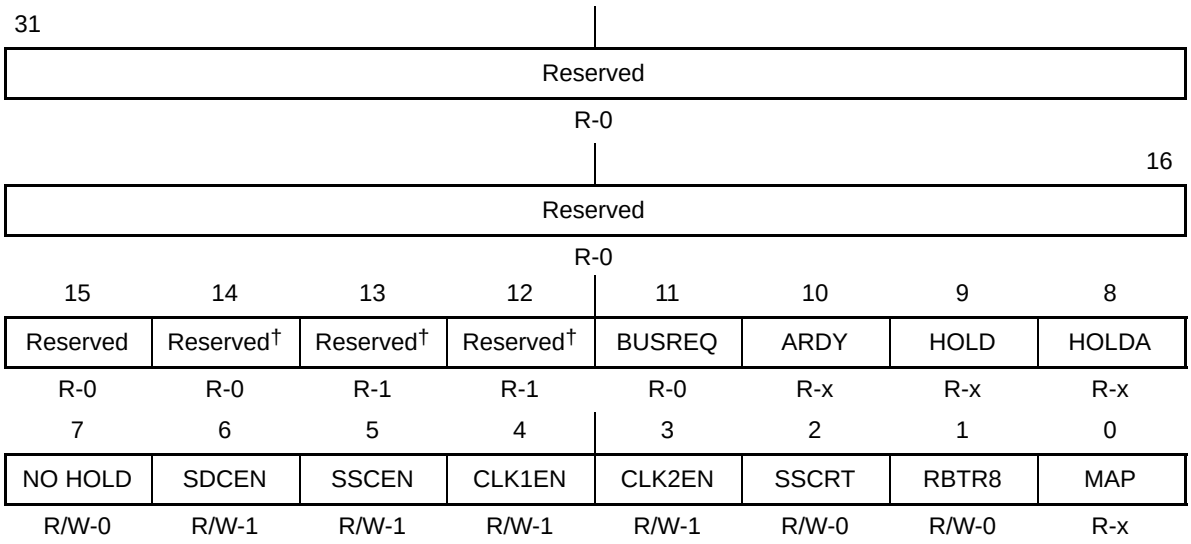
B.4 External Memory Interface (EMIF) Registers

Table B–61. EMIF Registers

Acronym	Register Name	Section
GBLCTL	EMIF global control register (C6201/C6701)	B.4.1
CECTL	EMIF CE space control register (C620x/C670x)	B.4.2
CECTL	EMIF CE space control register (C621x/C671x/C64x)	B.4.3
SDCTL	EMIF SDRAM control register (C620x/C670x)	B.4.4
SDCTL	EMIF SDRAM control register (C621x/C671x/C64x)	B.4.5
SDCTL	EMIF SDRAM control register (C664x, EMIFA/EMIFB only)	B.4.6
SDTIM	EMIF SDRAM timing register	B.4.7
SDEXT	EMIF SDRAM extension register (C621x/C671x/C64x)	B.4.8

B.4.1 EMIF Global Control Register (GBLCTL) (C6201/C6701)

Figure B–57. EMIF Global Control Register (GBLCTL)



† Group1 devices include: C6201/C6701.
Group 2 devices include: all C620x/C670x *except* C6201/C6701.
Group 3 devices include: C621x/C671x.
Group 4 devices include: C64x EMIFA.
Group 5 devices include: C64x EMIFB.

Legend: R/W-x = Read/Write-Reset value

Table B–62. EMIF Global Control Register (GBLCTL) Field Values (EMIF_GBLCTL_field_symval)

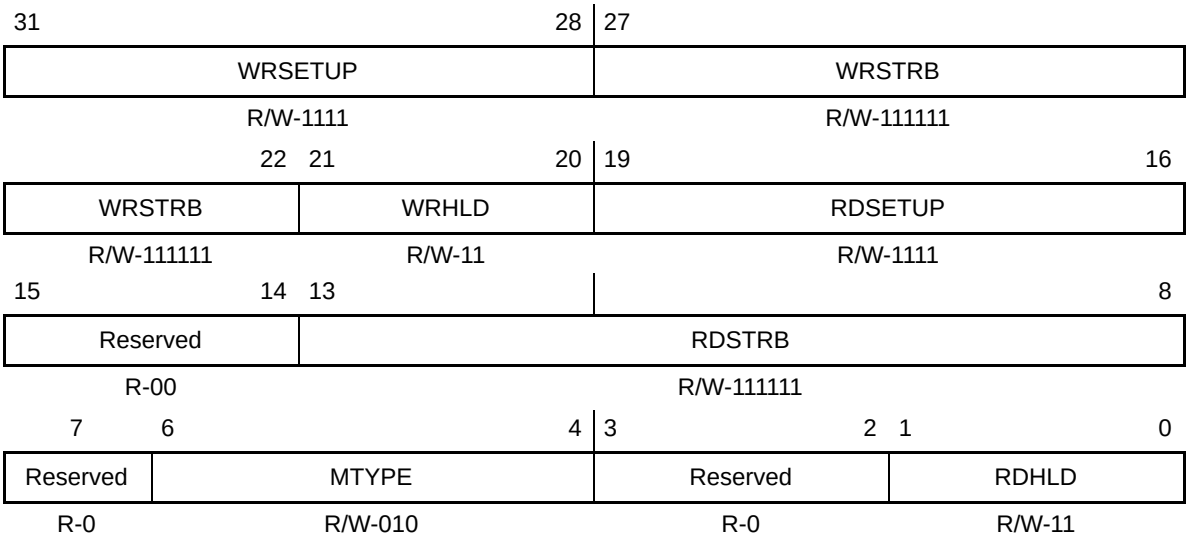
Bit	*field*	*symval*	Value	Description
31–16	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
15	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
14	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
13	Reserved			Reserved. The reserved bit location is always set to one. A value written to this field has no effect.
12	Reserved			Reserved. The reserved bit location is always set to one. A value written to this field has no effect.
11	BUSREQ			
		LOW	0	Bus request is low.
		HIGH	1	Bus request is high.
10	ARDY			
		LOW	0	ARDY input is low. External device not ready
		HIGH	1	ARDY input is high. External device ready
9	HOLD			
		LOW	0	$\overline{\text{HOLD}}$ input is low. External device requesting EMIF
		HIGH	1	$\overline{\text{HOLD}}$ input is high. No external request pending
8	HOLDA			
		LOW	0	$\overline{\text{HOLDA}}$ output is low. External device owns EMIF
		HIGH	1	$\overline{\text{HOLDA}}$ output is high. External device does not own EMIF
7	NOHOLD			External HOLD disable
		DISABLE	0	Hold enabled
		ENABLE	1	Hold disabled
6	SDCEN			SDCLK enable
		DISABLE	0	SDCLK held high
		ENABLE	1	SDCLK enabled to clock

Table B–62. EMIF Global Control Register (GBLCTL) Field Values (EMIF_GBLCTL_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
				SDCEN enables CLKOUT2 on C6202(B)/C6203(B)/C6204/C6205 if SDRAM is used in system (specified by the MTYPE field in in the CE space control register).
5	SSCEN			SSCLK enable
		DISABLE	0	SSCLK held high
		ENABLE	1	SSCLK enabled to clock
				SSCEN enables CLKOUT2 on the C6202(B)/C6203(B)/C6204/C6205 if SBSRAM is used in the system (specified by the MTYPE field in in the CE space control register).
4	CLK1EN			CLKOUT1 enable
		DISABLE	0	CLKOUT1 held high
		ENABLE	1	CLKOUT1 enabled to clock
3	CLK2EN			CLKOUT2 enable
		DISABLE	0	CLKOUT2 held high
		ENABLE	1	CLKOUT2 enabled to clock
				CLKOUT2 is enabled/disabled using SSCEN/SDCEN on the C6202(B)/C6203(B)/C6204/C6205
2	SSCRT			SBSRAM clock rate select
		CPUOVR2	0	SSCLK runs at 1/2 CPU clock rate
		CPU	1	SSCLK runs at CPU clock rate
1	RBTR8			Requester arbitration mode
		HPRI	0	
		8ACC	1	
0	MAP			Map mode, contains the value of the memory map mode of the device
		MAP1	0	Map 0 selected. External memory located at address 0
		MAP0	1	Map 1 selected. Internal memory located at address 0.

B.4.2 EMIF CE Space Control Register (CECTL) (C620x/C670x)

Figure B–58. EMIF CE Space Control Register (CECTL)



Legend: R/W-x = Read/Write-Reset value

Table B–63. EMIF CE Space Control Register (CECTL) Field Values (EMIF_CECTL_field_symval)

Bit	*field*	*symval*	Value	Description
31–28	WRSETUP	OF(*value*)	0–15	Write setup width. Number of clock† cycles of setup time for address (EA), chip enable ($\overline{CE}$), and byte enables ($\overline{BE}$[0-3]) before write strobe falls. For asynchronous read accesses, this is also the setup time of $\overline{AOE}$ before $\overline{ARE}$ falls.
27–22	WRSTRB	OF(*value*)	0–63	Write strobe width. The width of write strobe ($\overline{AWE}$) in clock† cycles
21–20	WRHLD	OF(*value*)	0–3	Write hold width. Number of clock† cycles that address (EA) and byte strobes ($\overline{BE}$[0-3]) are held after write strobe rises. For asynchronous read accesses, this is also the hold time of $\overline{AOE}$ after $\overline{ARE}$ rising.
19–16	RDSETUP	OF(*value*)	0–15	Read setup width. Number of clock† cycles of setup time for address (EA), chip enable ($\overline{CE}$), and byte enables ($\overline{BE}$[0-3]) before read strobe falls. For asynchronous read accesses, this is also the setup time of $\overline{AOE}$ before $\overline{ARE}$ falls.
15–14	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.

† Clock cycles are in terms of CLKOUT1 for C620x/C670x.

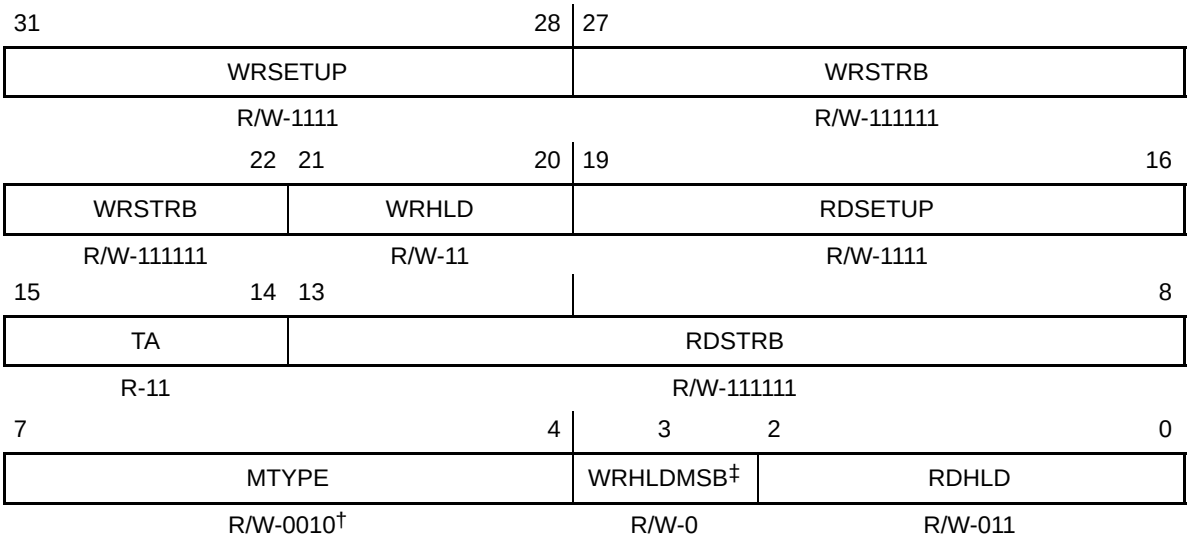
Table B–63. EMIF CE Space Control Register (CECTL) Field Values (EMIF_CECTL_field_symval)(Continued)

Bit	*field*	*symval*	Value	Description
13–8	RDSTRB	OF(*value*)	0–63	Read strobe width. The width of read strobe ($\overline{\text{ARE}}$) in clock† cycles
7	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
6–4	MTYPE			Memory type of the corresponding CE spaces.
		ASYNC8	000	8-bit-wide ROM (CE1 only)
		ASYNC16	001	16-bit-wide ROM (CE1 only)
		ASYNC32	010	32-bit-wide asynchronous interface
		SDRAM32	011	32-bit-wide SDRAM (CE0, CE2, CE3 only)
		SBSRAM32	100	32-bit-wide SBSRAM
			101–111	Reserved
3–2	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
1–0	RDHLD	OF(*value*)	0–3	Read hold width. Number of clock† cycles that address (EA) and byte strobes ($\overline{\text{BE}}$[0-3]) are held after read strobe rises. For asynchronous read accesses, this is also the hold time of $\overline{\text{AOE}}$ after $\overline{\text{ARE}}$ rising.

† Clock cycles are in terms of CLKOUT1 for C620x/C670x.

B.4.3 EMIF CE Space Control Register (CECTL) (C621x/C671x/C64x)

Figure B–59. EMIF CE Space Control Register (CECTL)



† MTYPE default value is R/W-0000.
‡ For C621x/C671x, this field is Reserved. R-0.

Legend: R/W-x = Read/Write-Reset value

Table B–64. EMIF CE Space Control Register (CECTL) Field Values (EMIF_CECTL_field_symval/EMIFA_CECTL_field_symval/ EMIFB_CECTL_field_symval)

Bit	*field*	*symval*	Value	Description
31–28	WRSETUP	OF(*value*)	0–15	Write setup width. Number of clock† cycles of setup time for address (EA), chip enable ($\overline{CE}$), and byte enables ($\overline{BE}$[0-3]) before write strobe falls. For asynchronous read accesses, this is also the setup time of $\overline{AOE}$ before $\overline{ARE}$ falls.
27–22	WRSTRB	OF(*value*)	0–63	Write strobe width. The width of write strobe ($\overline{AWE}$) in clock† cycles
21–20	WRHLD	OF(*value*)	0–3	Write hold width. Number of clock† cycles that address (EA) and byte strobes ($\overline{BE}$[0-3]) are held after write strobe rises. For asynchronous read accesses, this is also the hold time of $\overline{AOE}$ after $\overline{ARE}$ rising.

† Clock cycles are in terms of ECLKOUT for the C621x/C671x, and ECLKOUT1 for the C64x.
‡ 32-bit and 64-bit interfaces (MTYPE=0010b, 0011b, 0100b, 1100b, 1101b, 1110b) do not apply to C6712 and C64x EMIFB.

Table B–64. EMIF CE Space Control Register (CECTL) Field Values (EMIF_CECTL_field_symval/EMIFA_CECTL_field_symval/ EMIFB_CECTL_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
19–16	RDSETUP	OF(*value*)	0–15	Read setup width. Number of clock† cycles of setup time for address (EA), chip enable ($\overline{CE}$), and byte enables ($\overline{BE}$[0-3]) before read strobe falls. For asynchronous read accesses, this is also the setup time of $\overline{AOE}$ before $\overline{ARE}$ falls.
15–14	TA	OF(*value*)	0–3	Turn-around time. Turn-around time controls the number of ECLKOUT cycles between a read, and a write, or between reads, to different CE spaces (asynchronous memory types only).
13–8	RDSTRB	OF(*value*)	0–63	Read strobe width. The width of read strobe ($\overline{ARE}$) in clock† cycles
7–4	MTYPE			Memory type of the corresponding CE spaces
		ASYNC8	0000	Asynchronous interface
		ASYNC16	0001	16-bit-wide asynchronous interface
		ASYNC32	0010	32-bit-wide asynchronous interface
		SDRAM32	0011	32-bit-wide SDRAM
		SBSRAM32	0100	32-bit-wide SBSRAM (C621x/C671x) 32-bit-wide programmable synchronous memory (C64x)
			0101–0111	Reserved
		SDRAM8	1000	8-bit-wide SDRAM
		SDRAM16	1001	16-bit-wide SDRAM
		SBSRAM8	1010	8-bit-wide SBSRAM (C621x/C671x) 8-bit-wide programmable synchronous memory (C64x)
		SBSRAM16	1011	16-bit-wide SBSRAM (C621x/C671x) 16-bit-wide programmable synchronous memory (C64x)
			1100	64-bit-wide asynchronous interface (C64x only)
			1101	64-bit-wide SDRAM (C64x only)

† Clock cycles are in terms of ECLKOUT for the C621x/C671x, and ECLKOUT1 for the C64x.
‡ 32-bit and 64-bit interfaces (MTYPE=0010b, 0011b, 0100b, 1100b, 1101b, 1110b) do not apply to C6712 and C64x EMIFB.

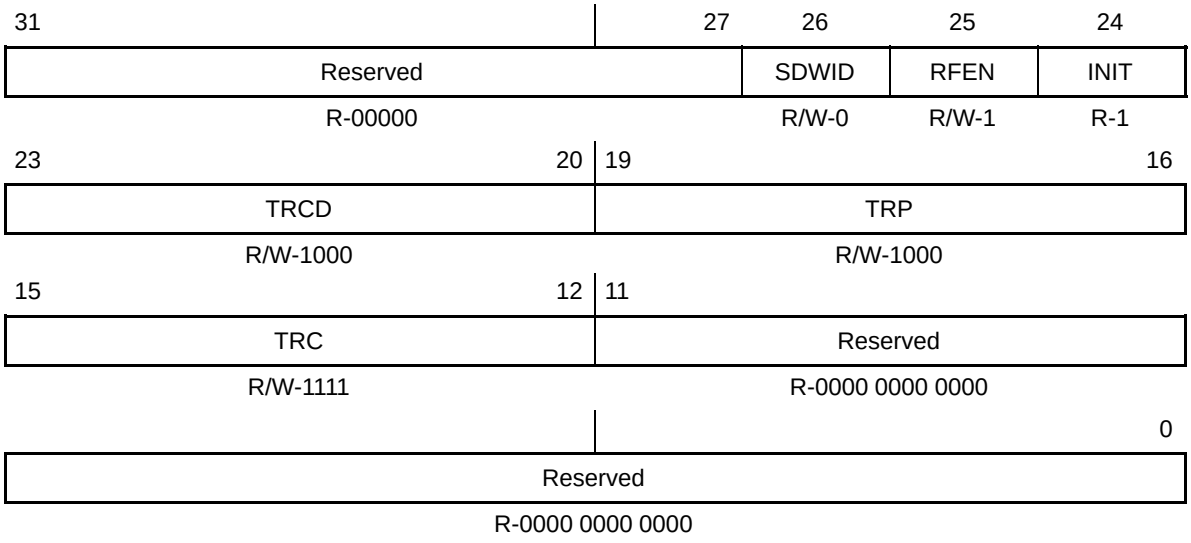
Table B–64. EMIF CE Space Control Register (CECTL) Field Values (EMIF_CECTL_field_symval/EMIFA_CECTL_field_symval/ EMIFB_CECTL_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
			1110	64-bit-wide programmable synchronous memory (C64x only)
			1111	Reserved
3	WRHLDMSB	OF(*value*)		
2–0	RDHLD	OF(*value*)	0–7	Read hold width. Number of clock† cycles that address (EA) and byte strobes (BE[0-3]) are held after read strobe rises. For asynchronous read accesses, this is also the hold time of AOE after ARE rising.

† Clock cycles are in terms of ECLKOUT for the C621x/C671x, and ECLKOUT1 for the C64x.
‡ 32-bit and 64-bit interfaces (MTYPE=0010b, 0011b, 0100b, 1100b, 1101b, 1110b) do not apply to C6712 and C64x EMIFB.

B.4.4 EMIF SDRAM Control Register (SDCTL) (C620x/C670x)

Figure B–60. EMIF SDRAM Control Register (SDCTL)



Legend: R/W-x = Read/Write-Reset value

Table B–65. EMIF SDRAM Control Register (SDCTL) Field Values (EMIF_SDCTL_field_symval)

Bit	*field*	*symval*	Value	Description
31–27	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
26	SDWID			SDRAM column width select
		4X8BIT	0	9 column address pins (512 elements per row)
		2X16BIT	1	8 column address pins (256 elements per row)
25	RFEN			Refresh enable
		DISABLE	0	SDRAM refresh disabled
		ENABLE	1	SDRAM refresh enabled
24	INIT			Forces initialization of all SDRAM present
		NO	0	No effect

† t_{cyc} refers to the EMIF clock period, which is equal to CLKOUT2 period for the C620x/C670x.

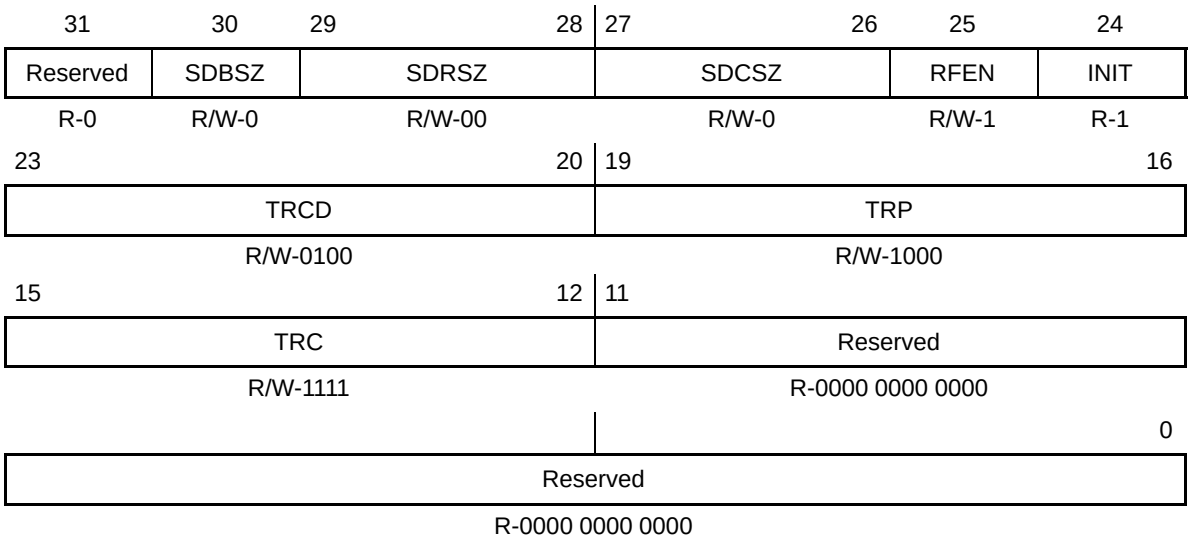
Table B–65. EMIF SDRAM Control Register (SDCTL) Field Values (EMIF_SDCTL_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
		YES	1	Initialize SDRAM in each CE space configured for SDRAM. EMIF automatically changes INIT back to 0 after SDRAM initialization is performed.
23–20	TRCD	OF(*value*)	0–15	Specifies the t_{RCD} value of the SDRAM in EMIF clock cycles† TRCD = t_{RCD} / t_{cyc} – 1
19–16	TRP	OF(*value*)	0–15	Specifies the t_{RC} value of the SDRAM in EMIF clock cycles† TRC = t_{RC} / t_{cyc} – 1
15–12	TRC	OF(*value*)	0–15	Specifies the t_{RC} value of the SDRAM in EMIF clock cycles† TRC = t_{RC} / t_{cyc} – 1
11–0	Reserved			Reserved

† t_{cyc} refers to the EMIF clock period, which is equal to CLKOUT2 period for the C620x/C670x.

B.4.5 EMIF SDRAM Control Register (SDCTL) (C621x/C671x)

Figure B–61. EMIF SDRAM Control Register (SDCTL)



Legend: R/W-x = Read/Write-Reset value

Table B–66. EMIF SDRAM Control Register (SDCTL) Field Values (EMIF_SDCTL_field_symval)

Bit	*field*	*symval*	Value	Description
31	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
30	SDBSZ			SDRAM bank size
		2BANKS	0	One bank-select pin (two banks)
		4BANKS	1	Two bank-select pins (four banks)
29–28	SDRSZ			SDRAM row size
		11ROW	00	11 row address pins (2048 rows per bank)
		12ROW	01	12 row address pins (4096 rows per bank)
		13ROW	10	13 row address pins (8192 rows per bank)
			11	Reserved

† t_{cyc} refers to the EMIF clock period, which is equal to ECLKOUT period for the C621x/C671x.

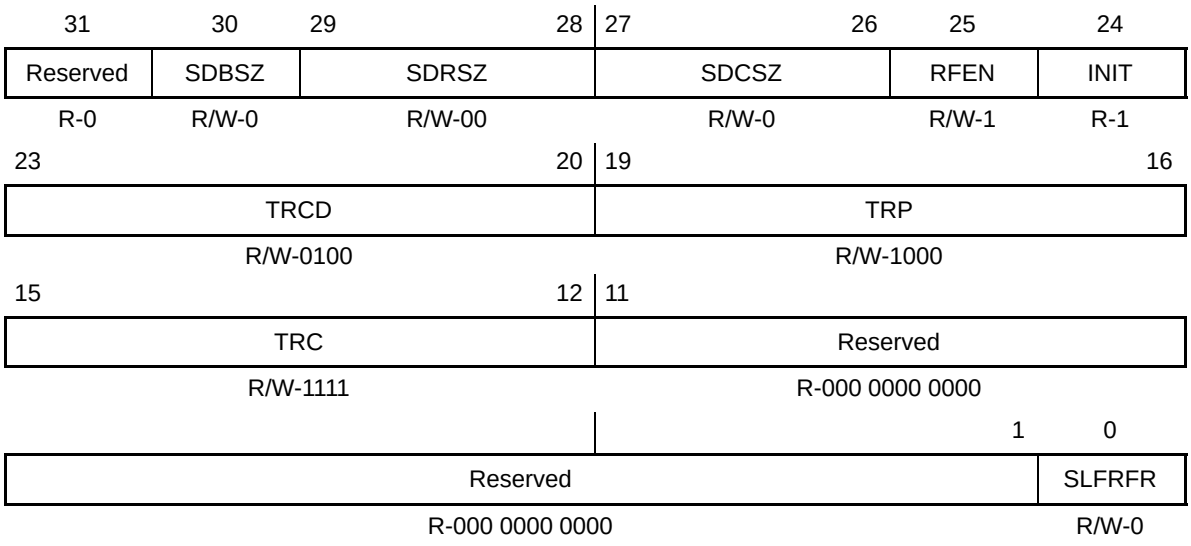
Table B–66. EMIF SDRAM Control Register (SDCTL) Field Values (EMIF_SDCTL_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
27–26	SDCSZ			SDRAM column size
		9COL	00	9 column address pins (512 elements per row)
		8COL	01	8 column address pins (256 elements per row)
		10COL	10	10 column address pins (1024 elements per row)
			11	Reserved
25	RFEN			Refresh enable
		DISABLE	0	SDRAM refresh disabled
		ENABLE	1	SDRAM refresh enabled
24	INIT			Forces initialization of all SDRAM present
		NO	0	No effect
		YES	1	Initialize SDRAM in each CE space configured for SDRAM. EMIF automatically changes INIT back to 0 after SDRAM initialization is performed.
23–20	TRCD	OF(*value*)	0–15	Specifies the t_{RCD} value of the SDRAM in EMIF clock cycles† TRCD = t_{RCD} / t_{cyc} – 1
19–16	TRP	OF(*value*)	0–15	Specifies the t_{RC} value of the SDRAM in EMIF clock cycles† TRC = t_{RC} / t_{cyc} – 1
15–12	TRC	OF(*value*)	0–15	Specifies the t_{RC} value of the SDRAM in EMIF clock cycles† TRC = t_{RC} / t_{cyc} – 1
11–0	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.

† t_{cyc} refers to the EMIF clock period, which is equal to ECLKOUT period for the C621x/C671x.

B.4.6 EMIF SDRAM Control Register (SDCTL) (C64x, EMIFA/EMIFB only)

Figure B–62. EMIF SDRAM Control Register (SDCTL)



Legend: R/W-x = Read/Write-Reset value

Table B–67. EMIF SDRAM Control Register (SDCTL) Field Values (EMIFA_SDCTL_field_symval)

Bit	*field*	*symval*	Value	Description
31	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
30	SDBSZ			SDRAM bank size
		2BANKS	0	One bank-select pin (two banks)
		4BANKS	1	Two bank-select pins (four banks)

† On the C64x, TRCD specifies the number of ECLKOUT1 cycles between an ACTV command and a READ or WRT command (CAS). The specified separation is maintained while driving write data one cycle earlier.
‡ t_{cyc} refers to the EMIF clock period, which is equal to or ECLKOUT1 period for the C64x.
§ SLFRFR only applies to EMIFA. Bit 0 is Reserved, RW+0, on EMIFB.

Table B–67. EMIF SDRAM Control Register (SDCTL) Field Values (EMIFA_SDCTL_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
29–28	SDRSZ			SDRAM row size
		11ROW	00	11 row address pins (2048 rows per bank)
		12ROW	01	12 row address pins (4096 rows per bank)
		13ROW	10	13 row address pins (8192 rows per bank)
			11	Reserved
27–26	SDCSZ			SDRAM column size
		9COL	00	9 column address pins (512 elements per row)
		8COL	01	8 column address pins (256 elements per row)
		10COL	10	10 column address pins (1024 elements per row)
			11	Reserved
25	RFEN			Refresh enable
		DISABLE	0	SDRAM refresh disabled
		ENABLE	1	SDRAM refresh enabled
24	INIT			Forces initialization of all SDRAM present
		NO	0	No effect
		YES	1	Initialize SDRAM in each CE space configured for SDRAM. EMIF automatically changes INIT back to 0 after SDRAM initialization is performed.
23–20	TRCD†	OF(*value*)	0–15	Specifies the t_{RCD} value of the SDRAM in EMIF clock cycles‡ TRCD = t_{RCD} / t_{cyc} – 1
19–16	TRP	OF(*value*)	0–15	Specifies the t_{RP} value of the SDRAM in EMIF clock cycles‡ TRP = t_{RP} / t_{cyc} – 1
15–12	TRC	OF(*value*)	0–15	Specifies the t_{RC} value of the SDRAM in EMIF clock cycles‡ TRC = t_{RC} / t_{cyc} – 1

† On the C64x, TRCD specifies the number of ECLKOUT1 cycles between an ACTV command and a READ or WRT command (CAS). The specified separation is maintained while driving write data one cycle earlier.
‡ t_{cyc} refers to the EMIF clock period, which is equal to or ECLKOUT1 period for the C64x.
§ SLFRFR only applies to EMIFA. Bit 0 is Reserved, RW+0, on EMIFB.

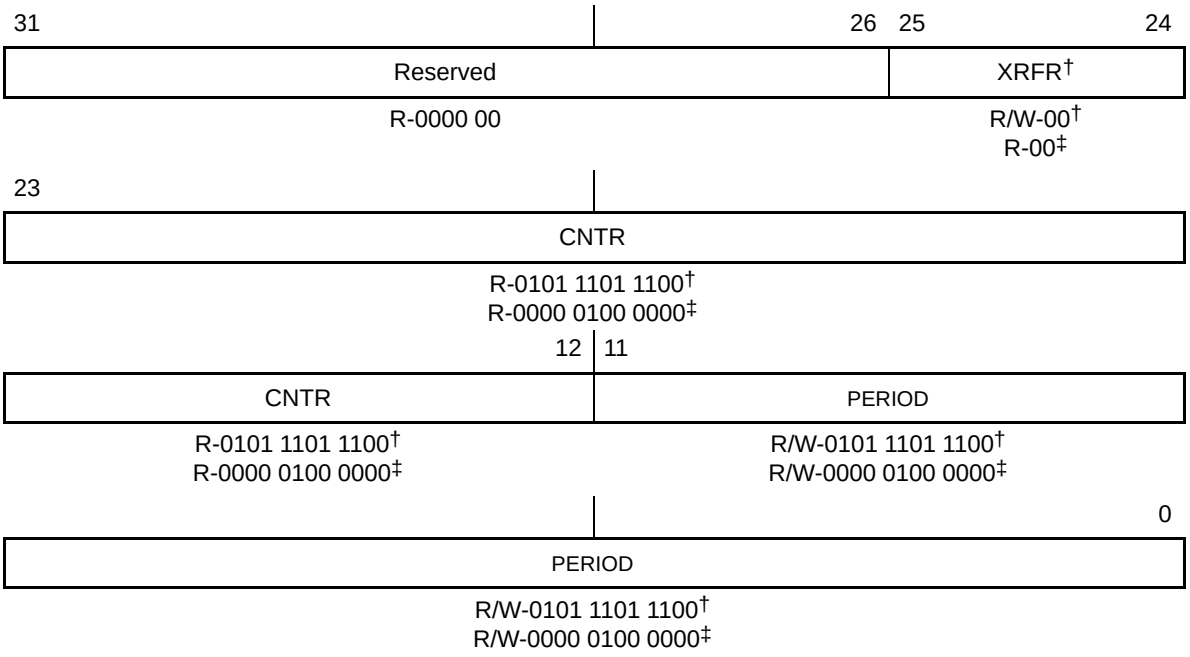
Table B–67. EMIF SDRAM Control Register (SDCTL) Field Values (EMIFA_SDCTL_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
11–1	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
0	SLFRFR §			Self-refresh mode, if SDRAM is used in the system:
			0	Self-refresh mode disabled
			1	Self-refresh mode enabled
				If SDRAM is not used:
			0	General-purpose output SDCKE = 1
			1	General-purpose output SDCKE = 0

† On the C64x, TRCD specifies the number of ECLKOUT1 cycles between an ACTV command and a READ or WRT command (CAS). The specified separation is maintained while driving write data one cycle earlier.
‡ t_{cyc} refers to the EMIF clock period, which is equal to or ECLKOUT1 period for the C64x.
§ SLFRFR only applies to EMIFA. Bit 0 is Reserved, RW+0, on EMIFB.

B.4.7 EMIF SDRAM Timing Register (SDTIM)

Figure B–63. EMIF SDRAM Timing Register (SDTIM)



† Applies to C621x/C671x/C64x
‡ Applies to C620x/C670x

Legend: R/W-x = Read/Write-Reset value

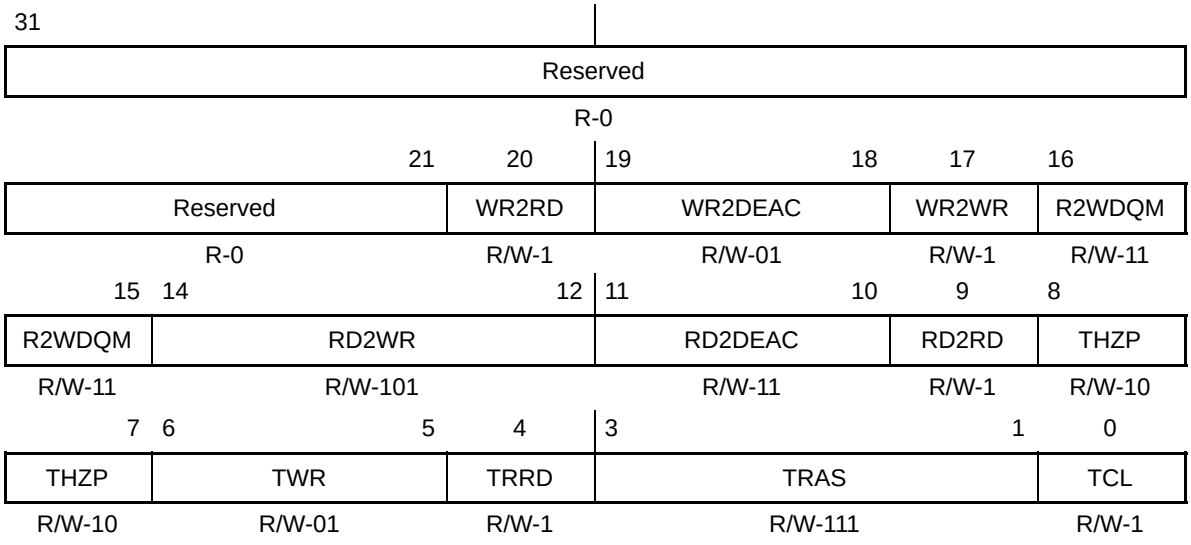
Table B–68. EMIF SDRAM Timing Register (SDTIM) Field Values (EMIF_SDTIM_field_symval)

Bit	*field*	*symval*	Value	Description
31–26	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
25–24	XRFR†	OF(*value*)	0–3	Extra refreshes controls the number of refreshes performed to SDRAM when the refresh counter expires.
23–12	CNTR	OF(*value)*	0–FFFh	Current value of the refresh counter
11–0	PERIOD	OF(*value)*	0–FFFh	Refresh period in EMIF clock cycles‡

† Applies to C621x/C671x/C64x only.
‡ For C620x/C670x, EMIF clock cycles = CLKOUT2 cycles.
For C621x/C671x, EMIF clock cycles = ECLKOUT cycles.
For C64x, EMIF clock cycles = ECLKOUT1 cycles.

B.4.8 EMIF SDRAM Extension Register (SDEXT) (C621x/C671x/C64x)

Figure B–64. EMIF SDRAM Extension Register (SDEXT)



Legend: R/W-x = Read/Write-Reset value

Table B–69. EMIF SDRAM Extension Register (SDEXT) Field Values (EMIF_SDEXT_field_symval)

Bit	*field*	*symval*	Value	Description
31–21	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
20	WR2RD	OF(*value*)		Specifies minimum number of cycles between WRITE to READ command of the SDRAM in ECLKOUT† cycles WR2RD = (# of cycles WRITE to READ) – 1
19–18	WR2DEAC	OF(*value*)	0–3	Specifies minimum number of cycles between WRITE to DEAC/DCAB command of the SDRAM in ECLKOUT† cycles WR2DEAC = (# of cycles WRITE to DEAC/DCAB) – 1
17	WR2WR	OF(*value*)		Specifies minimum number of cycles between WRITE to WRITE command of the SDRAM in ECLKOUT† cycles WR2WR = (# of cycles WRITE to WRITE) – 1
16–15	R2WDQM	OF(*value*)	0–3	Specifies number of of cycles that BEx signals must be high preceding a WRITE interrupting a READ R2WDQM = (# of cycles BEx high) – 1

† For C64x, ECLKOUT referenced in this table is equivalent to ECLKOUT1.
‡ t_{cyc} refers to the EMIF clock period, which is equal to ECLKOUT period for the C621x/C671x, ECLKOUT1 period for C64x.

Table B–69. EMIF SDRAM Extension Register (SDEXT) Field Values (EMIF_SDEXT_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
14–12	RD2WR	OF(*value*)	0–7	Specifies number of cycles between READ to WRITE command of the SDRAM in ECLKOUT† cycles RD2WR = (# of cycles READ to WRITE) – 1
11–10	RD2DEAC	OF(*value*)	0–3	Specifies number of cycles between READ to DEAC/DCAB of the SDRAM in ECLKOUT† cycles RD2DEAC = (# of cycles READ to DEAC/DCAB) – 1
9	RD2RD	OF(*value*)		Specifies number of cycles between READ to READ command (same CE space) of the SDRAM in ECLKOUT† cycles
			0	READ to READ = 1 ECLKOUT† cycle
			1	READ to READ = 2 ECLKOUT† cycle
8–7	THZP	OF(*value*)	0–3	Specifies t_{HZP} (also known as t_{ROH}) value of the SDRAM in ECLKOUT cycles THZP = t_{HZP} / t_{cyc}‡ – 1
6–5	TWR	OF(*value*)	0–3	Specifies t_{WR} value of the SDRAM in ECLKOUT† cycles TWR = t_{WR} / t_{cyc}‡ – 1
4	TRRD	OF(*value*)		Specifies t_{RRD} value of the SDRAM in ECLKOUT† cycles
			0	T_{RRD} = 2 ECLKOUT† cycles
			1	T_{RRD} = 3 ECLKOUT† cycles
3–1	TRAS	OF(*value*)	0–7	Specifies t_{RAS} value of the SDRAM in ECLKOUT† cycles TRAS = t_{RAS} / t_{cyc}‡ – 1
0	TCL	OF(*value*)		Specified CAS latency of the SDRAM in ECLKOUT† cycles
			0	CAS latency = 2 ECLKOUT† cycles
			1	CAS latency = 3 ECLKOUT† cycles

† For C64x, ECLKOUT referenced in this table is equivalent to ECLKOUT1.
‡ t_{cyc} refers to the EMIF clock period, which is equal to ECLKOUT period for the C621x/C671x, ECLKOUT1 period for C64x.

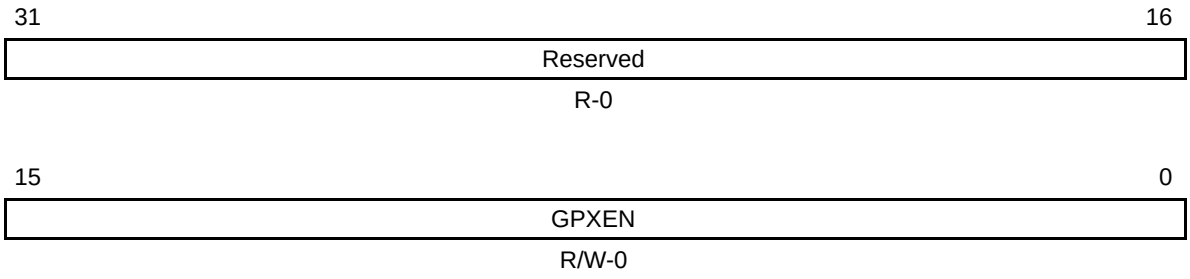
B.5 General-Purpose Input/Output (GPIO) Registers

Table B–70. GPIO Register

Acronym	Register Name	Section
GPEN	GPIO enabling register	B.5.1
GPDIR	GPIO direction register	B.5.2
GPVAL	GPIO value register	B.5.3
GPDH	GPIO delta high register	B.5.4
GPHM	GPIO high mask register	B.5.5
GPDL	GPIO delta low register	B.5.6
GPLM	GPIO low mask register	B.5.7
GPGC	GPIO global control register	B.5.8
GPPOL	GPIO interrupt polarity register	B.5.9

B.5.1 GPIO Enable Register (GPEN)

Figure B–65. GPIO Enable Register (GPEN)



Legend: R/W = Read/Write; -*n* = value after reset

Table B–71. GPIO Enable Register (GPEN) Bit Field Description

Bit	Field	*symval*†	Value	Description
31–16	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
15–0	GPXEN	OF(*value*)		GPIO Mode enable

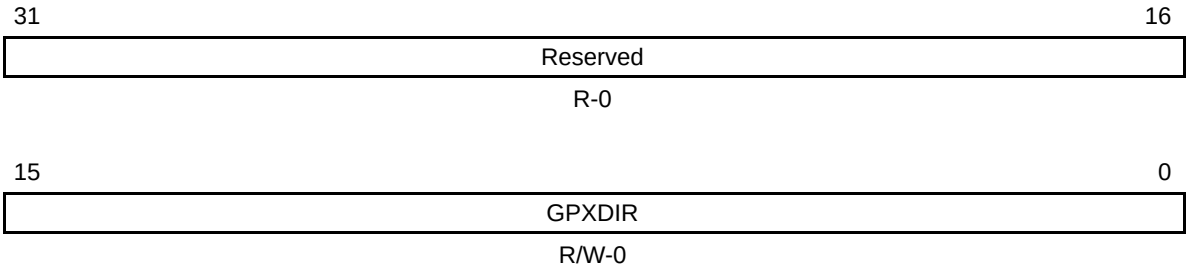
Table B–71. GPIO Enable Register (GPEN) Bit Field Description (Continued)

Bit	Field	*symval*†	Value	Description
			0	GPx pin is disabled as general-purpose input/output pin. It does not function as a GPIO pin and defaults to high impedance state.
			1	GPx pin is enabled as general-purpose input/output pin. It defaults to high impedance state.

† For CSL implementation, use the notation GPIO_GPEN_GPXEN_*symval*

B.5.2 GPIO Direction Register (GPDIR)

Figure B–66. GPIO Direction Register (GPDIR)



Legend: R/W = Read/Write; -*n* = value after reset

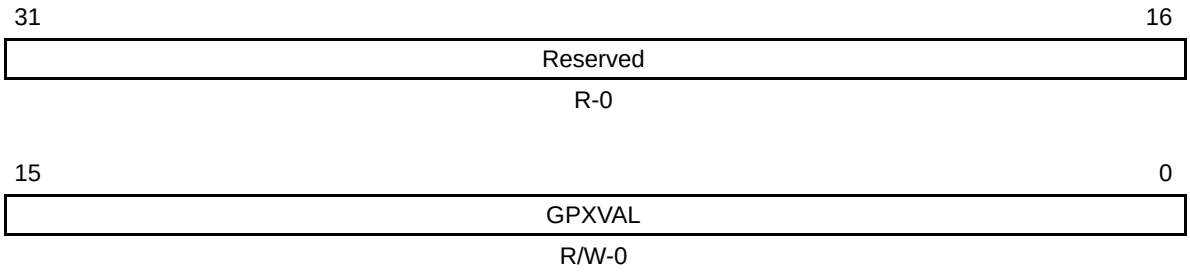
Table B–72. GPIO Direction Register (GPDIR) Bit Field Description

Bit	Field	*symval*†	Value	Description
31–16	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
15–0	GPXDIR	OF(*value*)		GPx Direction. Controls direction (input or output) of GPIO pin. Applies when the corresponding GPxEN bit in the GPEN register is set to 1.
			0	GPx pin is an input.
			1	GPx pin is an output.

† For CSL implementation, use the notation GPIO_GPDIR_GPXDIR_*symval*

B.5.3 GPIO Value Register (GPVAL)

Figure B–67. GPIO Value Register (GPVAL)



Legend: R/W = Read/Write; -*n* = value after reset

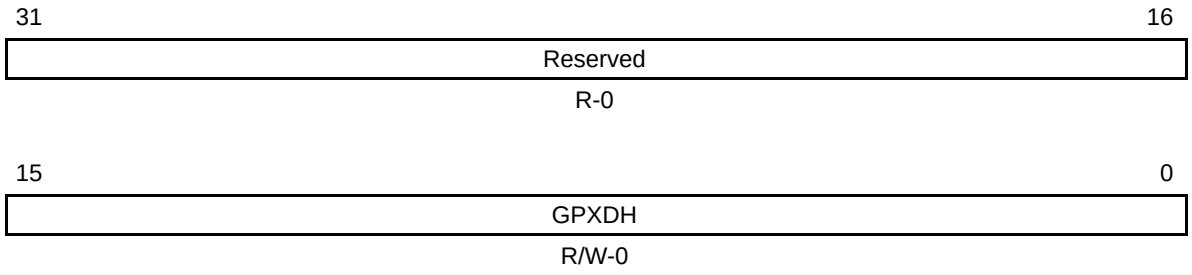
Table B–73. GPIO Value Register (GPVAL) Bit Field Description

Bit	Field	*symval*†	Value	Description
31–16	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
15–0	GPXVAL	OF(*value*)		Value detected at GPx input/output. Applies when the corresponding GPXEN bit in the GPEN register is set to 1.
				When GPx pin is an input.
			0	A value of 0 is latched from the GPx input pin.
			1	A value of 1 is latched from the GPx input pin.
				When GPx pin is an output.
			0	GPx signal is driven low.
			1	GPx signal is driven high.

† For CSL implementation, use the notation GPIO_GPVAL_GPXVAL_*symval*

B.5.4 GPIO Delta High Register (GPDH)

Figure B–68. GPIO Delta High Register (GPDH)



Legend: R/W = Read/Write; *-n* = value after reset

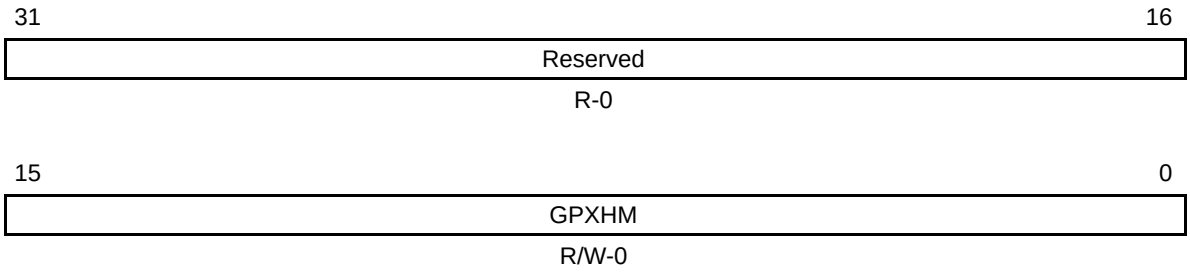
Table B–74. GPIO Delta High Register (GPDH) Bit Field Description

Bit	Field	*symval*†	Value	Description
31–16	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
15–0	GPXDH	OF(*value*)		GPx Delta High. A low-to-high transition is detected on the GPx input. Applies when the corresponding GPx pin is enabled as an input (GPXEN = 1 and GPXDIR = 0)
			0	A low-to-high transition is not detected on GPx
			1	A low-to-high transition is detected on GPx

† For CSL implementation, use the notation GPIO_GPDH_GPXDH_*symval*

B.5.5 GPIO High Mask Register (GPHM)

Figure B–69. GPIO High Mask Register (GPHM)



Legend: R/W = Read/Write; -*n* = value after reset

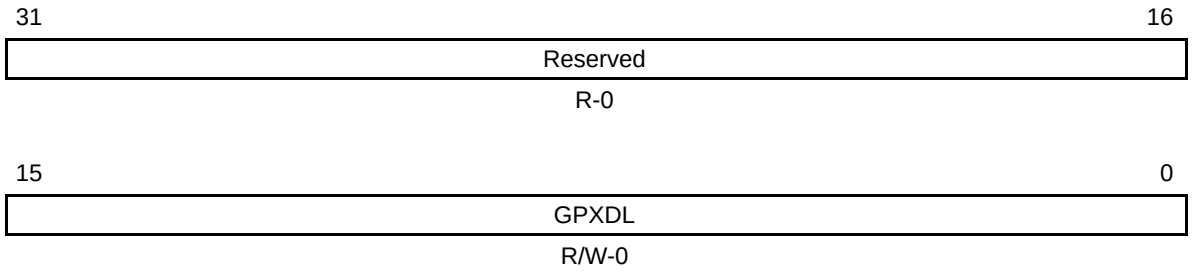
Table B–75. GPIO High Mask Register (GPHM) Bit Field Description

Bit	Field	*symval*†	Value	Description
31–16	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
15–0	GPXHM	OF(*value*)		GPx high mask. Enable interrupt/event generation based on either the corresponding GPxDH or GPxVAL bit in the GPDH and GPVAL registers, respectively. Applies when the corresponding GPxEN bit is enabled as an input (GPXEN = 1 and GPXDIR = 0)
			0	Interrupt/event generation disabled for GPx. The value or transition on GPx does not cause an interrupt/event generation.
			1	Interrupt/event generation enabled for GPx.

† For CSL implementation, use the notation GPIO_GPHM_GPXHM_*symval*

B.5.6 GPIO Delta Low Register (GPDL)

Figure B–70. GPIO Delta Low Register (GPDL)



Legend: R/W = Read/Write; *-n* = value after reset

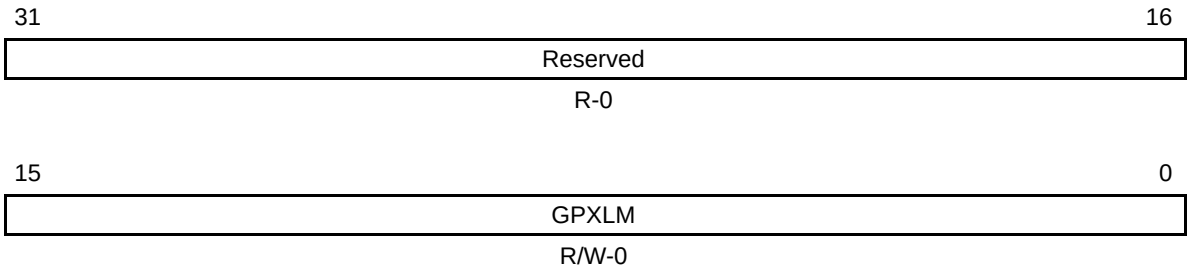
Table B–76. GPIO Delta Low Register (GPDL) Bit Field Description

Bit	Field	*symval*†	Value	Description
31–16	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
15–0	GPXDL	OF(*value*)		GPx Delta Low. A high-to-low transition is detected on the GPx input. Applies when the corresponding GPx pin is enabled as an input (GPXEN = 1 and GPxDIR = 0).
			0	A high-to-low transition is not detected on GPx.
			1	A high-to-low transition is detected on GPx.

† For CSL implementation, use the notation GPIO_GPDL_GPXDL_*symval*

B.5.7 GPIO Low Mask Register (GPLM)

Figure B–71. GPIO Low Mask Register (GPLM)



Legend: R/W = Read/Write; -*n* = value after reset

Table B–77. GPIO Low Mask Register (GPLM) Bit Field Description

Bit	Field	*symval*†	Value	Description
31–16	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
15–0	GPXLM	OF(*value*)		GPx low mask. Enable interrupt/event generation based on either the corresponding GPXDL or *inverted* GPXVAL bit in the GPDL and GPVAL registers, respectively. Applies when the corresponding GPxEN bit is enabled as an input (GPXEN = 1 and GPXDIR = 0)
			0	Interrupt/event generation disabled for GPx. The value or transition on GPx does not cause an interrupt/event generation.
			1	Interrupt/event generation enabled for GPx.

† For CSL implementation, use the notation GPIO_GPLM_GPXLM_*symval*

B.5.8 GPIO Global Control Register (GPGC)

Figure B–72. GPIO Global Control Register (GPGC)

31	8
Reserved	
R-0	

7	6	5	4	3	2	1	0
Reserved		GP0M	GPINT0M	Reserved	GPINTPOL	LOGIC	GPINTDV
R-0		R/W-0	R/W-0	R-0	R/W-0	R/W-0	R/W-0

Legend: R/W = Read/Write; -*n* = value after reset

Table B–78. GPIO Global Control Register (GPGC) Bit Field Description

Bit	***field***†	***symval***†	Value	Description
31–6	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
5	GP0M			GP0 Output Mode. Applies only if GP0 is configured as an output (GP0DIR = 1 in the GPDIR register).
		GPIOMODE	0	GPIO Mode—GP0 output is based on GP0 value (GP0VAL in GPVAL register)
		LOGICMODE	1	Logic Mode—GP0 output is based on the value of internal Logic Mode interrupt/event signal GPINT.
4	GPINT0M			GPINT0 interrupt/event generation mode.
		PASSMODE	0	Pass Through Mode—GPINT0 interrupt/event generation is based on GP0 input value (GP0VAL in the GPVAL register).
		LOGICMODE	1	Logic Mode—GPINT0 interrupt/event generation is based on GPINT.
3	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
2	GPINTPOL			GPINT Polarity. Applies to Logic Mode (GPINT0M = 1) only.
		LOGICTRUE	0	GPINT is active (high) when the logic combination of the GPIO inputs is evaluated true.
		LOGICFALSE	1	GPINT is active (high) when the logic combination of the GPIO inputs is evaluated false.

† For CSL implementation, use the notation GPIO_GPGC_*field_symval*

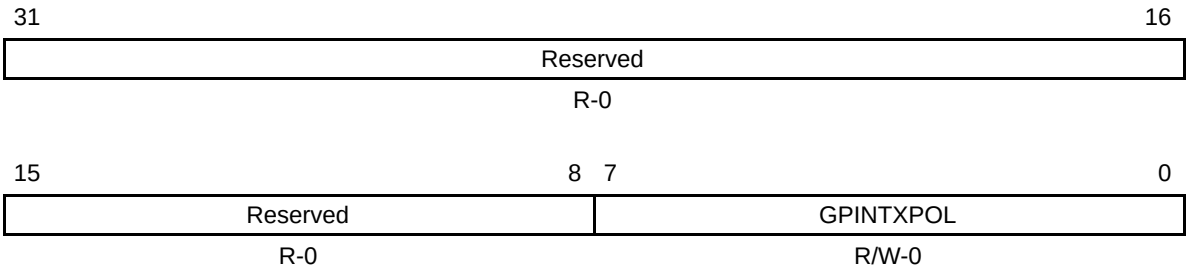
Table B–78. GPIO Global Control Register (GPGC) Bit Field Description (Continued)

Bit	*field*†	*symval*†	Value	Description
1	LOGIC			GPINT Logic. Applies to Logic Mode (GPINT0M = 1) only.
		ORMODE	0	OR Mode—GPINT is generated based on the logical-OR of all GPx events enabled in the GPHM or GPLM registers.
		ANDMODE	1	AND Mode—GPINT is generated based on the logical-AND of all GPx events enabled in the GPHM or GPLM registers.
0	GPINTDV			GPINT Delta/Value Mode. Applies to Logic Mode (GPINT0M = 1) only.
		DELTAMODE	0	Delta Mode—GPINT is generated based on a logic combination of *transitions* on the GPx pins. The corresponding bits in the GPHM and/or GPLM registers must be set.
		VALUEMODE	1	Value Mode—GPINT is generated based on a logic combination of *values* on the GPx pins. The corresponding bits in the GPHM and/or GPLM registers must be set.

† For CSL implementation, use the notation GPIO_GPGC_*field_symval*

B.5.9 GPIO Interrupt Polarity Register (GPPOL)

Figure B–73. GPIO Interrupt Polarity Register (GPPOL)



Legend: R/W = Read/Write; -*n* = value after reset

Table B–79. GPIO Interupt Polarity Register (GPPOL) Bit Field Description

Bit	Field	*symval*†	Value	Description
31–8	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
7–0	GPINTXPOL	OF(*value*)		GPINTx polarity bit. Applies to pass-through mode only (GPINT0M = 0 in GPGC).
			0	GPINTx is asserted (high) based on a rising edge of GPx (effectively based on the value of the corresponding GPXVAL)
			1	GPINTx is asserted (high) based on a falling edge of GPx (effectively based on the inverted value of the corresponding GPXVAL)

† For CSL implementation, use the notation GPIO_GPPOL_GPINTXPOL_*symval*

B.6 Host Port Interface (HPI) Register

Table B–80. HPI Registers for C62x/C67x DSP

		Read/Write Access		
Acronym	**Register Name**	**Host**	**CPU**	**Section**
HPID	HPI data register	R/W	–	B.6.1
HPIA	HPI address register	R/W	–	B.6.2
HPIC	HPI control register	R/W	R/W	B.6.3

Table B–81. HPI Registers for C64x DSP

		Read/Write Access		
Acronym	**Register Name**	**Host**	**CPU**	**Section**
HPID	HPI data register	R/W	–	B.6.1
HPIAW†	HPI address write register	R/W	R/W	B.6.2
HPIAR†	HPI address read register	R/W	R/W	B.6.2
HPIC	HPI control register	R/W	R/W	B.6.3
TRCTL	HPI transfer request control register	–	R/W	B.6.4

† Host access to the HPIA updates both HPIAW and HPIAR. The CPU can access HPIAW and HPIAR, independently.

B.6.1 HPI Data Register (HPID)

The HPI data register (HPID) contains the data that was read from the memory accessed by the HPI, if the current access is a read; HPID contains the data that is written to the memory, if the current access is a write.

B.6.2 HPI Address Register (HPIA)

The HPI address register (HPIA) contains the address of the memory accessed by the HPI at which the current access occurs. This address is a 32-bit word address with all 32-bits readable/writable. The two LSBs always function as 0, regardless of the value read from their location. The C62x/C67x HPIA is only accessible by the host, it is not mapped to the DSP memory.

The C64x HPIA is separated into two registers internally: the HPI address write register (HPIAW) and the HPI address read register (HPIAR). The HPIA is accessible by both the host and the CPU. By separating the HPIA into HPIAW and HPIAR internally, the CPU can update the read and write memory address independently to allow the host to perform read and write to different address ranges. When reading HPIA from the CPU, the value returned corresponds to the address currently being used by the HPI and DMA to transfer data inside the DSP. It is not the address for the current transfer at the external pins. Thus, reading HPIA does not indicate the status of a transfer, and should not be relied upon to do so.

For the C64x HPI, a host access to HPIA is identical to the operation of the C62x/C67x HPI. The HCNTL[1–0] control bits are set to 01b to indicate an access to HPIA. A host write to HPIA updates both HPIAW and HPIAR internally. A host read of HPIA returns the value in the most-recently-used HPIAx register. For example, if the most recent HPID access was a read, then an HPIA read by the external host returns the value in HPIAR; if the most recent HPID access was a write, then an HPIA read by the external host returns the value in HPIAW.

Systems that update HPIAR/HPIAW internally via the CPU must not allow HPIA updates via the external bus and conversely. The HPIAR/HPIAW registers can be read independently by both the CPU and the external host. The system must not allow HPID accesses via the external host while the DSP is updating the HPIAR/W registers internally. This can be controlled by any convenient means, including the use of general-purpose input/output pins to perform handshaking between the host and the DSP.

B.6.3 HPI Control Register (HPIC)

The HPI control register (HPIC) is normally the first register accessed to set configuration bits and initialize the interface. The HPIC is shown in Figure B–74, Figure B–75, Figure B–76, and Figure B–77 and described in Table B–82. From the host's view (Figure B–74 and Figure B–75), HPIC is organized as a 32-bit register with two identical halves, meaning the high halfword and low halfword contents are the same. On a host write, both halfwords must be identical, except when writing the DSPINT bits in HPI16 mode. In HPI16 mode when setting DSPINT = 1, the host must only write 1 to the lower 16-bit halfword or upper 16-bit halfword, but not both. In HPI32 mode, the upper and lower halfwords must always be identical.

From the C6000 CPU view (Figure B–76 and Figure B–77), HPIC is a 32-bit register with only 16-bits of useful data. Only CPU writes to the lower halfword affect HPIC values and HPI operation.

On C64x DSP, the HWOB bit is writable by the CPU. Therefore, care must be taken when writing to HPIC in order not to write an undesired value to HWOB.

Figure B–74. HPI Control Register (HPIC)—Host Reference View (C62x/C67x DSP)

31	21	20	19	18	17	16
Reserved		FETCH	HRDY	HINT	DSPINT	HWOB
HR-0		HR-0	HR-1	HR/W-0	HR/W-0	HR-0

15	5	4	3	2	1	0
Reserved		FETCH	HRDY	HINT	DSPINT	HWOB
HR-0		HR-0	HR-1	HR/W-0	HR/W-0	HR-0

Legend: H = Host access; R = Read only; R/W = Read/Write; -*n* = value after reset

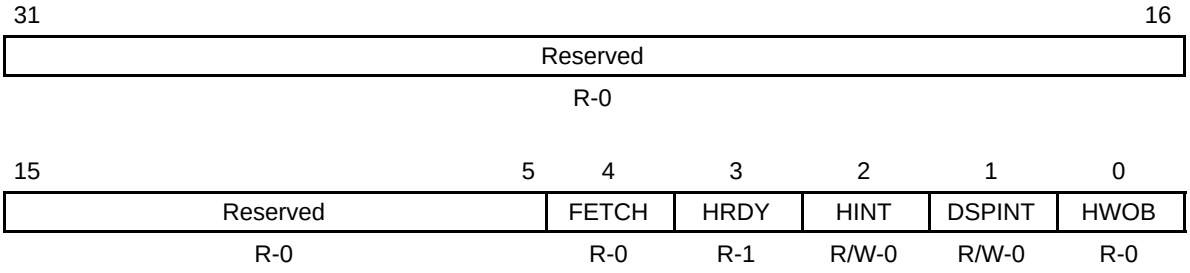
Figure B–75. HPI Control Register (HPIC)—Host Reference View (C64x DSP)

31	30	24	23	22	21	20	19	18	17	16
Reserved†	Reserved		Reserved†	Reserved		FETCH	HRDY	HINT	DSPINT	HWOB
HR/W-0	HR-0		HR/W-0	HR-0		HR-0	HR-1	HR/W-0	HR/W-0	HR/W-0

15	14	8	7	6	5	4	3	2	1	0
Reserved†	Reserved		Reserved†	Reserved		FETCH	HRDY	HINT	DSPINT	HWOB
HR/W-0	HR-0		HR/W-0	HR-0		HR-0	HR-1	HR/W-0	HR/W-0	HR/W-0

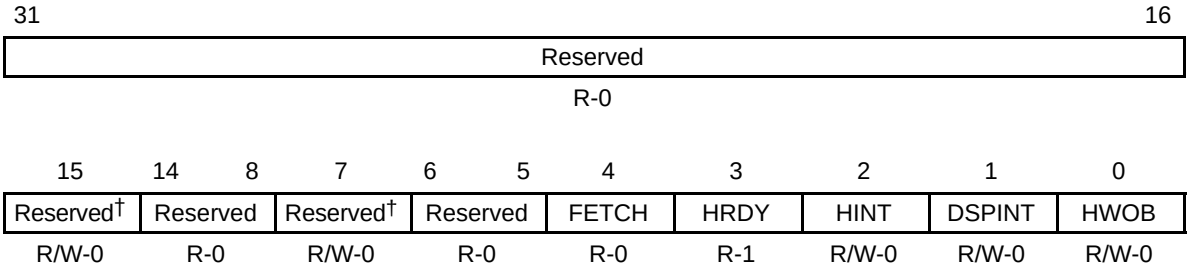
Legend: H = Host access; R = Read only; R/W = Read/Write; -*n* = value after reset
† These bits are writable fields and must be written with 0; otherwise, operation is undefined.

Figure B–76. HPI Control Register (HPIC)—CPU Reference View (C62x/C67x DSP)



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset

Figure B–77. HPI Control Register (HPIC)—CPU Reference View (C64x DSP)



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† These bits are writable fields and must be written with 0; otherwise, operation is undefined.

Table B–82. HPI Control Register (HPIC) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31–21	Reserved	–	0	Reserved. The reserved bit location is always read as 0.
20, 4	FETCH			Host fetch request bit.
		0	0	The value read by the host or CPU is always 0.
		1	1	The host writes a 1 to this bit to request a fetch into HPID of the word at the address pointed to by HPIA. The 1 is never actually written to this bit, however.
19, 3	HRDY			Ready signal to host bit. Not masked by $\overline{\text{HCS}}$ (as the $\overline{\text{HRDY}}$ pin is).
		0	0	The internal bus is waiting for an HPI data access request to finish.
		1	1	
18, 2	HINT			DSP-to-host interrupt bit. The inverted value of this bit determines the state of the CPU $\overline{\text{HINT}}$ output.
		0	0	CPU $\overline{\text{HINT}}$ output is logic 1.
		1	1	CPU $\overline{\text{HINT}}$ output is logic 0.
17, 1	DSPINT			The host processor-to-CPU/DMA interrupt bit.
		0	0	
		1	1	
16, 0	HWOB			Halfword ordering bit affects both data and address transfers. Only the host can modify this bit. HWOB must be initialized before the first data or address register access. For HPI32, HWOB is not used and the value of HWOB is irrelevant.
		0	0	The first halfword is most significant.
		1	1	The first halfword is least significant.
15–5	Reserved	–	0	Reserved. The reserved bit location is always read as 0.

† For CSL implementation, use the notation HPI_HPIC_*field_symval*

B.6.4 HPI Transfer Request Control Register (TRCTL) (C64x DSP only)

The HPI transfer request control register (TRCTL) controls how the HPI submits its requests to the EDMA subsystem. The TRCTL is shown in Figure B–179 and described in Table B–185.

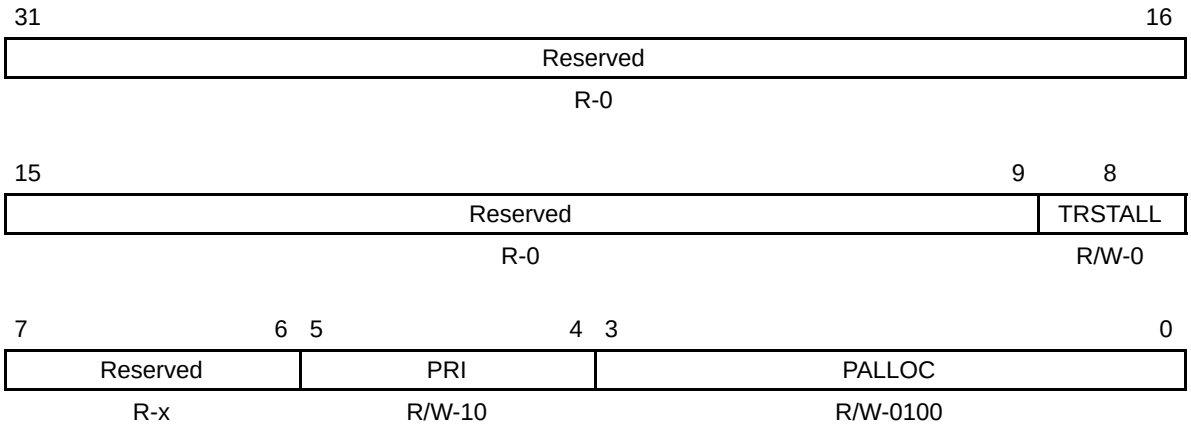
To safely change the PALLOC or PRI bits in TRCTL, the TRSTALL bit needs to be used to ensure a proper transition. The following procedure must be followed to change the PALLOC or PRI bits:

1) Set the TRSTALL bit to 1 to stop the HPI from submitting TR requests on the current PRI level. In the same write, the desired new PALLOC and PRI fields may be specified.
2) Clear all EDMA event enables (EER) corresponding to both old and new PRI levels to stop EDMA from submitting TR requests on both PRI levels. Do not manually submit additional events via the EDMA.
3) Do not submit new QDMA requests on either old or new PRI level.
4) Stop L2 cache misses on either old or new PRI level. This can be done by forcing program execution or data accesses in internal memory. Another way is to have the CPU executing a tight loop that does not cause additional cache misses.
5) Poll the appropriate PQ bits in the priority queue status register (PQSR) of the EDMA until both queues are empty (see the *Enhanced DMA (EDMA) Controller Reference Guide*, SPRU234).
6) Clear the TRSTALL bit to 0 to allow the HPI to continue normal operation.

Requestors are halted on the old HPI PRI level so that memory ordering can be preserved. In this case, all pending requests corresponding to the old PRI level must be let to complete before HPI is released from stall state.

Requestors are halted on the new PRI level to ensure that at no time can the sum of all requestor allocations exceed the queue length. By halting all requestors at a given level, you can be free to modify the queue allocation counters of each requestor.

Figure B–78. HPI Transfer Request Control Register (TRCTL)



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset; -x = value is indeterminate after reset

Table B–83. HPI Transfer Request Control Register (TRCTL) Field Descriptions

Bit	***field***†	***symval***†	Value	Description
31–9	Reserved	–	0	Reserved. The reserved bit location is always read as 0.
8	TRSTALL			Forces the HPI to stall all HPI requests to the EDMA. This bit allows the safe changing of the PALLOC and PRI fields.
			0	Allows HPI requests to be submitted to the EDMA.
			1	Halts the creation of new HPI requests to the EDMA.
7–6	Reserved	–	0	Reserved. The reserved bit location is always read as 0.
5–4	PRI		0–3h	Controls the priority queue level that HPI requests are submitted to.
			0	Urgent priority
			1h	High priority
			2h	Medium priority
			3h	Low priority
3–0	PALLOC		0–Fh	Controls the total number of outstanding requests that can be submitted by the HPI to the EDMA.

† For CSL implementation, use the notation HPI_TRCTL_*field_symval*

B.7 Inter-Integrated Circuit (I2C) Registers

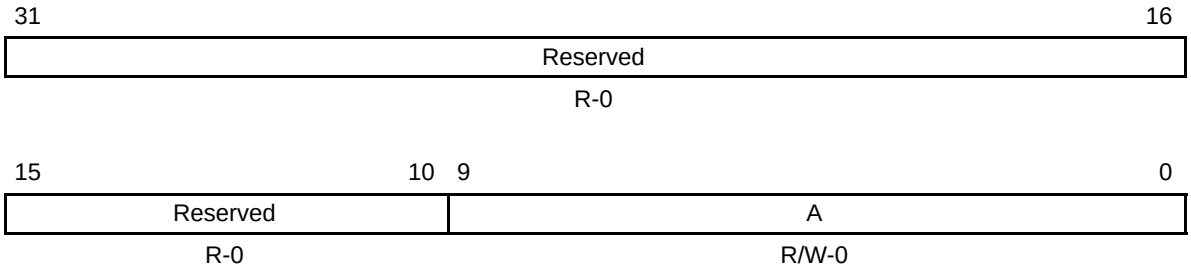
Table B–84. I2C Module Registers

Acronym	Register Name	Address Offset (hex)	Section
I2COAR	I2C own address register	00	B.7.1
I2CIER	I2C interrupt enable register	04	B.7.2
I2CSTR	I2C status register	08	B.7.3
I2CCLKL	I2C clock low-time divider register	0C	B.7.4
I2CCLKH	I2C clock high-time divider register	10	B.7.4
I2CCNT	I2C data count register	14	B.7.5
I2CDRR	I2C data receive register	18	B.7.6
I2CSAR	I2C slave address register	1C	B.7.7
I2CDXR	I2C data transmit register	20	B.7.8
I2CMDR	I2C mode register	24	B.7.9
I2CISR	I2C interrupt source register	28	B.7.10
I2CPSC	I2C prescaler register	30	B.7.11
I2CPID1	I2C peripheral identification register 1	—	B.7.12
I2CPID2	I2C peripheral identification register 2	—	B.7.12
I2CRSR	I2C receive shift register (not accessible to the CPU or EDMA)	—	—
I2CXSR	I2C transmit shift register (not accessible to the CPU or EDMA)	—	—

B.7.1 I2C Own Address Register (I2COAR)

The I2C own address register (I2COAR) is a 32-bit register mapped used to specify its own slave address, which distinguishes it from other slaves connected to the I2C-bus. If the 7-bit addressing mode is selected (XA = 0 in I2CMDR), only bits 6–0 are used; bits 9–7 are ignored. The I2COAR is shown in Figure B–79 and described in Table B–85.

Figure B–79. I2C Own Address Register (I2COAR)



Legend: R = Read only; R/W = Read/write; -*n* = value after reset

Table B–85. I2C Own Address Register (I2COAR) Field Descriptions

Bit	Field	*symval*†	Value	Description
31–10	Reserved	–	0	These Reserved bit locations are always read as zeros. A value written to this field has no effect.
9–0	A	OF(*value*)		**In 7-bit addressing mode (XA = 0 in I2CMDR):**
			00h–7Fh	Bits 6–0 provide the 7-bit slave address of the I2C module. Bits 9–7 are ignored.
				In 10-bit addressing mode (XA = 1 in I2CMDR):
			000h–3FFh	Bits 9–0 provide the 10-bit slave address of the I2C module.

† For CSL C macro implementation, use the notation I2C_I2COAR_A_*symval*

B.7.2 I2C Interrupt Enable Register (I2CIER)

The I2C interrupt enable register (I2CIER) is used by the CPU to individually enable or disable I2C interrupt requests. The I2CIER is shown in Figure B–80 and described Table B–86.

Figure B–80. I2C Interrupt Enable Register (I2CIER)

31	16
Reserved	
R-0	

15	5	4	3	2	1	0
Reserved		ICXRDY	ICRRDY	ARDY	NACK	AL
R-0		R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

Legend: R = Read only; R/W = Read/write; -*n* = value after reset

Table B–86. I2C Interrupt Enable Register (I2CIER) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31–5	Reserved	–	0	These Reserved bit locations are always read as zeros. A value written to this field has no effect.
4	ICXRDY			Transmit-data-ready interrupt enable bit.
		MSK	0	Interrupt request is disabled.
		UNMSK	1	Interrupt request is enabled.
3	ICRRDY			Receive-data-ready interrupt enable bit.
		MSK	0	Interrupt request is disabled.
		UNMSK	1	Interrupt request is enabled.
2	ARDY			Register-access-ready interrupt enable bit.
		MSK	0	Interrupt request is disabled.
		UNMSK	1	Interrupt request is enabled.
1	NACK			No-acknowledgement interrupt enable bit.
		MSK	0	Interrupt request is disabled.
		UNMSK	1	Interrupt request is enabled.

† For CSL C macro implementation, use the notation I2C_I2CIER_*field_symval*

Table B–86. I2C Interrupt Enable Register (I2CIER) Field Descriptions (Continued)

Bit	*field*†	*symval*†	Value	Description
0	AL			Arbitration-lost interrupt enable bit
		MSK	0	Interrupt request is disabled.
		UNMSK	1	Interrupt request is enabled.

† For CSL C macro implementation, use the notation I2C_I2CIER_*field_symval*

B.7.3 I2C Status Register (I2CSTR)

The I2C status register (I2CSTR) is used by the CPU to determine which interrupt has occurred and to read status information. The I2CSTR is shown in Figure B–81 and described in Table B–87.

Figure B–81. I2C Status Register (I2CSTR)

31	16
Reserved	
R-0	

15	14	13	12	11	10	9	8
Reserved		NACKSNT	BB	RSFULL	XSMT	AAS	AD0
R-0		R/W1C-0	R/W1C-0	R-0	R-1	R-0	R-0

7	5	4	3	2	1	0
Reserved		ICXRDY	ICRRDY	ARDY	NACK	AL
R-0		R/W1C-1	R/W1C-0	R/W1C-0	R/W1C-0	R/W1C-0

Legend: R = Read; W1C = Write 1 to clear (writing 0 has no effect); -*n* = value after reset

Table B–87. I2C Status Register (I2CSTR) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31–14	Reserved	–	0	These Reserved bit locations are always read as zeros. A value written to this field has no effect.
13	NACKSNT			NACK sent bit is used when the I2C module is in the receiver mode. One instance in which NACKSNT is affected is when the NACK mode is used (see the description for NACKMOD in section B.7.9).
		NONE	0	NACK is not sent. NACKSNT bit is cleared by any one of the following events: ❏ It is manually cleared. To clear this bit, write a 1 to it. ❏ The I2C module is reset (either when 0 is written to the IRS bit of I2CMDR or when the whole DSP is reset).
		INT CLR	1	NACK is sent: A no-acknowledge bit was sent during the acknowledge cycle on the I^2C-bus.
12	BB			Bus busy bit. BB indicates whether the I^2C-bus is busy or is free for another data transfer.
		NONE	0	Bus is free. BB is cleared by any one of the following events: ❏ The I2C module receives or transmits a STOP bit (bus free). ❏ BB is manually cleared. To clear this bit, write a 1 to it. ❏ The I2C module is reset.
		INT CLR	1	Bus is busy: The I2C module has received or transmitted a START bit on the bus.
11	RSFULL			Receive shift register full bit. RSFULL indicates an overrun condition during reception. Overrun occurs when the receive shift register (I2CRSR) is full with new data but the previous data has not been read from the data receive register (I2CDRR). The new data will not be copied to I2CDRR until the previous data is read. As new bits arrive from the SDA pin, they overwrite the bits in I2CRSR.
		NONE	0	No overrun is detected. RSFULL is cleared by any one of the following events: ❏ I2CDRR is read. ❏ The I2C module is reset.
		INT	1	Overrun is detected.

† For CSL C macro implementation, use the notation I2C_I2CSTR_*field_symval*

Table B–87. I2C Status Register (I2CSTR) Field Descriptions (Continued)

Bit	*field*†	*symval*†	Value	Description
10	XSMT			Transmit shift register empty bit. XSMT indicates that the transmitter has experienced underflow. Underflow occurs when the transmit shift register (I2CXSR) is empty but the data transmit register (I2CDXR) has not been loaded since the last I2CDXR-to-I2CXSR transfer. The next I2CDXR-to-I2CXSR transfer will not occur until new data is in I2CDXR. If new data is not transferred in time, the previous data may be re-transmitted on the SDA pin.
		NONE	0	Underflow is detected.
		INT	1	No underflow is detected. XSMT is set by one of the following events: ❑ Data is written to I2CDXR. ❑ The I2C module is reset.
9	AAS			Addressed-as-slave bit.
		NONE	0	The AAS bit has been cleared by a repeated START condition or by a STOP condition.
		INT	1	The I2C module has recognized its own slave address or an address of all zeros (general call). The AAS bit is also set if the first data word has been received in the free data format (FDF = 1 in I2CMDR).
8	AD0			Address 0 bit.
		NONE	0	AD0 has been cleared by a START or STOP condition.
		INT	1	An address of all zeros (general call) is detected.
7–5	Reserved	–	0	These Reserved bit locations are always read as zeros. A value written to this field has no effect.

† For CSL C macro implementation, use the notation I2C_I2CSTR_*field_symval*

Table B–87. I2C Status Register (I2CSTR) Field Descriptions (Continued)

Bit	*field*†	*symval*†	Value	Description
4	ICXRDY			Transmit-data-ready interrupt flag bit. ICXRDY indicates that the data transmit register (I2CDXR) is ready to accept new data because the previous data has been copied from I2CDXR to the transmit shift register (I2CXSR). The CPU can poll ICXRDY or use the XRDY interrupt request.
		NONE	0	I2CDXR is not ready. ICXRDY is cleared by one of the following events: ❏ Data is written to I2CDXR. ❏ ICXRDY is manually cleared. To clear this bit, write a 1 to it.
		INT CLR	1	I2CDXR is ready: Data has been copied from I2CDXR to I2CXSR. ICXRDY is forced to 1 when the I2C module is reset.
3	ICRRDY			Receive-data-ready interrupt flag bit. ICRRDY indicates that the data receive register (I2CDRR) is ready to be read because data has been copied from the receive shift register (I2CRSR) to I2CDRR. The CPU can poll ICRRDY or use the RRDY interrupt request.
		NONE	0	I2CDRR is not ready. ICRRDY is cleared by any one of the following events: ❏ I2CDRR is read. ❏ ICRRDY is manually cleared. To clear this bit, write a 1 to it. ❏ The I2C module is reset.
		INT CLR	1	I2CDRR is ready: Data has been copied from I2CRSR to I2CDRR.

† For CSL C macro implementation, use the notation I2C_I2CSTR_*field_symval*

Table B–87. I2C Status Register (I2CSTR) Field Descriptions (Continued)

Bit	*field*†	*symval*†	Value	Description
2	ARDY			Register-access-ready interrupt flag bit (only applicable when the I2C module is in the master mode). ARDY indicates that the I2C module registers are ready to be accessed because the previously programmed address, data, and command values have been used. The CPU can poll ARDY or use the ARDY interrupt request.
		NONE	0	The registers are not ready to be accessed. ARDY is cleared by any one of the following events: ❏ The I2C module starts using the current register contents. ❏ ARDY is manually cleared. To clear this bit, write a 1 to it. ❏ The I2C module is reset.
		INT CLR	1	The registers are ready to be accessed. **In the nonrepeat mode (RM = 0 in I2CMDR):** If STP = 0 in I2CMDR, the ARDY bit is set when the internal data counter counts down to 0. If STP = 1, ARDY is not affected (instead, the I2C module generates a STOP condition when the counter reaches 0). **In the repeat mode (RM = 1):** ARDY is set at the end of each data word transmitted from I2CDXR.
1	NACK			No-acknowledgement interrupt flag bit. NACK applies when the I2C module is a transmitter (master or slave). NACK indicates whether the I2C module has detected an acknowledge bit (ACK) or a no-acknowledge bit (NACK) from the receiver. The CPU can poll NACK or use the NACK interrupt request.
		NONE	0	ACK received/NACK is not received. This bit is cleared by any one of the following events: ❏ An acknowledge bit (ACK) has been sent by the receiver. ❏ NACK is manually cleared. To clear this bit, write a 1 to it. ❏ The CPU reads the interrupt source register (I2CISR) when the register contains the code for a NACK interrupt. ❏ The I2C module is reset.
		INT CLR	1	NACK bit is received. The hardware detects that a no-acknowledge (NACK) bit has been received. **Note:** While the I2C module performs a general call transfer, NACK is 1, even if one or more slaves send acknowledgement.

† For CSL C macro implementation, use the notation I2C_I2CSTR_*field_symval*

Table B–87. I2C Status Register (I2CSTR) Field Descriptions (Continued)

Bit	*field*†	*symval*†	Value	Description
0	AL			Arbitration-lost interrupt flag bit (only applicable when the I2C module is a master-transmitter). AL primarily indicates when the I2C module has lost an arbitration contest with another master-transmitter. The CPU can poll AL or use the AL interrupt request.
		NONE	0	Arbitration is not lost. AL is cleared by any one of the following events: ❑ AL is manually cleared. To clear this bit, write a 1 to it. ❑ The CPU reads the interrupt source register (I2CISR) when the register contains the code for an AL interrupt. ❑ The I2C module is reset.
		INT	1	Arbitration is lost. AL is set by any one of the following events:
		CLR		❑ The I2C module senses that it has lost an arbitration with two or more competing transmitters that started a transmission almost simultaneously. ❑ The I2C module attempts to start a transfer while the BB (bus busy) bit is set to 1. When AL becomes 1, the MST and STP bits of I2CMDR are cleared, and the I2C module becomes a slave-receiver.

† For CSL C macro implementation, use the notation I2C_I2CSTR_*field_symval*

B.7.4 I2C Clock Divider Registers (I2CCLKL and I2CCLKH)

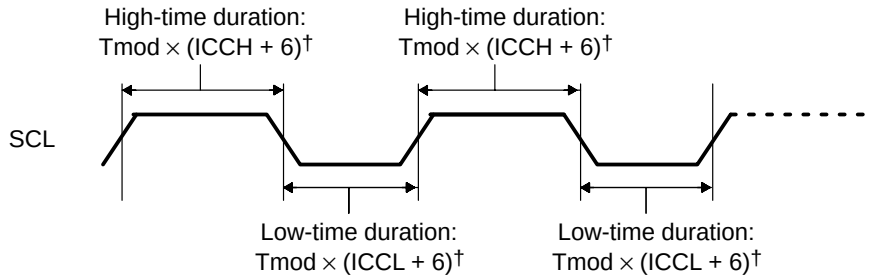
When the I2C module is a master, the module clock is divided down for use as the master clock on the SCL pin. As shown in Figure B–82, the shape of the master clock depends on two divide-down values:

- ICCL in I2CCLKL (shown in Figure B–83 and described in Table B–88). For each master clock cycle, ICCL determines the amount of time the signal is low.
- ICCH in I2CCLKH (shown in Figure B–84 and described in Table B–89). For each master clock cycle, ICCH determines the amount of time the signal is high.

The frequency of the master clock can be calculated as:

$$\text{master clock frequency} = \frac{\text{module clock frequency}}{(\text{ICCL} + 6) + (\text{ICCH} + 6)}$$

Figure B–82. Roles of the Clock Divide-Down Values (ICCL and ICCH)



† Tmod = module clock period = 1 / module clock frequency

Figure B–83. I2C Clock Low-Time Divider Register (I2CCLKL)

31	16
Reserved	
R-0	

15	0
ICCL	
R/W-0	

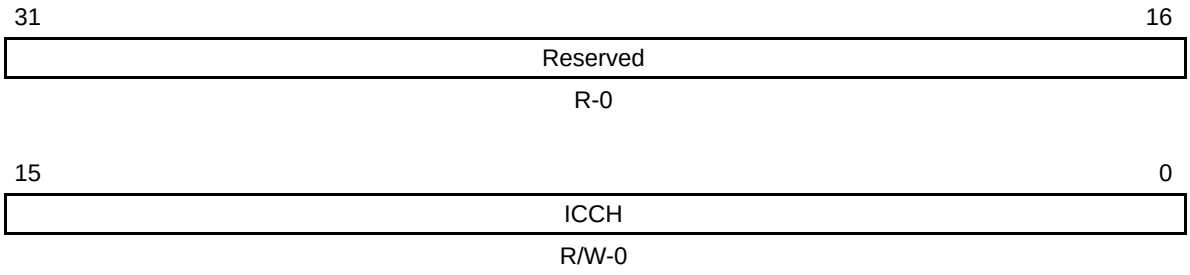
Legend: R = Read only; R/W = Read/write; -*n* = value after reset

Table B–88. I2C Clock Low-Time Divider Register (I2CCLKL) Field Descriptions

Bit	Field	*symval*†	Value	Description
31–16	Reserved	–	0	These Reserved bit locations are always read as zeros. A value written to this field has no effect.
15–0	ICCL	OF(*value*)	0000h–FFFFh	Clock low-time divide-down value of 1–65536. The period of the module clock is multiplied by (ICCL + 6) to produce the low-time duration of the master clock on the SCL pin.

† For CSL C macro implementation, use the notation I2C_I2CCLKL_ICCL_*symval*

Figure B–84. I2C Clock High-Time Divider Register (I2CCLKH)



Legend: R = Read only; R/W = Read/write; -*n* = value after reset

Table B–89. I2C Clock High-Time Divider Register (I2CCLKH) Field Descriptions

Bit	Field	*symval*†	Value	Description
31–16	Reserved	–	0	These Reserved bit locations are always read as zeros. A value written to this field has no effect.
15–0	ICCH	OF(*value*)	0000h–FFFFh	Clock high-time divide-down value of 1–65536. The period of the module clock is multiplied by (ICCH + 6) to produce the high-time duration of the master clock on the SCL pin.

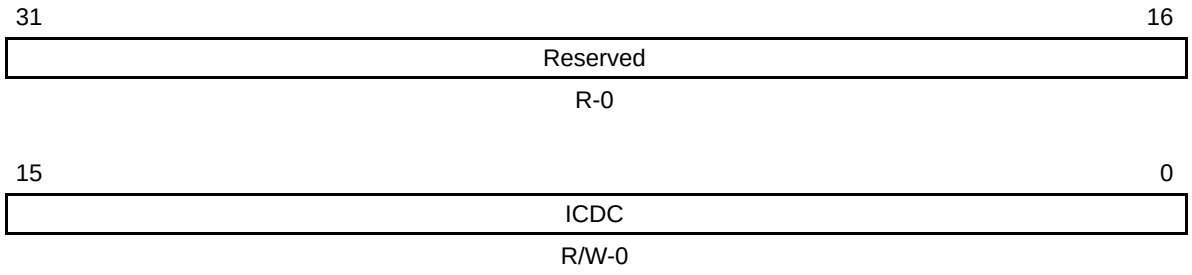
† For CSL C macro implementation, use the notation I2C_I2CCLKH_ICCH_*symval*

B.7.5 I2C Data Count Register (I2CCNT)

The I2C data count register (I2CCNT) is used to indicate how many data words to transfer when the I2C module is configured as a master-transmitter (MST = 1 and TRX = 1 in I2CMDR) and the repeat mode is off (RM = 0 in I2CMDR). In the repeat mode (RM = 1), I2CCNT is not used. The I2CCNT is shown in Figure B–85 and described in Table B–90.

The value written to I2CCNT is copied to an internal data counter. The internal data counter is decremented by 1 for each data word transferred (I2CCNT remains unchanged). If a STOP condition is requested (STP = 1 in I2CMDR), the I2C module terminates the transfer with a STOP condition when the count-down is complete (that is, when the last data word has been transferred).

Figure B–85. I2C Data Count Register (I2CCNT)



Legend: R = Read only; R/W = Read/write; -*n* = value after reset

Table B–90. I2C Data Count Register (I2CCNT) Field Descriptions

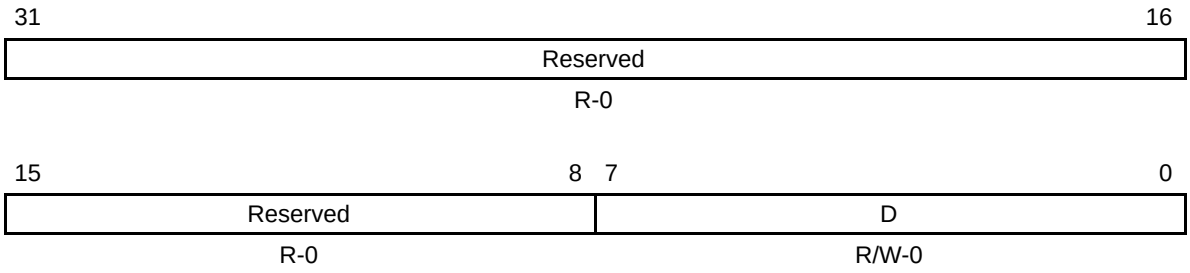
Bit	Field	*symval*†	Value	Description
31–16	Reserved	–	0	These Reserved bit locations are always read as zeros. A value written to this field has no effect.
15–0	ICDC	OF(*value*)		Data count value. ICDC indicates the number of data words to transfer in the nonrepeat mode (RM = 0 in I2CMDR). The value in I2CCNT is a don't care when the RM bit in I2CMDR is set to 1.
			0000h	The start value loaded to the internal data counter is 65536.
			0001h–FFFFh	The start value loaded to internal data counter is 1–65535.

† For CSL C macro implementation, use the notation I2C_I2CCNT_ICDC_*symval*

B.7.6 I2C Data Receive Register (I2CDRR)

The I2C data receive register (I2CDRR) is used by the DSP to read the receive data. The I2CDRR can receive a data value of up to 8 bits; data values with fewer than 8 bits are right-aligned in the D bits and the remaining D bits are undefined. The number of data bits is selected by the bit count bits (BC) of I2CMDR. The I2C receive shift register (I2CRSR) shifts in the received data from the SDA pin. Once data is complete, the I2C module copies the contents of I2CRSR into I2CDRR. The CPU and the EDMA controller cannot access I2CRSR. The I2CDRR is shown in Figure B–86 and described in Table B–91.

Figure B–86. I2C Data Receive Register (I2CDRR)



Legend: R = Read only; R/W = Read/write; -*n* = value after reset

Table B–91. I2C Data Receive Register (I2CDRR) Field Descriptions

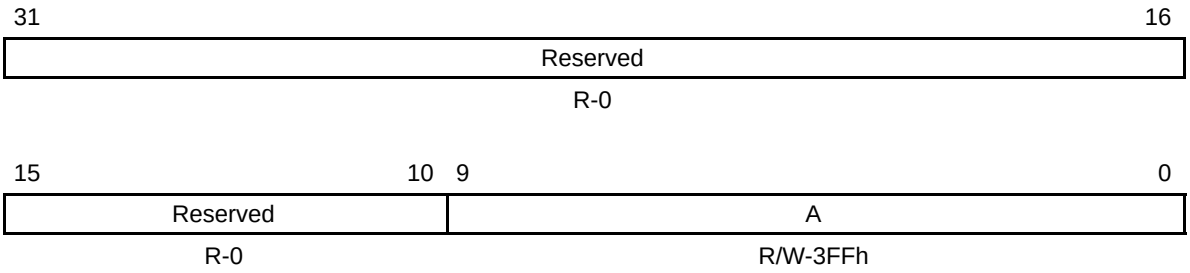
Bit	Field	*symval*†	Value	Description
31–8	Reserved	_	0	These Reserved bit locations are always read as zeros. A value written to this field has no effect.
7–0	D	OF(value)	00h–FFh	Receive data.

† For CSL C macro implementation, use the notation I2C_I2CDRR_D_*symval*

B.7.7 I2C Slave Address Register (I2CSAR)

The I2C slave address register (I2CSAR) contains a 7-bit or 10-bit slave address. When the I2C module is not using the free data format (FDF = 0 in I2CMDR), it uses this address to initiate data transfers with a slave or slaves. When the address is nonzero, the address is for a particular slave. When the address is 0, the address is a general call to all slaves. If the 7-bit addressing mode is selected (XA = 0 in I2CMDR), only bits 6–0 of I2CSAR are used; bits 9–7 are ignored. The I2CSAR is shown in Figure B–87 and described in Table B–92.

Figure B–87. I2C Slave Address Register (I2CSAR)



Legend: R = Read; W = Write; -*n* = Value after reset

Table B–92. I2C Slave Address Register (I2CSAR) Field Descriptions

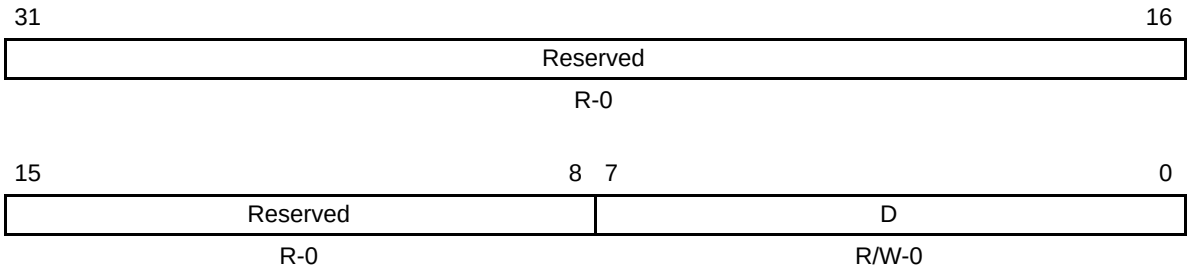
Bit	Field	*symval*†	Value	Description
31–10	Reserved	–	0	These Reserved bit locations are always read as zeros. A value written to this field has no effect.
9–0	A	OF(*value*)		**In 7-bit addressing mode (XA = 0 in I2CMDR):**
			00h–7Fh	Bits 6–0 provide the 7-bit slave address that the I2C module transmits when it is in the master-transmitter mode. Bits 9–7 are ignored.
				In 10-bit addressing mode (XA = 1 in I2CMDR):
			000h–3FFh	Bits 9–0 provide the 10-bit slave address that the I2C module transmits when it is in the master-transmitter mode.

† For CSL C macro implementation, use the notation I2C_I2CSAR_A_*symval*

B.7.8 I2C Data Transmit Register (I2CDXR)

The DSP writes transmit data to the I2C data transmit register (I2CDXR). The I2CDXR can accept a data value of up to 8 bits. When writing a data value with fewer than 8 bits, the DSP must make sure that the value is right-aligned in the D bits. The number of data bits is selected by the bit count bits (BC) of I2CMDR. Once data is written to I2CDXR, the I2C module copies the contents of I2CDXR into the I2C transmit shift register (I2CXSR). The I2CXSR shifts out the transmit data from the SDA pin. The CPU and the EDMA controller cannot access I2CXSR. The I2CDXR is shown in Figure B–88 and described in Table B–93.

Figure B–88. I2C Data Transmit Register (I2CDXR)



Legend: R = Read only; R/W = Read/write; -*n* = value after reset

Table B–93. I2C Data Transmit Register (I2CDXR) Field Descriptions

Bit	Field	*symval*†	Value	Description
31–8	Reserved	–	0	These Reserved bit locations are always read as zeros. A value written to this field has no effect.
7–0	D	OF(value)	00h–FFh	Transmit data

† For CSL C macro implementation, use the notation I2C_I2CDXR_D_*symval*

B.7.9 I2C Mode Register (I2CMDR)

The I2C mode register (I2CMDR) contains the control bits of the I2C module. The I2CMDR is shown in Figure B–89 and described in Table B–94.

Figure B–89. I2C Mode Register (I2CMDR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
R-0															

15	14	13	12	11	10	9	8
NACKMOD	FREE	STT	Reserved	STP	MST	TRX	XA
R/W-0	R/W-0	R/W-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0

7	6	5	4	3	2	1	0
RM	DLB	IRS	STB	FDF	BC		
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0		

Legend: R/W = Read/write; -*n* = value after reset

Table B–94. I2C Mode Register (I2CMDR) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31–16	Reserved	–	0	These Reserved bit locations are always read as zeros. A value written to this field has no effect.
15	NACKMOD			NACK mode bit is only applicable when the I2C module is acting as a receiver.
		ACK	0	**In slave-receiver mode:** The I2C module sends an acknowledge (ACK) bit to the transmitter during the each acknowledge cycle on the bus. The I2C module only sends a no-acknowledge (NACK) bit if you set the NACKMOD bit. **In master-receiver mode:** The I2C module sends an ACK bit during each acknowledge cycle until the internal data counter counts down to 0. At that point, the I2C module sends a NACK bit to the transmitter. To have a NACK bit sent earlier, you must set the NACKMOD bit.
		NACK	1	**In either slave-receiver or master-receiver mode:** The I2C module sends a NACK bit to the transmitter during the next acknowledge cycle on the bus. Once the NACK bit has been sent, NACKMOD is cleared. To send a NACK bit in the next acknowledge cycle, you must set NACKMOD before the rising edge of the last data bit.

† For CSL C macro implementation, use the notation I2C_I2CMDR_*field_symval*

Table B–94. I2C Mode Register (I2CMDR) Field Descriptions (Continued)

Bit	*field*†	*symval*†	Value	Description
14	FREE			This emulation mode bit is used to determine what the state of the I2C module will be when a breakpoint is encountered in the high-level language debugger.
		BSTOP	0	**When I2C module is master:** If SCL is low when the breakpoint occurs, the I2C module stops immediately and keeps driving SCL low, whether the I2C module is the transmitter or the receiver. If SCL is high, the I2C module waits until SCL becomes low and then stops. **When I2C module is slave:** A breakpoint forces the I2C module to stop when the current transmission/reception is complete.
		RFREE	1	The I2C module runs free; that is, it continues to operate when a breakpoint occurs.
13	STT			START condition bit (only applicable when the I2C module is a master). The RM, STT, and STP bits determine when the I2C module starts and stops data transmissions (see Table B–95). Note that the STT and STP bits can be used to terminate the repeat mode.
		NONE	0	In the master mode, STT is automatically cleared after the START condition has been generated. In the slave mode, if STT is 0, the I2C module does not monitor the bus for commands from a master. As a result, the I2C module performs no data transfers.
		START	1	In the master mode, setting STT to 1 causes the I2C module to generate a START condition on the I^2C-bus. In the slave mode, if STT is 1, the I2C module monitors the bus and transmits/receives data in response to commands from a master.
12	Reserved	–	0	This Reserved bit location is always read as zero. A value written to this field has no effect.

† For CSL C macro implementation, use the notation I2C_I2CMDR_*field_symval*

Table B–94. I2C Mode Register (I2CMDR) Field Descriptions (Continued)

Bit	*field*†	*symval*†	Value	Description
11	STP			STOP condition bit (only applicable when the I2C module is a master). In the master mode, the RM, STT, and STP bits determine when the I2C module starts and stops data transmissions (see Table B–95). Note that the STT and STP bits can be used to terminate the repeat mode.
		NONE	0	STP is automatically cleared after the STOP condition has been generated.
		STOP	1	STP has been set by the DSP to generate a STOP condition when the internal data counter of the I2C module counts down to 0.
10	MST			Master mode bit. MST determines whether the I2C module is in the slave mode or the master mode. MST is automatically changed from 1 to 0 when the I2C master generates a STOP condition.
		SLAVE	0	Slave mode. The I2C module is a slave and receives the serial clock from the master.
		MASTER	1	Master mode. The I2C module is a master and generates the serial clock on the SCL pin.
9	TRX			Transmitter mode bit. When relevant, TRX selects whether the I2C module is in the transmitter mode or the receiver mode. Table B–96 summarizes when TRX is used and when it is a don't care.
		RCV	0	Receiver mode. The I2C module is a receiver and receives data on the SDA pin.
		XMT	1	Transmitter mode. The I2C module is a transmitter and transmits data on the SDA pin.
8	XA			Expanded address enable bit.
		7BIT	0	7-bit addressing mode (normal address mode). The I2C module transmits 7-bit slave addresses (from bits 6–0 of I2CSAR), and its own slave address has 7 bits (bits 6–0 of I2COAR).
		10BIT	1	10-bit addressing mode (expanded address mode). The I2C module transmits 10-bit slave addresses (from bits 9–0 of I2CSAR), and its own slave address has 10 bits (bits 9–0 of I2COAR).

† For CSL C macro implementation, use the notation I2C_I2CMDR_*field_symval*

Table B–94. I2C Mode Register (I2CMDR) Field Descriptions (Continued)

Bit	*field*†	*symval*†	Value	Description
7	RM			Repeat mode bit (only applicable when the I2C module is a master-transmitter). The RM, STT, and STP bits determine when the I2C module starts and stops data transmissions (see Table B–95).
		NONE	0	Nonrepeat mode. The value in the data count register (I2CCNT) determines how many data words are received/ transmitted by the I2C module.
		REPEAD	1	Repeat mode. Data words are continuously received/ transmitted by the I2C module until the STP bit is manually set to 1, regardless of the value in I2CCNT.
6	DLB			Digital loopback mode bit. This bit disables or enables the digital loopback mode of the I2C module. The effects of this bit are shown in Figure B–90. Note that DLB in the free data format mode (DLB = 1 and FDF = 1) is not supported.
		NONE	0	Digital loopback mode is disabled.
		LOOPBACK	1	Digital loopback mode is enabled. In this mode, the MST bit must be set to 1 and data transmitted out of I2CDXR is received in I2CDRR after *n* DSP cycles by an internal path, where: *n* = ((I2C input clock frequency/module clock frequency) $\times$ 8) The transmit clock is also the receive clock. The address transmitted on the SDA pin is the address in I2COAR.
5	IRS			I2C module reset bit.
		RST	0	The I2C module is in reset/disabled. When this bit is cleared to 0, all status bits (in I2CSTR) are set to their default values.
		NRST	1	The I2C module is enabled.

† For CSL C macro implementation, use the notation I2C_I2CMDR_*field_symval*

Table B–94. I2C Mode Register (I2CMDR) Field Descriptions (Continued)

Bit	*field*†	*symval*†	Value	Description
4	STB			START byte mode bit. This bit is only applicable when the I2C module is a master. As described in version 2.1 of the Philips I^{2}C-bus specification, the START byte can be used to help a slave that needs extra time to detect a START condition. When the I2C module is a slave, the I2C module ignores a START byte from a master, regardless of the value of the STB bit.
		NONE	0	The I2C module is not in the START byte mode.
		SET	1	The I2C module is in the START byte mode. When you set the START condition bit (STT), the I2C module begins the transfer with more than just a START condition. Specifically, it generates: 1) A START condition 2) A START byte (0000 0001b) 3) A dummy acknowledge clock pulse 4) A repeated START condition The I2C module sends the slave address that is in I2CSAR.
3	FDF			Free data format mode bit. Note that DLB in the free data format mode (DLB = 1 and FDF = 1) is not supported.
		NONE	0	Free data format mode is disabled. Transfers use the 7-/10-bit addressing format selected by the XA bit.
		SET	1	Free data format mode is enabled. Transfers have the free data (no address) format.

† For CSL C macro implementation, use the notation I2C_I2CMDR_*field_symval*

Table B–94. I2C Mode Register (I2CMDR) Field Descriptions (Continued)

Bit	*field*†	*symval*†	Value	Description
2–0	BC	OF(*value*)		Bit count bits. BC defines the number of bits (1 to 8) in the next data word that is to be received or transmitted by the I2C module. The number of bits selected with BC must match the data size of the other device. Notice that when BC = 000b, a data word has 8 bits. If the bit count is less than 8, receive data is right aligned in the D bits of I2CDRR and the remaining D bits are undefined. Also, transmit data written to I2CDXR must be right aligned.
		BIT8FDF	0	8 bits per data word
		BIT1FDF	1h	1 bit per data word
		BIT2FDF	2h	2 bits per data word
		BIT3FDF	3h	3 bits per data word
		BIT4FDF	4h	4 bits per data word
		BIT5FDF	5h	5 bits per data word
		BIT6FDF	6h	6 bits per data word
		BIT7FDF	7h	7 bits per data word

† For CSL C macro implementation, use the notation I2C_I2CMDR_*field_symval*

Table B–95. Master-Transmitter/Receiver Bus Activity Defined by RM, STT, and STP Bits

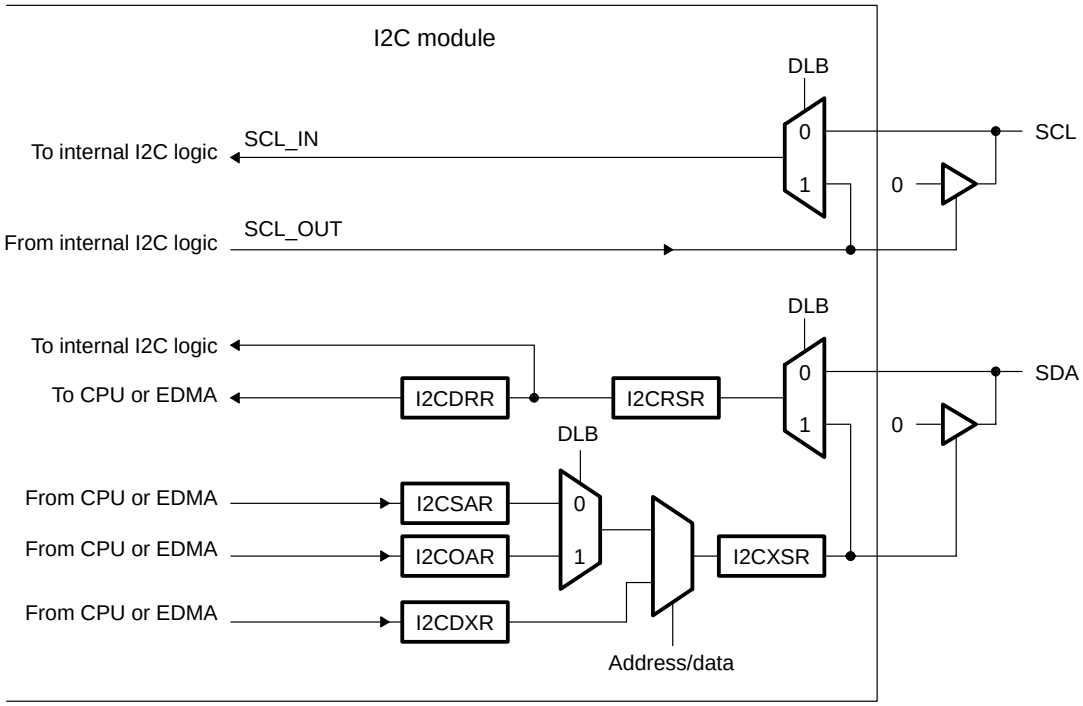
I2CMDR Bit				
RM	**STT**	**STP**	**Bus Activity†**	**Description**
0	0	0	None	No activity
0	0	1	P	STOP condition
0	1	0	S-A-D..(*n*)..D	START condition, slave address, *n* data words (*n* = value in I2CCNT)
0	1	1	S-A-D..(*n*)..D-P	START condition, slave address, *n* data words, STOP condition (*n* = value in I2CCNT)
1	0	0	None	No activity
1	0	1	P	STOP condition
1	1	0	S-A-D-D-D…..	Repeat mode transfer: START condition, slave address, continuous data transfers until STOP condition or next START condition
1	1	1	None	Reserved bit combination (No activity)

† A = Address; D = Data word; P = STOP condition; S = START condition

Table B–96. How the MST and FDF Bits Affect the Role of TRX Bit

I2CMDR Bit			
MST	**FDF**	**I2C Module State**	**Function of TRX Bit**
0	0	In slave mode but not free data format mode	TRX is a don't care. Depending on the command from the master, the I2C module responds as a receiver or a transmitter.
0	1	In slave mode and free data format mode	The free data format mode requires that the transmitter and receiver be fixed. TRX identifies the role of the I2C module: TRX = 0: The I2C module is a receiver. TRX = 1: The I2C module is a transmitter.
1	X	In master mode; free data format mode on or off	TRX = 0: The I2C module is a receiver. TRX = 1: The I2C module is a transmitter.

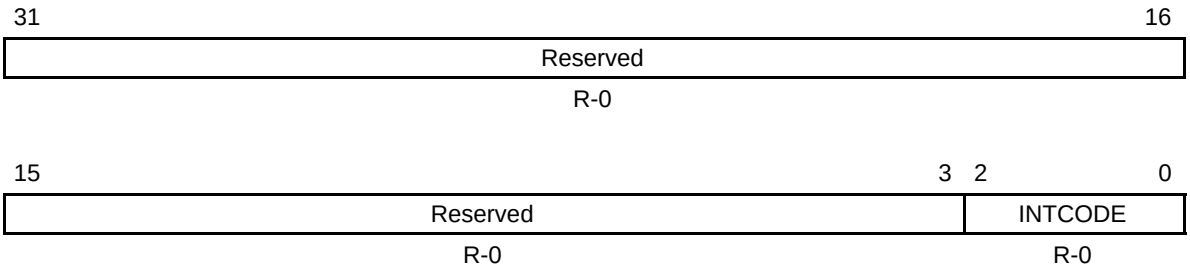
Figure B–90. Block Diagram Showing the Effects of the Digital Loopback Mode (DLB) Bit



B.7.10 I2C Interrupt Source Register (I2CISR)

The I2C interrupt source register (I2CISR) is used by the CPU to determine which event generated the I2C interrupt. The I2CISR is shown in Figure B–91 and described in Table B–97.

Figure B–91. I2C Interrupt Source Register (I2CISR)



Legend: R = Read only; R/W = Read/write; -*n* = value after reset

Table B–97. I2C Interrupt Source Register (I2CISR) Field Descriptions

Bit	Field	*symval*†	Value	Description
31–3	Reserved	–	0	These Reserved bit locations are always read as zeros. Always write 0 to this field.
2–0	INTCODE			Interrupt code bits. The binary code in INTCODE indicates which event generated an I2C interrupt.
		NONE	000	None
		AL	001	Arbitration is lost.
		NACK	010	No-acknowledgement condition is detected.
		RAR	011	Registers are ready to be accessed.
		RDR	100	Receive data is ready.
		XDR	101	Transmit data is ready.
		–	110–111	Reserved

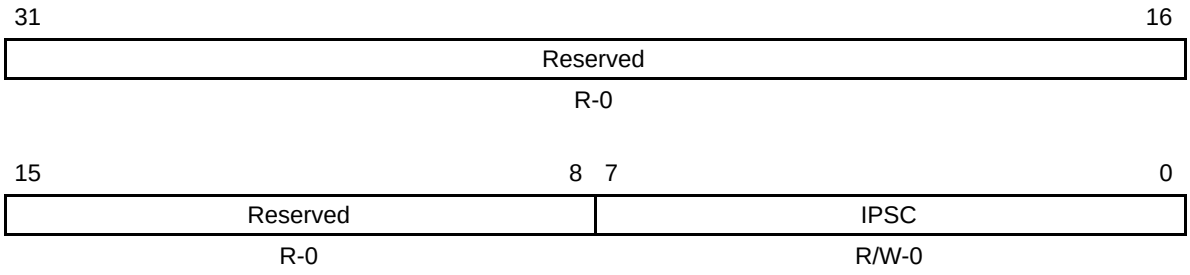
† For CSL C macro implementation, use the notation I2C_I2CISR_INTCODE_*symval*

B.7.11 I2C Prescaler Register (I2CPSC)

The I2C prescaler register (I2CPSC) is used for dividing down the I2C input clock to obtain the desired module clock for the operation of the I2C module. The I2CPSC is shown in Figure B–92 and described in Table B–98.

The IPSC bits must be initialized while the I2C module is in reset (IRS = 0 in I2CMDR). The prescaled frequency takes effect only when the IRS bit is changed to 1. Changing the IPSC value while IRS = 1 has no effect.

Figure B–92. I2C Prescaler Register (I2CPSC)



Legend: R = Read; W = Write; -*n* = Value after reset

Table B–98. I2C Prescaler Register (I2CPSC) Field Descriptions

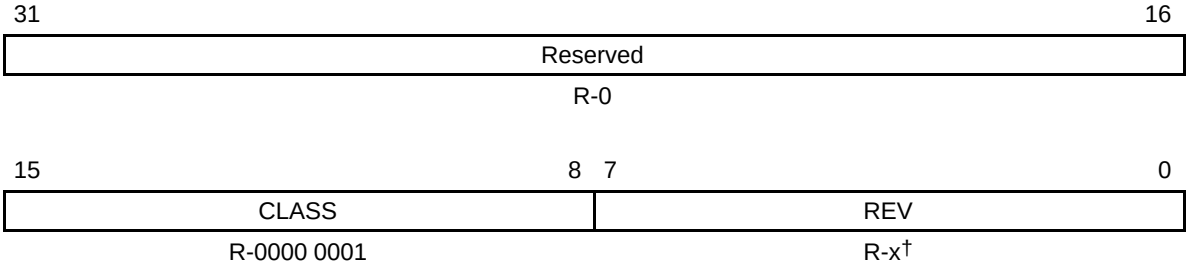
Bit	Field	*symval*†	Value	Description
31–8	Reserved	–	0	These Reserved bit locations are always read as zeros. A value written to this field has no effect.
7–0	IPSC	OF(*value*)	00h–FFh	I2C prescaler divide-down value. IPSC determines how much the CPU clock is divided to create the module clock of the I2C module: module clock frequency = I2C input clock frequency/(IPSC + 1) **Note:** IPSC must be initialized while the I2C module is in reset (IRS = 0 in I2CMDR).

† For CSL C macro implementation, use the notation I2C_I2CPSC_IPSC_*symval*

B.7.12 I2C Peripheral Identification Registers (I2CPID1 and I2CPID2)

The peripheral identification registers (PID) contain identification data for the I2C module. I2CPID1 is shown in Figure B–93 and described in Table B–99. I2CPID2 is shown in Figure B–94 and described in Table B–100.

Figure B–93. I2C Peripheral Identification Register 1 (I2CPID1)

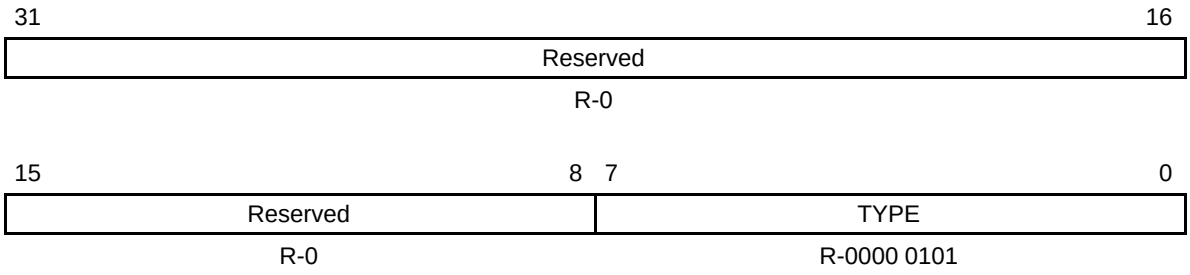


Legend: R = Read only; -x = value after reset
† See the device-specific datasheet for the default value of this field.

Table B–99. I2C Peripheral Identification Register 1 (I2CPID1) Field Descriptions

Bit	Field	Value	Description
31–16	Reserved	0	These Reserved bit locations are always read as zeros. A value written to this field has no effect.
15–8	CLASS		Identifies class of peripheral.
		1	Serial port
7–0	REV		Identifies revision of peripheral.
		x	See the device-specific datasheet for the value.

Figure B–94. I2C Peripheral Identification Register 2 (I2CPID2)



Legend: R = Read only; -x = value after reset

Table B–100. I2C Peripheral Identification Register 2 (I2CPID2) Field Descriptions

Bit	Field	Value	Description
31–8	Reserved	0	These Reserved bit locations are always read as zeros. A value written to this field has no effect.
7–0	TYPE		Identifies type of peripheral.
		05h	I2C

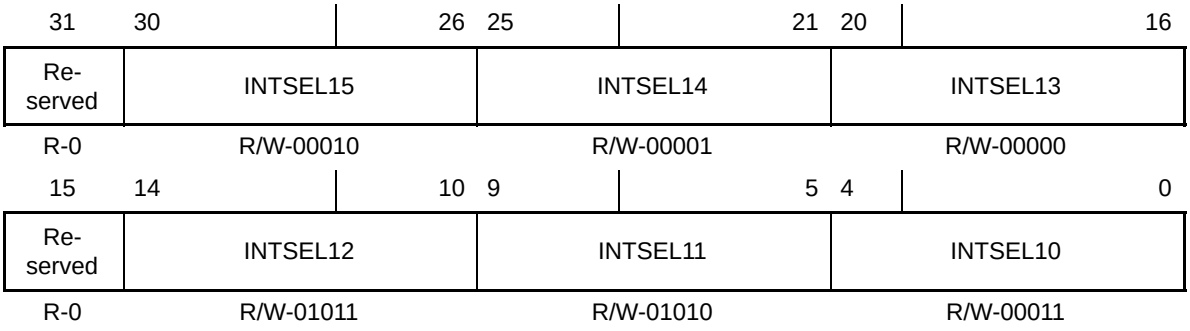
B.8 Interrupt Request (IRQ) Registers

Table B–101. IRQ Registers

Acronym	Register Name	Section
MUXH	Interrupt multiplexter high register	B.8.1
MUXL	Interrupt multiplexter low register	B.8.2
EXTPOL	External interrupt polarity register	B.8.3

B.8.1 Interrupt Multiplexer High Register (MUXH)

Figure B–95. Interrupt Multiplexer High Register (MUXH)



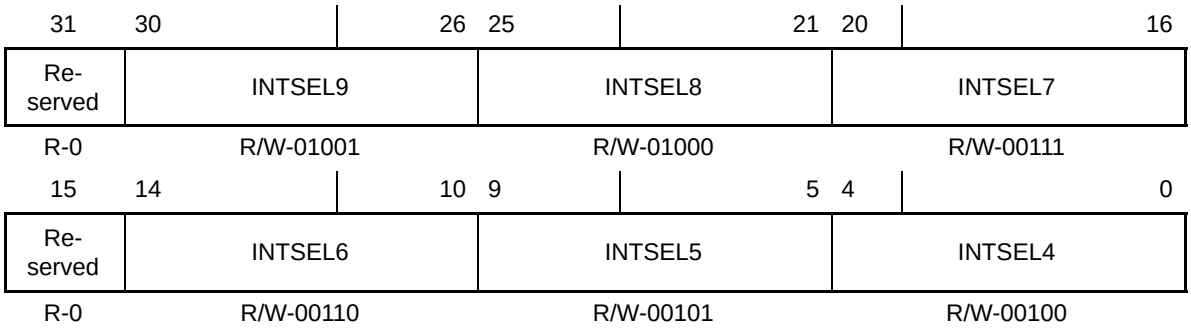
Legend: R/W-x = Read/Write-Reset value

Table B–102. Interrupt Multiplexer High Register (MUXH) Field Values (IRQ_MUXH_field_symval)

Bit	*field*	*symval*	Value	Description
31	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
30–26	INTSEL15	OF(*value*)	0–31	
25–21	INTSEL14	OF(*value*)	0–31	
20–16	INTSEL13	OF(*value*)	0–31	
15	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
14–10	INTSEL12	OF(*value*)	0–31	
9–5	INTSEL11	OF(*value*)	0–31	
4–0	INTSEL10	OF(*value*)	0–31	

B.8.2 Interrupt Multiplexer Low Register (MUXL)

Figure B–96. Interrupt Multiplexer Low Register (MUXL)



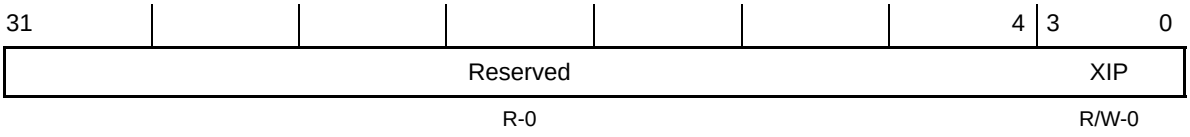
Legend: R/W-x = Read/Write-Reset value

Table B–103. Interrupt Multiplexer Low Register (MUXL) Field Values (IRQ_MUXL_field_symval)

Bit	*field*	*symval*	Value	Description
31	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
30–26	INTSEL9	OF(*value*)	0–31	
25–21	INTSEL8	OF(*value*)	0–31	
20–16	INTSEL7	OF(*value*)	0–31	
15	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
14–10	INTSEL6	OF(*value*)	0–31	
9–5	INTSEL5	OF(*value*)	0–31	
4–0	INTSEL4	OF(*value*)	0–31	

B.8.3 External Interrupt Polarity Register (EXTPOL)

Figure B–97. External Interrupt Polarity Register (EXTPOL)



Legend: R/W-x = Read/Write-Reset value

Table B–104. External Interrupt Polarity Register (EXTPOL) Field Values (IRQ_EXTPOL_field_symval)

Bit	*field*	*symval*	Value	Description
31–4	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
3–0	XIP	OF(*value*)	0–15	

B.9 MDIO Module Registers

Control registers for the MDIO module are summarized in Table B–105. See the device-specific datasheet for the memory address of these registers.

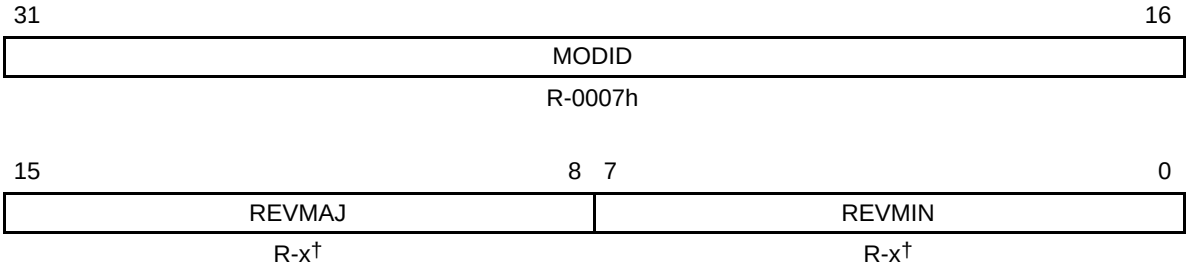
Table B–105. MDIO Module Registers

Acronym	Register Name	Section
VERSION	MDIO Version Register	B.9.1
CONTROL	MDIO Control Register	B.9.2
ALIVE	MDIO PHY Alive Indication Register	B.9.3
LINK	MDIO PHY Link Status Register	B.9.4
LINKINTRAW	MDIO Link Status Change Interrupt Register	B.9.5
LINKINTMASKED	MDIO Link Status Change Interrupt (Masked) Register	B.9.6
USERINTRAW	MDIO User Command Complete Interrupt Register	B.9.7
USERINTMASKED	MDIO User Command Complete Interrupt (Masked) Register	B.9.8
USERINTMASKSET	MDIO User Command Complete Interrupt Mask Set Register	B.9.9
USERINTMASKCLEAR	MDIO User Command Complete Interrupt Mask Clear Register	B.9.10
USERACCESS0	MDIO User Access Register 0	B.9.11
USERACCESS1	MDIO User Access Register 1	B.9.12
USERPHYSEL0	MDIO User PHY Select Register 0	B.9.13
USERPHYSEL1	MDIO User PHY Select Register 1	B.9.14

B.9.1 MDIO Version Register (VERSION)

The MDIO version register (VERSION) is shown in Figure B–98 and described in Table B–106.

Figure B–98. MDIO Version Register (VERSION)



Legend: R = Read only; -*n* = value after reset
† See the device-specific datasheet for the default value of this field.

Table B–106. MDIO Version Register (VERSION) Field Descriptions

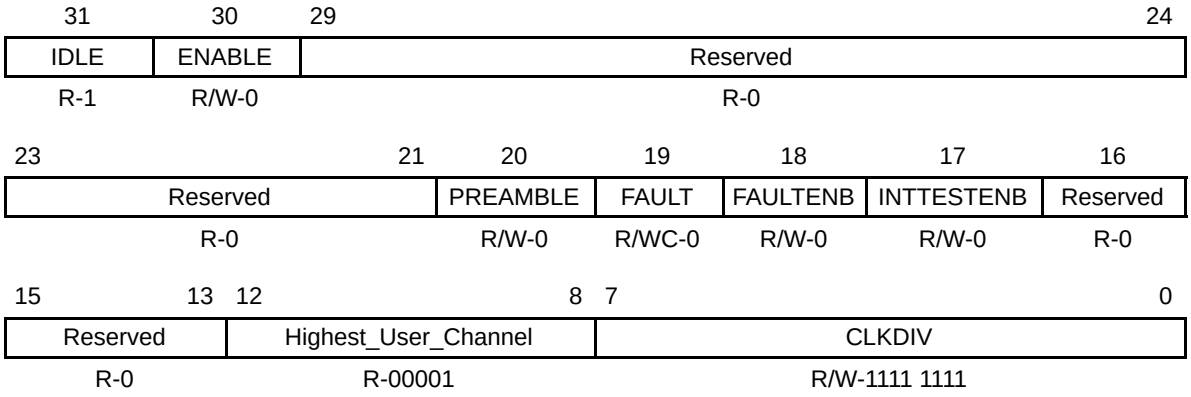
Bit	***field***†	***symval***†	Value	Description
31–16	MODID			Identifies type of peripheral.
			7h	MDIO
15–8	REVMAJ			Identifies major revision of peripheral.
			x	See the device-specific datasheet for the value.
7–0	REVMIN			Identifies minor revision of peripheral.
			x	See the device-specific datasheet for the value.

† For CSL implementation, use the notation MDIO_VERSION_*field_symval*

B.9.2 MDIO Control Register (CONTROL)

The MDIO control register (CONTROL) is shown in Figure B–99 and described in Table B–107.

Figure B–99. MDIO Control Register (CONTROL)



Legend: R = Read only; WC = Write to clear; R/W = Read/Write; -*n* = value after reset

Table B–107. MDIO Control Register (CONTROL) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31	IDLE			MDIO state machine IDLE status bit.
		NO	0	State machine is not in the idle state.
		YES	1	State machine is in the idle state.
30	ENABLE			MDIO state machine enable control bit. If the MDIO state machine is active at the time it is disabled, it completes the current operation before halting and setting the IDLE bit. If using byte access, the ENABLE bit has to be the last bit written in this register.
		NO	0	Disables the MDIO state machine.
		YES	1	Enables the MDIO state machine.
29–21	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
20	PREAMBLE			MDIO frame preamble disable bit.
		ENABLED	0	Standard MDIO preamble is used.
		DISABLED	1	Disables this device from sending MDIO frame preambles.

† For CSL implementation, use the notation MDIO_CONTROL_*field_symval*

Table B–107. MDIO Control Register (CONTROL) Field Descriptions (Continued)

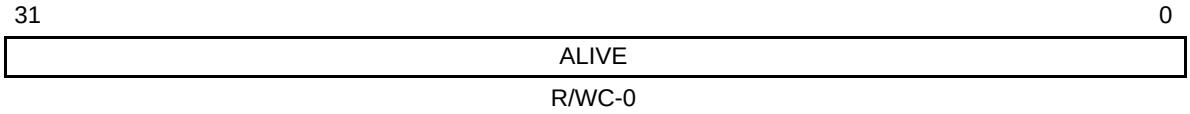
Bit	*field*†	*symval*†	Value	Description
19	FAULT			Fault indicator bit. Writing a 1 to this bit clears this bit.
		NO	0	No failure.
		YES	1	The MDIO pins fail to read back what the device is driving onto them indicating a physical layer fault. The MDIO state machine is reset.
18	FAULTENB			Fault detect enable bit.
		NO	0	Disables the physical layer fault detection.
		YES	1	Enables the physical layer fault detection.
17	INTTESTENB			Interrupt test enable bit.
		NO	0	Interrupt test bits are not set.
		YES	1	Enables the host to set the USERINTRAW, USERINTMASKED, LINKINTRAW, and LINKINTMASKED register bits for test purposes.
16–13	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
12–8	Highest_User _Channel		0–1Fh	Highest user-access channel bits specify the highest user-access channel that is available in the MDIO and is currently set to 1.
7–0	CLKDIV		0–FFh	Clock divider bits. Specifies the division ratio between peripheral clock and the frequency of MDCLK. MDCLK is disabled when CLKDIV is cleared to 0. MDCLK frequency = peripheral clock/(CLKDIV + 1).
			0	MDCLK is disabled.

† For CSL implementation, use the notation MDIO_CONTROL_*field_symval*

B.9.3 MDIO PHY Alive Indication Register (ALIVE)

The MDIO PHY alive indication register (ALIVE) is shown in Figure B–100 and described in Table B–108.

Figure B–100. MDIO PHY Alive Indication Register (ALIVE)



Legend: R = Read only; WC = Write 1 to clear, write of 0 has no effect; -*n* = value after reset

Table B–108. MDIO PHY Alive Indication Register (ALIVE) Field Descriptions

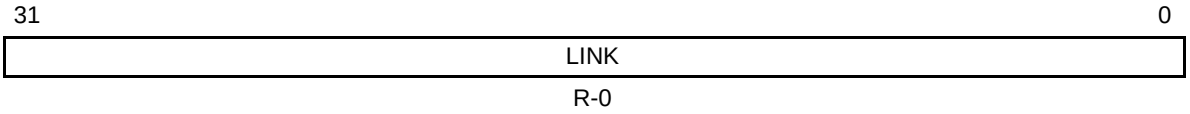
Bit	Field	*symval*†	Value	Description
31–0	ALIVE			MDIO ALIVE bits. Both user and polling accesses to a PHY cause the corresponding ALIVE bit to be updated. The ALIVE bits are only meant to give an indication of the presence or not of a PHY with the corresponding address. Writing a 1 to any bit clears that bit, writing a 0 has no effect.
			0	The PHY fails to acknowledge the access.
			1	The most recent access to the PHY with an address corresponding to the register bit number was acknowledged by the PHY.

† For CSL implementation, use the notation MDIO_ALIVE_ALIVE_*symval*

B.9.4 MDIO PHY Link Status Register (LINK)

The MDIO PHY link status register (LINK) is shown in Figure B–101 and described in Table B–109.

Figure B–101. MDIO PHY Link Status Register (LINK)



Legend: R = Read only; -*n* = value after reset

Table B–109. MDIO PHY Link Status Register (LINK) Field Descriptions

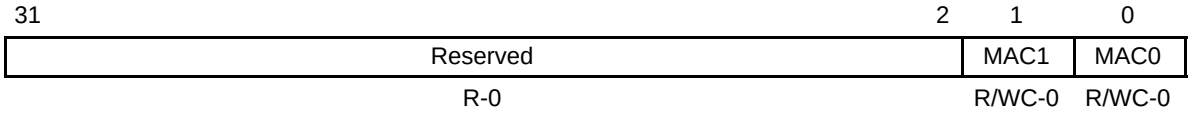
Bit	Field	*symval*†	Value	Description
31–0	LINK			MDIO link state bits. These bits are updated after a read of the PHY generic status register. Writes to these bits have no effect.
			0	The PHY indicates it does not have a link or fails to acknowledge the read transaction.
			1	The PHY with the corresponding address has a link and the PHY acknowledges the read transaction.

† For CSL implementation, use the notation MDIO_LINK_LINK_*symval*

B.9.5 MDIO Link Status Change Interrupt Register (LINKINTRAW)

The MDIO PHY link status change interrupt register (LINKINTRAW) is shown in Figure B–102 and described in Table B–110.

Figure B–102. MDIO Link Status Change Interrupt Register (LINKINTRAW)



Legend: R = Read only; WC = Write 1 to clear, write of 0 has no effect; -*n* = value after reset

Table B–110. MDIO Link Status Change Interrupt Register (LINKINTRAW) Field Descriptions

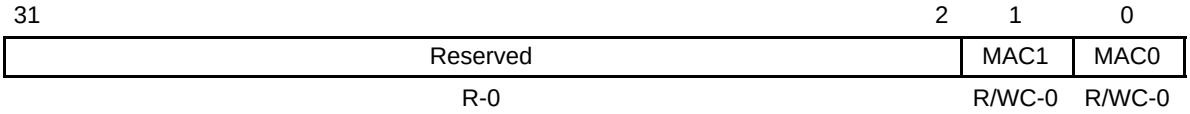
Bit	*field*†	*symval*†	Value	Description
31–2	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
1	MAC1			MDIO link change event bit. Writing a 1 clears the event and writing a 0 has no effect. If the INTTESTENB bit in the MDIO control register is set to 1, the host may set the MAC1 bit to 1 for test purposes.
		NO	0	No MDIO link change event.
		YES	1	An MDIO link change event (change in the MDIO PHY link status register) corresponding to the PHY address in MDIO user PHY select register 1 (USERPHYSEL1).
0	MAC0			MDIO link change event bit. Writing a 1 clears the event and writing a 0 has no effect. If the INTTESTENB bit in the MDIO control register is set to 1, the host may set the MAC0 bit to 1 for test purposes.
		NO	0	No MDIO link change event.
		YES	1	An MDIO link change event (change in the MDIO PHY link status register) corresponding to the PHY address in MDIO user PHY select register 0 (USERPHYSEL0).

† For CSL implementation, use the notation MDIO_LINKINTRAW_*field_symval*

B.9.6 MDIO Link Status Change Interrupt (Masked) Register (LINKINTMASKED)

The MDIO PHY link status change interrupt (masked) register (LINKINTMASKED) is shown in Figure B–103 and described in Table B–111.

Figure B–103. MDIO Link Status Change Interrupt (Masked) Register (LINKINTMASKED)



Legend: R = Read only; WC = Write 1 to clear, write of 0 has no effect; -*n* = value after reset

Table B–111. MDIO Link Status Change Interrupt (Masked) Register (LINKINTMASKED) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31–2	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
1	MAC1			MDIO link change interrupt bit. Writing a 1 clears the interrupt and writing a 0 has no effect. If the INTTESTENB bit in the MDIO control register is set to 1, the host may set the MAC1 bit to 1 for test purposes.
		NO	0	No MDIO link change event.
		YES	1	An MDIO link change event (change in the MDIO PHY link status register) corresponding to the PHY address in MDIO user PHY select register 1 (USERPHYSEL1) and the LINKINTENB bit in USERPHYSEL1 is set to 1.
0	MAC0			MDIO link change interrupt bit. Writing a 1 clears the interrupt and writing a 0 has no effect. If the INTTESTENB bit in the MDIO control register is set to 1, the host may set the MAC0 bit to 1 for test purposes.
		NO	0	No MDIO link change event.
		YES	1	An MDIO link change event (change in the MDIO PHY link status register) corresponding to the PHY address in MDIO user PHY select register 0 (USERPHYSEL0) and the LINKINTENB bit in USERPHYSEL0 is set to 1.

† For CSL implementation, use the notation MDIO_LINKINTMASKED_*field_symval*

B.9.7 MDIO User Command Complete Interrupt Register (USERINTRAW)

The MDIO user command complete interrupt register (USERINTRAW) is shown in Figure B–104 and described in Table B–112.

Figure B–104. MDIO User Command Complete Interrupt Register (USERINTRAW)

31	2	1	0
Reserved		MAC1	MAC0
R-0		R/WC-0	R/WC-0

Legend: R = Read only; WC = Write 1 to clear, write of 0 has no effect; -*n* = value after reset

Table B–112. MDIO User Command Complete Interrupt Register (USERINTRAW) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31–2	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
1	MAC1			MDIO user command complete event bit. Writing a 1 clears the event and writing a 0 has no effect. If the INTTESTENB bit in the MDIO control register is set to 1, the host may set the MAC1 bit to 1 for test purposes.
		NO	0	No MDIO user command complete event.
		YES	1	The previously scheduled PHY read or write command using MDIO user access register 1 (USERACCESS1) has completed.
0	MAC0			MDIO user command complete event bit. Writing a 1 clears the event and writing a 0 has no effect. If the INTTESTENB bit in the MDIO control register is set to 1, the host may set the MAC0 bit to 1 for test purposes.
		NO	0	No MDIO user command complete event.
		YES	1	The previously scheduled PHY read or write command using MDIO user access register 0 (USERACCESS0) has completed.

† For CSL implementation, use the notation MDIO_USERINTRAW_*field_symval*

B.9.8 MDIO User Command Complete Interrupt (Masked) Register (USERINTMASKED)

The MDIO user command complete interrupt (masked) register (USERINTMASKED) is shown in Figure B–105 and described in Table B–113.

Figure B–105. MDIO User Command Complete Interrupt (Masked) Register (USERINTMASKED)

31	2	1	0
Reserved		MAC1	MAC0
R-0		R/WC-0	R/WC-0

Legend: R = Read only; WC = Write 1 to clear, write of 0 has no effect; -*n* = value after reset

Table B–113. MDIO User Command Complete Interrupt (Masked) Register (USERINTMASKED) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31–2	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
1	MAC1			MDIO user command complete interrupt bit. Writing a 1 clears the interrupt and writing a 0 has no effect. If the INTTESTENB bit in the MDIO control register is set to 1, the host may set the MAC1 bit to 1 for test purposes.
		NO	0	No MDIO user command complete event.
		YES	1	The previously scheduled PHY read or write command using MDIO user access register 1 (USERACCESS1) has completed and the MAC1 bit in USERINTMASKSET is set to 1.
0	MAC0			MDIO user command complete interrupt bit. Writing a 1 clears the interrupt and writing a 0 has no effect. If the INTTESTENB bit in the MDIO control register is set to 1, the host may set the MAC0 bit to 1 for test purposes.
		NO	0	No MDIO user command complete event.
		YES	1	The previously scheduled PHY read or write command using MDIO user access register 0 (USERACCESS0) has completed and the MAC0 bit in USERINTMASKSET is set to 1.

† For CSL implementation, use the notation MDIO_USERINTMASKED_*field_symval*

B.9.9 MDIO User Command Complete Interrupt Mask Set Register (USERINTMASKSET)

The MDIO user command complete interrupt mask set register (USERINTMASKSET) is shown in Figure B–106 and described in Table B–114.

Figure B–106. MDIO User Command Complete Interrupt Mask Set Register (USERINTMASKSET)

31	2	1	0
Reserved		MAC1	MAC0
R-0		R/WS-0	R/WS-0

Legend: R = Read only; WS = Write 1 to set, write of 0 has no effect; -*n* = value after reset

Table B–114. MDIO User Command Complete Interrupt Mask Set Register (USERINTMASKSET) Field Descriptions

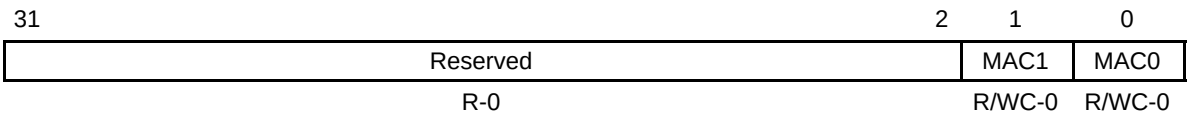
Bit	*field*†	*symval*†	Value	Description
31–2	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
1	MAC1			MDIO user command complete interrupt mask set bit for MAC1 in USERINTMASKED. Writing a 1 sets the bit and writing a 0 has no effect.
		NO	0	MDIO user command complete interrupts for the MDIO user access register 1 (USERACCESS1) are disabled.
		YES	1	MDIO user command complete interrupts for the MDIO user access register 1 (USERACCESS1) are enabled.
0	MAC0			MDIO user command complete interrupt mask set bit for MAC0 in USERINTMASKED. Writing a 1 sets the bit and writing a 0 has no effect.
		NO	0	MDIO user command complete interrupts for the MDIO user access register 0 (USERACCESS0) are disabled.
		YES	1	MDIO user command complete interrupts for the MDIO user access register 0 (USERACCESS0) are enabled.

† For CSL implementation, use the notation MDIO_USERINTMASKSET_*field_symval*

B.9.10 MDIO User Command Complete Interrupt Mask Clear Register (USERINTMASKCLEAR)

The MDIO user command complete interrupt mask clear register (USERINTMASKCLEAR) is shown in Figure B–107 and described in Table B–115.

Figure B–107. MDIO User Command Complete Interrupt Mask Clear Register (USERINTMASKCLEAR)



Legend: R = Read only; WC = Write 1 to clear, write of 0 has no effect; -*n* = value after reset

Table B–115. MDIO User Command Complete Interrupt Mask Clear Register (USERINTMASKCLEAR) Field Descriptions

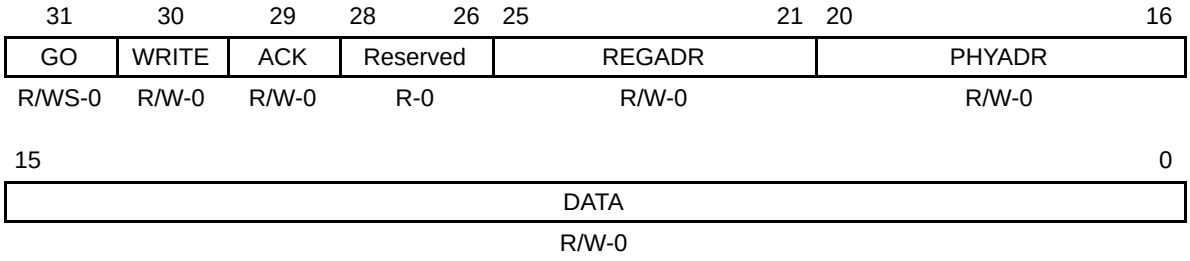
Bit	*field*†	*symval*†	Value	Description
31–2	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
1	MAC1			MDIO user command complete interrupt mask clear bit for MAC1 in USERINTMASKED. Writing a 1 clears the bit and writing a 0 has no effect.
		NO	0	MDIO user command complete interrupts for the MDIO user access register 1 (USERACCESS1) are enabled.
		YES	1	MDIO user command complete interrupts for the MDIO user access register 1 (USERACCESS1) are disabled.
0	MAC0			MDIO user command complete interrupt mask clear bit for MAC0 in USERINTMASKED. Writing a 1 clears the bit and writing a 0 has no effect.
		NO	0	MDIO user command complete interrupts for the MDIO user access register 0 (USERACCESS0) are enabled.
		YES	1	MDIO user command complete interrupts for the MDIO user access register 0 (USERACCESS0) are disabled.

† For CSL implementation, use the notation MDIO_USERINTMASKCLEAR_*field_symval*

B.9.11 MDIO User Access Register 0 (USERACCESS0)

The MDIO user access register 0 (USERACCESS0) is shown in Figure B–108 and described in Table B–116.

Figure B–108. MDIO User Access Register 0 (USERACCESS0)



Legend: R = Read only; R/W = Read/Write; WS = Write 1 to set, write of 0 has no effect; -*n* = value after reset

Table B–116. MDIO User Access Register 0 (USERACCESS0) Field Descriptions

Bit	***field***†	***symval***†	Value	Description
31	GO			GO bit is writable only if the MDIO state machine is enabled (ENABLE bit in MDIO control register is set to 1). If byte access is being used, the GO bit should be written last. Writing a 1 sets the bit and writing a 0 has no effect.
			0	No effect. The GO bit clears when the requested access has been completed.
			1	The MDIO state machine performs an MDIO access when it is convenient, this is not an instantaneous process. Any writes to USERACCESS0 are blocked.
30	WRITE			Write enable bit determines the MDIO transaction type.
			0	MDIO transaction is a register read.
			1	MDIO transaction is a register write.
29	ACK			Acknowledge bit determines if the PHY acknowledges the read transaction.
			0	No acknowledge.
			1	PHY acknowledges the read transaction.
28–26	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.

† For CSL implementation, use the notation MDIO_USERACCESS0_*field_symval*

Table B–116. MDIO User Access Register 0 (USERACCESS0) Field Descriptions (Continued)

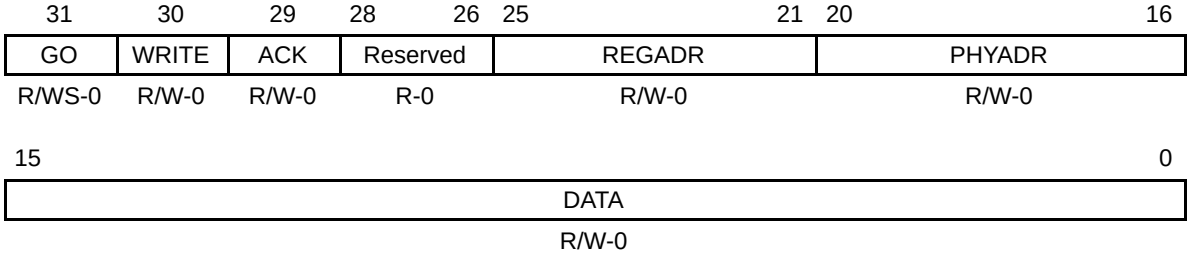
Bit	*field*†	*symval*†	Value	Description
25–21	REGADR		0–1Fh	Register address bits specify the PHY register to be accessed for this transaction.
20–16	PHYADR		0–1Fh	PHY address bits specify the PHY to be accessed for this transaction.
15–0	DATA		0–FFFFh	User data bits specify the data value read from or to be written to the specified PHY register.

† For CSL implementation, use the notation MDIO_USERACCESS0_*field_symval*

B.9.12 MDIO User Access Register 1 (USERACCESS1)

The MDIO user access register 1 (USERACCESS1) is shown in Figure B–109 and described in Table B–117.

Figure B–109. MDIO User Access Register 1 (USERACCESS1)



Legend: R = Read only; R/W = Read/Write; WS = Write 1 to set, write of 0 has no effect; -*n* = value after reset

Table B–117. MDIO User Access Register 1 (USERACCESS1) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31	GO			GO bit is writable only if the MDIO state machine is enabled (ENABLE bit in MDIO control register is set to 1). If byte access is being used, the GO bit should be written last. Writing a 1 sets the bit and writing a 0 has no effect.
			0	No effect. The GO bit clears when the requested access has been completed.
			1	The MDIO state machine performs an MDIO access when it is convenient, this is not an instantaneous process. Any writes to USERACCESS1 are blocked.
30	WRITE			Write enable bit determines the MDIO transaction type.
			0	MDIO transaction is a register read.
			1	MDIO transaction is a register write.
29	ACK			Acknowledge bit determines if the PHY acknowledges the read transaction.
			0	No acknowledge.
			1	PHY acknowledges the read transaction.
28–26	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.

† For CSL implementation, use the notation MDIO_USERACCESS1_*field_symval*

Table B–117. MDIO User Access Register 1 (USERACCESS1) Field Descriptions (Continued)

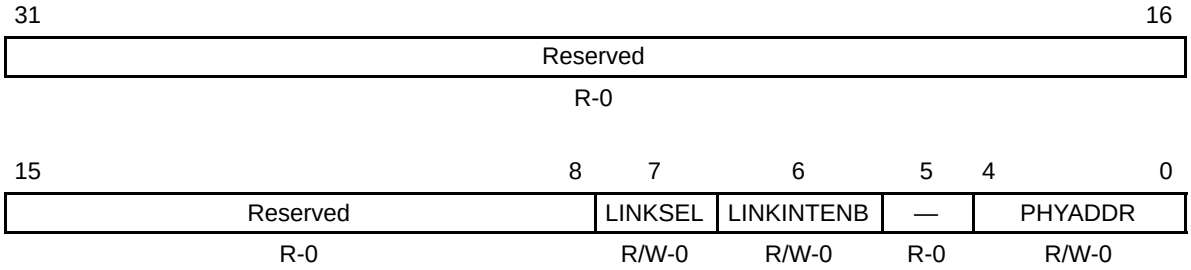
Bit	*field*†	*symval*†	Value	Description
25–21	REGADR		0–1Fh	Register address bits specify the PHY register to be accessed for this transaction.
20–16	PHYADR		0–1Fh	PHY address bits specify the PHY to be accessed for this transaction.
15–0	DATA		0–FFFFh	User data bits specify the data value read from or to be written to the specified PHY register.

† For CSL implementation, use the notation MDIO_USERACCESS1_*field_symval*

B.9.13 MDIO User PHY Select Register 0 (USERPHYSEL0)

The MDIO user PHY select register 0 (USERPHYSEL0) is shown in Figure B–110 and described in Table B–118.

Figure B–110. MDIO User PHY Select Register 0 (USERPHYSEL0)



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset

Table B–118. MDIO User PHY Select Register 0 (USERPHYSEL0) Field Descriptions

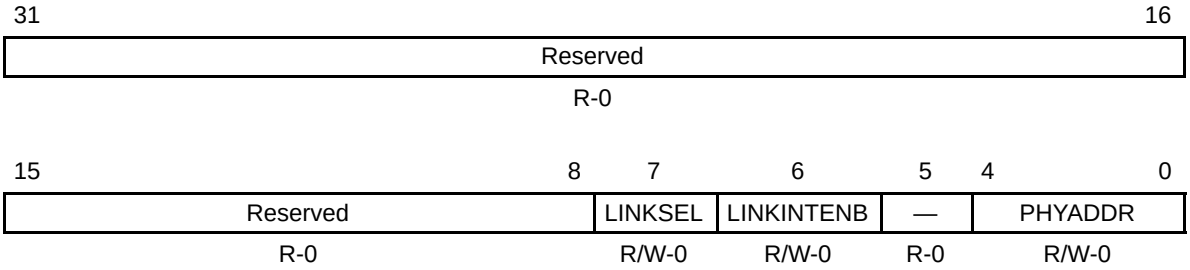
Bit	*field*†	*symval*†	Value	Description
31–8	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
7	LINKSEL			Link status determination select bit.
		MDIO	0	Link status is determined by the MDIO state machine.
			1	Value must be set to MDIO.
6	LINKINTENB			Link change interrupt enable bit.
		DISABLE	0	Link change interrupts are disabled.
		ENABLE	1	Link change status interrupts for PHY address specified in PHYADDR bits are enabled.
5	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
4–0	PHYADDR		0–1Fh	PHY address bits specify the PHY address to be monitored.

† For CSL implementation, use the notation MDIO_USERPHYSEL0_*field_symval*

B.9.14 MDIO User PHY Select Register 1 (USERPHYSEL1)

The MDIO user PHY select register 1 (USERPHYSEL1) is shown in Figure B–111 and described in Table B–119.

Figure B–111. MDIO User PHY Select Register 1 (USERPHYSEL1)



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset

Table B–119. MDIO User PHY Select Register 1 (USERPHYSEL1) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31–8	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
7	LINKSEL			Link status determination select bit.
		MDIO	0	Link status is determined by the MDIO state machine.
			1	Value must be set to MDIO.
6	LINKINTENB			Link change interrupt enable bit.
		DISABLE	0	Link change interrupts are disabled.
		ENABLE	1	Link change status interrupts for PHY address specified in PHYADDR bits are enabled.
5	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
4–0	PHYADDR		0–1Fh	PHY address bits specify the PHY address to be monitored.

† For CSL implementation, use the notation MDIO_USERPHYSEL1_*field_symval*

B.10 Multichannel Audio Serial Port (McASP) Registers

Table B–120. McASP Registers Accessed Through Configuration Bus

Acronym	Register Name	Address Offset (hex)	Section
PID	Peripheral identification register	0000	B.10.1
PWRDEMU	Power down and emulation management register	0004	B.10.2
PFUNC	Pin function register	0010	B.10.3
PDIR	Pin direction register	0014	B.10.4
PDOUT	Pin data output register	0018	B.10.5
PDIN	Read returns: Pin data input register	001C	B.10.6
PDSET	Writes affect: Pin data set register (alternate write address: PDOUT)	001C	B.10.7
PDCLR	Pin data clear register (alternate write address: PDOUT)	0020	B.10.8
GBLCTL	Global control register	0044	B.10.9
AMUTE	Audio mute control register	0048	B.10.10
DLBCTL	Digital loopback control register	004C	B.10.11
DITCTL	DIT mode control register	0050	B.10.12
RGBLCTL	Receiver global control register. Alias of GBLCTL, only receive bits are affected – allows receiver to be reset independently from transmitter	0060	B.10.13
RMASK	Receive format unit bit mask register	0064	B.10.14
RFMT	Receive bit stream format register	0068	B.10.15
AFSRCTL	Receive frame sync control register	006C	B.10.16
ACLKRCTL	Receive clock control register	0070	B.10.17
AHCLKRCTL	Receive high-frequency clock control register	0074	B.10.18
RTDM	Receive TDM time slot 0–31 register	0078	B.10.19
RINTCTL	Receiver interrupt control register	007C	B.10.20
RSTAT	Receiver status register	0080	B.10.21
RSLOT	Current receive TDM time slot register	0084	B.10.22
RCLKCHK	Receive clock check control register	0088	B.10.23

† Available only on DA6x DSP.
‡ CFG BUS only if XBUSEL = 1.
§ CFG BUS only if RBUSEL = 1.

Table B–120. McASP Registers Accessed Through Configuration Bus (Continued)

Acronym	Register Name	Address Offset (hex)	Section
REVCTL†	Receiver DMA event control register	008C	B.10.24
XGBLCTL	Transmitter global control register. Alias of GBLCTL, only transmit bits are affected– allows transmitter to be reset independently from receiver	00A0	B.10.25
XMASK	Transmit format unit bit mask register	00A4	B.10.26
XFMT	Transmit bit stream format register	00A8	B.10.27
AFSXCTL	Transmit frame sync control register	00AC	B.10.28
ACLKXCTL	Transmit clock control register	00B0	B.10.29
AHCLKXCTL	Transmit high-frequency clock control register	00B4	B.10.30
XTDM	Transmit TDM time slot 0–31 register	00B8	B.10.31
XINTCTL	Transmitter interrupt control register	00BC	B.10.32
XSTAT	Transmitter status register	00C0	B.10.33
XSLOT	Current transmit TDM time slot register	00C4	B.10.34
XCLKCHK	Transmit clock check control register	00C8	B.10.35
XEVCTL†	Transmitter DMA event control register	00CC	B.10.36
DITCSRA0	Left (even TDM time slot) channel status register (DIT mode) 0	0100	B.10.38
DITCSRA1	Left (even TDM time slot) channel status register (DIT mode) 1	0104	B.10.38
DITCSRA2	Left (even TDM time slot) channel status register (DIT mode) 2	0108	B.10.38
DITCSRA3	Left (even TDM time slot) channel status register (DIT mode) 3	010C	B.10.38
DITCSRA4	Left (even TDM time slot) channel status register (DIT mode) 4	0110	B.10.38
DITCSRA5	Left (even TDM time slot) channel status register (DIT mode) 5	0114	B.10.38
DITCSRB0	Right (odd TDM time slot) channel status register (DIT mode) 0	0118	B.10.39
DITCSRB1	Right (odd TDM time slot) channel status register (DIT mode) 1	011C	B.10.39
DITCSRB2	Right (odd TDM time slot) channel status register (DIT mode) 2	0120	B.10.39
DITCSRB3	Right (odd TDM time slot) channel status register (DIT mode) 3	0124	B.10.39
DITCSRB4	Right (odd TDM time slot) channel status register (DIT mode) 4	0128	B.10.39

† Available only on DA6x DSP.
‡ CFG BUS only if XBUSEL = 1.
§ CFG BUS only if RBUSEL = 1.

Table B–120. McASP Registers Accessed Through Configuration Bus (Continued)

Acronym	Register Name	Address Offset (hex)	Section
DITCSRB5	Right (odd TDM time slot) channel status register (DIT mode) 5	012C	B.10.39
DITUDRA0	Left (even TDM time slot) channel user data register (DIT mode) 0	0130	B.10.40
DITUDRA1	Left (even TDM time slot) channel user data register (DIT mode) 1	0134	B.10.40
DITUDRA2	Left (even TDM time slot) channel user data register (DIT mode) 2	0138	B.10.40
DITUDRA3	Left (even TDM time slot) channel user data register (DIT mode) 3	013C	B.10.40
DITUDRA4	Left (even TDM time slot) channel user data register (DIT mode) 4	0140	B.10.40
DITUDRA5	Left (even TDM time slot) channel user data register (DIT mode) 5	0144	B.10.40
DITUDRB0	Right (odd TDM time slot) channel user data register (DIT mode) 0	0148	B.10.41
DITUDRB1	Right (odd TDM time slot) channel user data register (DIT mode) 1	014C	B.10.41
DITUDRB2	Right (odd TDM time slot) channel user data register (DIT mode) 2	0150	B.10.41
DITUDRB3	Right (odd TDM time slot) channel user data register (DIT mode) 3	0154	B.10.41
DITUDRB4	Right (odd TDM time slot) channel user data register (DIT mode) 4	0158	B.10.41
DITUDRB5	Right (odd TDM time slot) channel user data register (DIT mode) 5	015C	B.10.41
SRCTL0	Serializer control register 0	0180	B.10.37
SRCTL1	Serializer control register 1	0184	B.10.37
SRCTL2	Serializer control register 2	0188	B.10.37
SRCTL3	Serializer control register 3	018C	B.10.37
SRCTL4	Serializer control register 4	0190	B.10.37
SRCTL5	Serializer control register 5	0194	B.10.37
SRCTL6	Serializer control register 6	0198	B.10.37
SRCTL7	Serializer control register 7	019C	B.10.37
SRCTL8†	Serializer control register 8	01A0	B.10.37
SRCTL9†	Serializer control register 9	01A4	B.10.37
SRCTL10†	Serializer control register 10	01A8	B.10.37
SRCTL11†	Serializer control register 11	01AC	B.10.37

† Available only on DA6x DSP.
‡ CFG BUS only if XBUSEL = 1.
§ CFG BUS only if RBUSEL = 1.

Table B–120. McASP Registers Accessed Through Configuration Bus (Continued)

Acronym	Register Name	Address Offset (hex)	Section
SRCTL12†	Serializer control register 12	01B0	B.10.37
SRCTL13†	Serializer control register 13	01B4	B.10.37
SRCTL14†	Serializer control register 14	01B8	B.10.37
SRCTL15†	Serializer control register 15	01BC	B.10.37
XBUF0‡	Transmit buffer register for serializer 0	0200	B.10.42
XBUF1‡	Transmit buffer register for serializer 1	0204	B.10.42
XBUF2‡	Transmit buffer register for serializer 2	0208	B.10.42
XBUF3‡	Transmit buffer register for serializer 3	020C	B.10.42
XBUF4‡	Transmit buffer register for serializer 4	0210	B.10.42
XBUF5‡	Transmit buffer register for serializer 5	0214	B.10.42
XBUF6‡	Transmit buffer register for serializer 6	0218	B.10.42
XBUF7‡	Transmit buffer register for serializer 7	021C	B.10.42
XBUF8†‡	Transmit buffer register for serializer 8	0220	B.10.42
XBUF9†‡	Transmit buffer register for serializer 9	0224	B.10.42
XBUF10†‡	Transmit buffer register for serializer 10	0228	B.10.42
XBUF11†‡	Transmit buffer register for serializer 11	022C	B.10.42
XBUF12†‡	Transmit buffer register for serializer 12	0230	B.10.42
XBUF13†‡	Transmit buffer register for serializer 13	0234	B.10.42
XBUF14†‡	Transmit buffer register for serializer 14	0238	B.10.42
XBUF15†‡	Transmit buffer register for serializer 15	023C	B.10.42
RBUF0§	Receive buffer register for serializer 0	0280	B.10.43
RBUF1§	Receive buffer register for serializer 1	0284	B.10.43
RBUF2§	Receive buffer register for serializer 2	0288	B.10.43
RBUF3§	Receive buffer register for serializer 3	028C	B.10.43
RBUF4§	Receive buffer register for serializer 4	0290	B.10.43

† Available only on DA6x DSP.
‡ CFG BUS only if XBUSEL = 1.
§ CFG BUS only if RBUSEL = 1.

Table B–120. McASP Registers Accessed Through Configuration Bus (Continued)

Acronym	Register Name	Address Offset (hex)	Section
RBUF5§	Receive buffer register for serializer 5	0294	B.10.43
RBUF6§	Receive buffer register for serializer 6	0298	B.10.43
RBUF7§	Receive buffer register for serializer 7	029C	B.10.43
RBUF8†§	Receive buffer register for serializer 8	02A0	B.10.43
RBUF9†§	Receive buffer register for serializer 9	02A4	B.10.43
RBUF10†§	Receive buffer register for serializer 10	02A8	B.10.43
RBUF11†§	Receive buffer register for serializer 11	02AC	B.10.43
RBUF12†§	Receive buffer register for serializer 12	02B0	B.10.43
RBUF13†§	Receive buffer register for serializer 13	02B4	B.10.43
RBUF14†§	Receive buffer register for serializer 14	02B8	B.10.43
RBUF15†§	Receive buffer register for serializer 15	02BC	B.10.43

† Available only on DA6x DSP.
‡ CFG BUS only if XBUSEL = 1.
§ CFG BUS only if RBUSEL = 1.

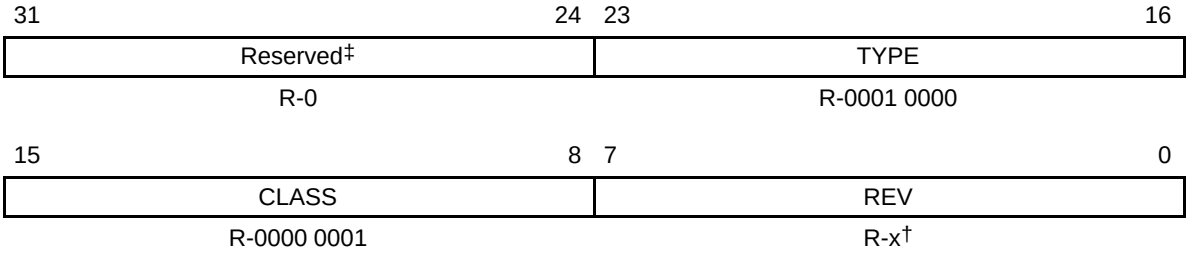
Table B–121. McASP Registers Accessed Through Data Port

Hex Address	Register Name	Register Description
Read Accesses	RBUF	Receive buffer data port address. Cycles through receive serializers, skipping over transmit serializers and inactive serializers. Starts at the lowest serializer at the beginning of each time slot. DAT BUS only if XBUSEL = 0.
Write Accesses	XBUF	Transmit buffer data port address. Cycles through transmit serializers, skipping over receive and inactive serializers. Starts at the lowest serializer at the beginning of each time slot. DAT BUS only if RBUSEL = 0.

B.10.1 Peripheral Identification Register (PID)

The peripheral identification register (PID) is shown in Figure B–112 and described in Table B–122.

Figure B–112. Peripheral Identification Register (PID) [Offset 0000h]



Legend: R = Read only; -*n* = value after reset
† See the device-specific datasheet for the default value of this field.
‡ If writing to this field, always write the default value for future device compatibility.

Table B–122. Peripheral Identification Register (PID) Field Descriptions

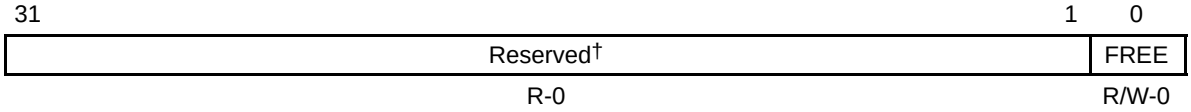
Bit	***field***†	***symval***†	Value	Description
31–24	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
23–16	TYPE			Identifies type of peripheral.
		MCASP	10h	McASP
15–8	CLASS			Identifies class of peripheral.
		SERPORT	1	Serial port
7–0	REV			Identifies revision of peripheral.
		–	x	See the device-specific datasheet for the value.

† For CSL implementation, use the notation MCASP_PID_*field_symval*

B.10.2 Power Down and Emulation Management Register (PWRDEMU)

The power down and emulation management register (PWRDEMU) is shown in Figure B–113 and described in Table B–123.

Figure B–113. Power Down and Emulation Management Register (PWRDEMU) [Offset 0004h]



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.

Table B–123. Power Down and Emulation Management Register (PWRDEMU) Field Descriptions

Bit	Field	*symval*†	Value	Description
31–1	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
0	FREE			Free-running mode enable bit. This bit determines the state of the serial port clock during emulation halt.
		OFF	0	Reserved.
		ON	1	Free-running mode is enabled. Peripheral ignores the emulation suspend signal and continues to function as normal. During emulation suspend, EDMA requests continue to be generated and are serviced by the EDMA. Error conditions are flagged as usual.

† For CSL implementation, use the notation MCASP_PWRDEMU_FREE_*symval*

B.10.3 Pin Function Register (PFUNC)

The pin function register (PFUNC) specifies the function of AXR[n], ACLKX, AHCLKX, AFSX, ACLKR, AHCLKR, and AFSR pins as either a McASP pin or a general-purpose input/output (GPIO) pin. The PFUNC is shown in Figure B–114 and described in Table B–124.

Writing to Reserved Bits

Writing a value other than 0 to reserved bits in this register may cause improper device operation. This includes bits that are not implemented on a particular DSP.

Figure B–114. Pin Function Register (PFUNC) [Offset 0010h]

31	30	29	28	27	26	25	24
AFSR	AHCLKR	ACLKR	AFSX	AHCLKX	ACLKX	AMUTE	Reserved†
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-0

23	16
Reserved†	
R-0	

15	14	13	12	11	10	9	8
AXR15‡	AXR14‡	AXR13‡	AXR12‡	AXR11‡	AXR10‡	AXR9‡	AXR8‡
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

7	6	5	4	3	2	1	0
AXR7	AXR6	AXR5	AXR4	AXR3	AXR2	AXR1	AXR0
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.
‡ On DA6x DSP only; reserved on C6713 DSP.

Table B–124. Pin Function Register (PFUNC) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31	AFSR			Determines if specified pin functions as McASP or GPIO.
30	AHCLKR	MCASP	0	Pin functions as McASP pin.
29	ACLKR	GPIO	1	Pin functions as GPIO pin.
28	AFSX			
27	AHCLKX			
26	ACLKX			
25	AMUTE			
24–16	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
15–8	AXR[15–8]‡			Determines if AXR[n] pin functions as McASP or GPIO.
		MCASP	0	Pin functions as McASP pin.
		GPIO	1	Pin functions as GPIO pin.
7–0	AXR[7–0]			Determines if AXR[n] pin functions as McASP or GPIO.
		MCASP	0	Pin functions as McASP pin.
		GPIO	1	Pin functions as GPIO pin.

† For CSL implementation, use the notation MCASP_PFUNC_*field_symval*
‡ On DA6x DSP only; reserved on C6713 DSP.

B.10.4 Pin Direction Register (PDIR)

The pin direction register (PDIR) specifies the direction of AXR[n], ACLKX, AHCLKX, AFSX, ACLKR, AHCLKR, and AFSR pins as either an input or an output pin. The PDIR is shown in Figure B–115 and described in Table B–125.

Regardless of the pin function register (PFUNC) setting, each PDIR bit must be set to 1 for the specified pin to be enabled as an output and each PDIR bit must be cleared to 0 for the specified pin to be an input.

For example, if the McASP is configured to use an internally-generated bit clock and the clock is to be driven out to the system, the PFUNC bit must be cleared to 0 (McASP function) and the PDIR bit must be set to 1 (an output).

When AXR[n] is configured to transmit, the PFUNC bit must be cleared to 0 (McASP function) and the PDIR bit must be set to 1 (an output). Similarly, when AXR[n] is configured to receive, the PFUNC bit must be cleared to 0 (McASP function) and the PDIR bit must be cleared to 0 (an input).

Writing to Reserved Bits

Writing a value other than 0 to reserved bits in this register may cause improper device operation. This includes bits that are not implemented on a particular DSP.

Figure B–115. Pin Direction Register (PDIR) [Offset 0014h]

31	30	29	28	27	26	25	24
AFSR	AHCLKR	ACLKR	AFSX	AHCLKX	ACLKX	AMUTE	Reserved†
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-0

23	16
Reserved†	
R-0	

15	14	13	12	11	10	9	8
AXR15‡	AXR14‡	AXR13‡	AXR12‡	AXR11‡	AXR10‡	AXR9‡	AXR8‡
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

7	6	5	4	3	2	1	0
AXR7	AXR6	AXR5	AXR4	AXR3	AXR2	AXR1	AXR0
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.
‡ On DA6x DSP only; reserved on C6713 DSP.

Table B–125. Pin Direction Register (PDIR) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31	AFSR			Determines if specified pin functions as an input or output.
30	AHCLKR	IN	0	Pin functions as input.
29	ACLKR	OUT	1	Pin functions as output.
28	AFSX			
27	AHCLKX			
26	ACLKX			
25	AMUTE			
24–16	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
15–8	AXR[15–8]‡			Determines if AXR[n] pin functions as an input or output.
		IN	0	Pin functions as input.
		OUT	1	Pin functions as output.
7–0	AXR[7–0]			Determines if AXR[n] pin functions as an input or output.
		IN	0	Pin functions as input.
		OUT	1	Pin functions as output.

† For CSL implementation, use the notation MCASP_PDIR_*field_symval*
‡ On DA6x DSP only; reserved on C6713 DSP.

B.10.5 Pin Data Output Register (PDOUT)

The pin data output register (PDOUT) holds a value for data out at all times, and may be read back at all times. The value held by PDOUT is not affected by writing to PDIR and PFUNC. However, the data value in PDOUT is driven out onto the McASP pin only if the corresponding bit in PFUNC is set to 1 (GPIO function) and the corresponding bit in PDIR is set to 1 (output). The PDOUT is shown in Figure B–116 and described in Table B–126.

PDOUT has these aliases or alternate addresses:

- PDSET — when written to at this address, writing a 1 to a bit in PDSET sets the corresponding bit in PDOUT to 1; writing a 0 has no effect and keeps the bits in PDOUT unchanged.
- PDCLR — when written to at this address, writing a 1 to a bit in PDCLR clears the corresponding bit in PDOUT to 0; writing a 0 has no effect and keeps the bits in PDOUT unchanged.

There is only one set of data out bits, PDOUT[31–0]. The other registers, PDSET and PDCLR, are just different addresses for the same control bits, with different behaviors during writes.

Writing to Reserved Bits

Writing a value other than 0 to reserved bits in this register may cause improper device operation. This includes bits that are not implemented on a particular DSP.

Figure B–116. Pin Data Output Register (PDOUT) [Offset 0018h]

31	30	29	28	27	26	25	24
AFSR	AHCLKR	ACLKR	AFSX	AHCLKX	ACLKX	AMUTE	Reserved†
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-0

23	16
Reserved†	
R-0	

15	14	13	12	11	10	9	8
AXR15‡	AXR14‡	AXR13‡	AXR12‡	AXR11‡	AXR10‡	AXR9‡	AXR8‡
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

7	6	5	4	3	2	1	0
AXR7	AXR6	AXR5	AXR4	AXR3	AXR2	AXR1	AXR0
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.
‡ On DA6x DSP only; reserved on C6713 DSP.

Table B–126. Pin Data Output Register (PDOUT) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31	AFSR			Determines drive on specified output pin when the corresponding PFUNC[n] and PDIR[n] bits are set to 1. When reading data, returns the corresponding bit value in PDOUT[n], does not return input from I/O pin. When writing data, writes to the corresponding PDOUT[n] bit.
30	AHCLKR			
29	ACLKR			
28	AFSX			
27	AHCLKX	LOW	0	Pin drives low.
26	ACLKX	HIGH	1	Pin drives high.
25	AMUTE			
24–16	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
15–8	AXR[15–8]‡			Determines drive on AXR[n] pin when PFUNC[n] and PDIR[n] bits are set to 1. When reading data, returns the bit value in PDOUT[n], does not return input from I/O pin. When writing data, writes to PDOUT[n] bit.
		LOW	0	Pin drives low.
		HIGH	1	Pin drives high.
7–0	AXR[7–0]			Determines drive on AXR[n] pin when PFUNC[n] and PDIR[n] bits are set to 1. When reading data, returns the bit value in PDOUT[n], does not return input from I/O pin. When writing data, writes to PDOUT[n] bit.
		LOW	0	Pin drives low.
		HIGH	1	Pin drives high.

† For CSL implementation, use the notation MCASP_PDOUT_*field_symval*
‡ On DA6x DSP only; reserved on C6713 DSP.

B.10.6 Pin Data Input Register (PDIN)

The pin data input register (PDIN) holds the I/O pin state of each of the McASP pins. PDIN allows the actual value of the pin to be read, regardless of the state of PFUNC and PDIR. The value after reset for registers 1 through 15 and 24 through 31 depends on how the pins are being driven. The PDIN is shown in Figure B–117 and described in Table B–127.

Writing to Reserved Bits

Writing a value other than 0 to reserved bits in this register may cause improper device operation. This includes bits that are not implemented on a particular DSP.

Figure B–117. Pin Data Input Register (PDIN) [Offset 001Ch]

31	30	29	28	27	26	25	24
AFSR	AHCLKR	ACLKR	AFSX	AHCLKX	ACLKX	AMUTE	Reserved†
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-0

23	16
Reserved†	
R-0	

15	14	13	12	11	10	9	8
AXR15‡	AXR14‡	AXR13‡	AXR12‡	AXR11‡	AXR10‡	AXR9‡	AXR8‡
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

7	6	5	4	3	2	1	0
AXR7	AXR6	AXR5	AXR4	AXR3	AXR2	AXR1	AXR0
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.
‡ On DA6x DSP only; reserved on C6713 DSP.

Table B–127. Pin Data Input Register (PDIN) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31	AFSR			Provides logic level of the specified pin.
30	AHCLKR		0	Pin is logic low.
29	ACLKR	SET	1	Pin is logic high.
28	AFSX			
27	AHCLKX			
26	ACLKX			
25	AMUTE			
24–16	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
15–8	AXR[15–8]‡			Provides logic level of AXR[n] pin.
			0	Pin is logic low.
		SET	1	Pin is logic high.
7–0	AXR[7–0]			Provides logic level of AXR[n] pin.
			0	Pin is logic low.
		SET	1	Pin is logic high.

† For CSL implementation, use the notation MCASP_PDIN_*field_symval*
‡ On DA6x DSP only; reserved on C6713 DSP.

B.10.7 Pin Data Set Register (PDSET)

The pin data set register (PDSET) is an alias of the pin data output register (PDOUT) for writes only. Writing a 1 to the PDSET bit sets the corresponding bit in PDOUT and, if PFUNC = 1 (GPIO function) and PDIR = 1 (output), drives a logic high on the pin. PDSET is useful for a multitasking system because it allows you to set to a logic high only the desired pin(s) within a system without affecting other I/O pins controlled by the same McASP. The PDSET is shown in Figure B–118 and described in Table B–128.

Writing to Reserved Bits

Writing a value other than 0 to reserved bits in this register may cause improper device operation. This includes bits that are not implemented on a particular DSP.

Figure B–118. Pin Data Set Register (PDSET) [Offset 001Ch]

31	30	29	28	27	26	25	24
AFSR	AHCLKR	ACLKR	AFSX	AHCLKX	ACLKX	AMUTE	Reserved†
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-0

23	16
Reserved†	
R-0	

15	14	13	12	11	10	9	8
AXR15†	AXR14†	AXR13†	AXR12†	AXR11†	AXR10†	AXR9†	AXR8†
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

7	6	5	4	3	2	1	0
AXR7	AXR6	AXR5	AXR4	AXR3	AXR2	AXR1	AXR0
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.
† On DA6x DSP only; reserved on C6713 DSP.

Table B–128. Pin Data Set Register (PDSET) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31	AFSR			Allows the corresponding PDOUT[n] bit to be set to a logic high without affecting other I/O pins controlled by the same port.
30	AHCLKR			
29	ACLKR		0	No effect.
28	AFSX	SET	1	Sets corresponding PDOUT[n] bit to 1.
27	AHCLKX			
26	ACLKX			
25	AMUTE			
24–16	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
15–8	AXR[15–8]‡			Allows PDOUT[n] bit to be set to a logic high without affecting other I/O pins controlled by the same port.
			0	No effect.
		SET	1	Sets PDOUT[n] bit to 1.
7–0	AXR[7–0]			Allows PDOUT[n] bit to be set to a logic high without affecting other I/O pins controlled by the same port.
			0	No effect.
		SET	1	Sets PDOUT[n] bit to 1.

† For CSL implementation, use the notation MCASP_PDSET_*field_symval*
‡ On DA6x DSP only; reserved on C6713 DSP.

B.10.8 Pin Data Clear Register (PDCLR)

The pin data clear register (PDCLR) is an alias of the pin data output register (PDOUT) for writes only. Writing a 1 to the PDCLR bit clears the corresponding bit in PDOUT and, if PFUNC = 1 (GPIO function) and PDIR = 1 (output), drives a logic low on the pin. PDCLR is useful for a multitasking system because it allows you to clear to a logic low only the desired pin(s) within a system without affecting other I/O pins controlled by the same McASP. The PDCLR is shown in Figure B–119 and described in Table B–129.

Writing to Reserved Bits

Writing a value other than 0 to reserved bits in this register may cause improper device operation. This includes bits that are not implemented on a particular DSP.

Figure B–119. PDCLR Pin Data Clear Register (PDCLR) [Offset 0020h]

31	30	29	28	27	26	25	24
AFSR	AHCLKR	ACLKR	AFSX	AHCLKX	ACLKX	AMUTE	Reserved†
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-0

23	16
Reserved†	
R-0	

15	14	13	12	11	10	9	8
AXR15‡	AXR14‡	AXR13‡	AXR12‡	AXR11‡	AXR10‡	AXR9‡	AXR8‡
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

7	6	5	4	3	2	1	0
AXR7	AXR6	AXR5	AXR4	AXR3	AXR2	AXR1	AXR0
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.
‡ On DA6x DSP only; reserved on C6713 DSP.

Table B–129. Pin Data Clear Register (PDCLR) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31	AFSR			Allows the corresponding PDOUT[n] bit to be cleared to a logic low without affecting other I/O pins controlled by the same port.
30	AHCLKR			
29	ACLKR		0	No effect.
28	AFSX	CLR	1	Clears corresponding PDOUT[n] bit to 0.
27	AHCLKX			
26	ACLKX			
25	AMUTE			
24–16	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
15–8	AXR[15–8]‡			Allows PDOUT[n] bit to be cleared to a logic low without affecting other I/O pins controlled by the same port.
			0	No effect.
		CLR	1	Clears PDOUT[n] bit to 0.
7–0	AXR[7–0]			Allows PDOUT[n] bit to be cleared to a logic low without affecting other I/O pins controlled by the same port.
			0	No effect.
		CLR	1	Clears PDOUT[n] bit to 0.

† For CSL implementation, use the notation MCASP_PDCLR_*field_symval*
‡ On DA6x DSP only; reserved on C6713 DSP.

B.10.9 Global Control Register (GBLCTL)

The global control register (GBLCTL) provides initialization of the transmit and receive sections. The GBLCTL is shown in Figure B–120 and described in Table B–130.

The bit fields in GBLCTL are synchronized and latched by the corresponding clocks (ACLKX for bits 12–8 and ACLKR for bits 4–0). Before GBLCTL is programmed, you must ensure that serial clocks are running. If the corresponding external serial clocks, ACLKX and ACLKR, are not yet running, you should select the internal serial clock source in AHCLKXCTL, AHCLKRCTL, ACLKXCTL, and ACLKRCTL before GBLCTL is programmed. Also, after programming any bits in GBLCTL you should not proceed until you have read back from GBLCTL and verified that the bits are latched in GBLCTL.

Figure B–120. Global Control Register (GBLCTL) [Offset 0044h]

31	16
Reserved†	
R-0	

15	13	12	11	10	9	8
Reserved†		XFRST	XSMRST	XSRCLR	XHCLKRST	XCLKRST
R-0		R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

7	5	4	3	2	1	0
Reserved†		RFRST	RSMRST	RSRCLR	RHCLKRST	RCLKRST
R-0		R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.

Table B–130. Global Control Register (GBLCTL) Field Descriptions

Bit	***field***†	***symval***†	Value	Description
31–13	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
12	XFRST			Transmit frame sync generator reset enable bit.
		RESET	0	Transmit frame sync generator is reset.
		ACTIVE	1	Transmit frame sync generator is active. When released from reset, the transmit frame sync generator begins counting serial clocks and generating frame sync as programmed.

† For CSL implementation, use the notation MCASP_GBLCTL_*field_symval*

Table B–130. Global Control Register (GBLCTL) Field Descriptions (Continued)

Bit	*field*†	*symval*†	Value	Description
11	XSMRST			Transmit state machine reset enable bit.
		RESET	0	Transmit state machine is held in reset. AXR[n] pin state: If PFUNC[n] = 0 and PDIR[n] = 1; then the serializer drives the AXR[n] pin to the state specified for inactive time slot (as determined by DISMOD bits in SRCTL).
		ACTIVE	1	Transmit state machine is released from reset. When released from reset, the transmit state machine immediately transfers data from XRBUF[n] to XRSR[n]. The transmit state machine sets the underrun flag (XUNDRN) in XSTAT, if XRBUF[n] have not been preloaded with data before reset is released. The transmit state machine also immediately begins detecting frame sync and is ready to transmit. Transmit TDM time slot begins at slot 0 after reset is released.
10	XSRCLR			Transmit serializer clear enable bit. By clearing then setting this bit, the transmit buffer is flushed to an empty state (XDATA = 1). If XSMRST = 1, XSRCLR = 1, XDATA = 1, and XBUF is not loaded with new data before the start of the next active time slot, an underrun will occur.
		CLEAR	0	Transmit serializers are cleared.
		ACTIVE	1	Transmit serializers are active. When the transmit serializers are first taken out of reset (XSRCLR changes from 0 to 1), the transmit data ready bit (XDATA) in XSTAT is set to indicate XBUF is ready to be written.
9	XHCLKRST			Transmit high-frequency clock divider reset enable bit.
		RESET	0	Transmit high-frequency clock divider is held in reset.
		ACTIVE	1	Transmit high-frequency clock divider is running.
8	XCLKRST			Transmit clock divider reset enable bit.
		RESET	0	Transmit clock divider is held in reset. When the clock divider is in reset, it passes through a divide-by-1 of its input.
		ACTIVE	1	Transmit clock divider is running.
7–5	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.

† For CSL implementation, use the notation MCASP_GBLCTL_*field_symval*

Table B–130. Global Control Register (GBLCTL) Field Descriptions (Continued)

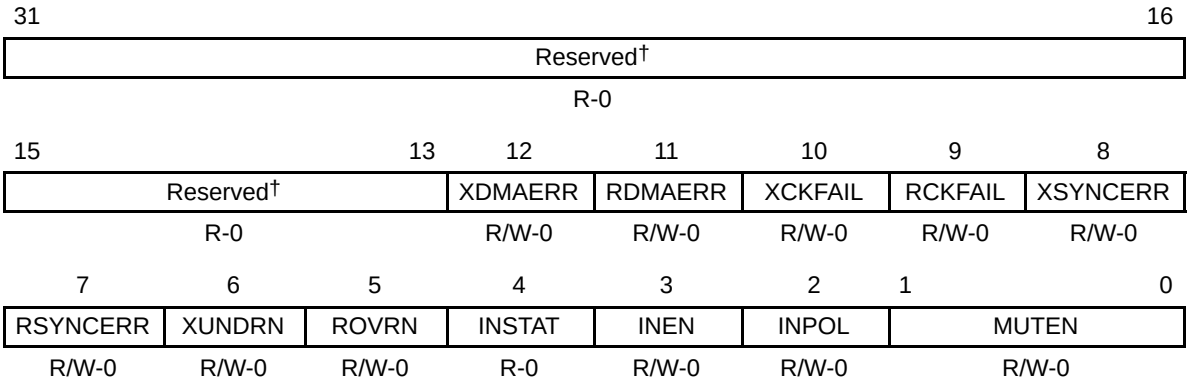
Bit	*field*†	*symval*†	Value	Description
4	RFRST			Receive frame sync generator reset enable bit.
		RESET	0	Receive frame sync generator is reset.
		ACTIVE	1	Receive frame sync generator is active. When released from reset, the receive frame sync generator begins counting serial clocks and generating frame sync as programmed.
3	RSMRST			Receive state machine reset enable bit.
		RESET	0	Receive state machine is held in reset.
		ACTIVE	1	Receive state machine is released from reset. When released from reset, the receive state machine immediately begins detecting frame sync and is ready to receive.
				Receive TDM time slot begins at slot 0 after reset is released.
2	RSRCLR			Receive serializer clear enable bit. By clearing then setting this bit, the receive buffer is flushed.
		CLEAR	0	Receive serializers are cleared.
		ACTIVE	1	Receive serializers are active.
1	RHCLKRST			Receive high-frequency clock divider reset enable bit.
		RESET	0	Receive high-frequency clock divider is held in reset.
		ACTIVE	1	Receive high-frequency clock divider is running.
0	RCLKRST			Receive clock divider reset enable bit.
		RESET	0	Receive clock divider is held in reset. When the clock divider is in reset, it passes through a divide-by-1 of its input.
		ACTIVE	1	Receive clock divider is running.

† For CSL implementation, use the notation MCASP_GBLCTL_*field_symval*

B.10.10 Audio Mute Control Register (AMUTE)

The audio mute control register (AMUTE) controls the McASP audio mute (AMUTE) output pin. The value after reset for register 4 depends on how the pins are being driven. The AMUTE is shown in Figure B–121 and described in Table B–131.

Figure B–121. Audio Mute Control Register (AMUTE) [Offset 0048h]



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.

Table B–131. Audio Mute Control Register (AMUTE) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31–13	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
12	XDMAERR			If transmit EDMA error (XDMAERR), drive AMUTE active enable bit.
		DISABLE	0	Drive is disabled. Detection of transmit EDMA error is ignored by AMUTE.
		ENABLE	1	Drive is enabled (active). Upon detection of transmit EDMA error, AMUTE is active and is driven according to MUTEN bit.
11	RDMAERR			If receive EDMA error (RDMAERR), drive AMUTE active enable bit.
		DISABLE	0	Drive is disabled. Detection of receive EDMA error is ignored by AMUTE.
		ENABLE	1	Drive is enabled (active). Upon detection of receive EDMA error, AMUTE is active and is driven according to MUTEN bit.

† For CSL implementation, use the notation MCASP_AMUTE_*field_symval*

Table B–131. Audio Mute Control Register (AMUTE) Field Descriptions (Continued)

Bit	*field*†	*symval*†	Value	Description
10	XCKFAIL			If transmit clock failure (XCKFAIL), drive AMUTE active enable bit.
		DISABLE	0	Drive is disabled. Detection of transmit clock failure is ignored by AMUTE.
		ENABLE	1	Drive is enabled (active). Upon detection of transmit clock failure, AMUTE is active and is driven according to MUTEN bit.
9	RCKFAIL			If receive clock failure (RCKFAIL), drive AMUTE active enable bit.
		DISABLE	0	Drive is disabled. Detection of receive clock failure is ignored by AMUTE.
		ENABLE	1	Drive is enabled (active). Upon detection of receive clock failure, AMUTE is active and is driven according to MUTEN bit.
8	XSYNCERR			If unexpected transmit frame sync error (XSYNCERR), drive AMUTE active enable bit.
		DISABLE	0	Drive is disabled. Detection of unexpected transmit frame sync error is ignored by AMUTE.
		ENABLE	1	Drive is enabled (active). Upon detection of unexpected transmit frame sync error, AMUTE is active and is driven according to MUTEN bit.
7	RSYNCERR			If unexpected receive frame sync error (RSYNCERR), drive AMUTE active enable bit.
		DISABLE	0	Drive is disabled. Detection of unexpected receive frame sync error is ignored by AMUTE.
		ENABLE	1	Drive is enabled (active). Upon detection of unexpected receive frame sync error, AMUTE is active and is driven according to MUTEN bit.
6	XUNDRN			If transmit underrun error (XUNDRN), drive AMUTE active enable bit.
		DISABLE	0	Drive is disabled. Detection of transmit underrun error is ignored by AMUTE.
		ENABLE	1	Drive is enabled (active). Upon detection of transmit underrun error, AMUTE is active and is driven according to MUTEN bit.

† For CSL implementation, use the notation MCASP_AMUTE_*field_symval*

Table B–131. Audio Mute Control Register (AMUTE) Field Descriptions (Continued)

Bit	*field*†	*symval*†	Value	Description
5	ROVRN			If receiver overrun error (ROVRN), drive AMUTE active enable bit.
		DISABLE	0	Drive is disabled. Detection of receiver overrun error is ignored by AMUTE.
		ENABLE	1	Drive is enabled (active). Upon detection of receiver overrun error, AMUTE is active and is driven according to MUTEN bit.
4	INSTAT	OF(*value*)		Audio mute in (AMUTEIN) error detection status pin.
			0	AMUTEIN pin is inactive.
			1	AMUTEIN pin is active. Audio mute in error is detected.
3	INEN			Drive AMUTE active when AMUTEIN error is active (INSTAT = 1).
		DISABLE	0	Drive is disabled. AMUTEIN is ignored by AMUTE.
		ENABLE	1	Drive is enabled (active). INSTAT = 1 drives AMUTE active.
2	INPOL			Audio mute in (AMUTEIN) polarity select bit.
		ACTHIGH	0	Polarity is active high. A high on AMUTEIN sets INSTAT to 1.
		ACTLOW	1	Polarity is active low. A low on AMUTEIN sets INSTAT to 1.
1–0	MUTEN			AMUTE pin enable bit (unless overridden by GPIO registers).
		DISABLE	0	AMUTE pin is disabled, pin goes to tri-state condition.
		ERRHIGH	1h	AMUTE pin is driven high if error is detected.
		ERRLOW	2h	AMUTE pin is driven low if error is detected.
			3h	Reserved

† For CSL implementation, use the notation MCASP_AMUTE_*field_symval*

B.10.11 Digital Loopback Control Register (DLBCTL)

The digital loopback control register (DLBCTL) controls the internal loopback settings of the McASP in TDM mode. The DLBCTL is shown in Figure B–122 and described in Table B–132.

Figure B–122. Digital Loopback Control Register (DLBCTL) [Offset 004Ch]

31	4	3	2	1	0
Reserved†		MODE		ORD	DLBEN
R-0		R/W-0		R/W-0	R/W-0

Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.

Table B–132. Digital Loopback Control Register (DLBCTL) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31–4	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
3–2	MODE			Loopback generator mode bits. Applies only when loopback mode is enabled (DLBEN = 1).
			0	Default and reserved on loopback mode (DLBEN = 1). When in non–loopback mode (DLBEN = 0), MODE should be left at default (00). When in loopback mode (DLBEN = 1), MODE = 00 is reserved and not applicable.
		XMTCLK	1h	Transmit clock and frame sync generators used by both transmit and receive sections. When in loopback mode (DLBEN = 1), MODE must be 01.
			2h–3h	Reserved
1	ORD			Loopback order bit when loopback mode is enabled (DLBEN = 1).
		XMTODD	0	Odd serializers N+1 transmit to even serializers N that receive. The corresponding serializers must be programmed properly.
		XMTEVEN	1	Even serializers N transmit to odd serializers N+1 that receive. The corresponding serializers must be programmed properly.
0	DLBEN			Loopback mode enable bit.
		DISABLE	0	Loopback mode is disabled.
		ENABLE	1	Loopback mode is enabled.

† For CSL implementation, use the notation MCASP_DLBCTL_*field_symval*

B.10.12 DIT Mode Control Register (DITCTL)

The DIT mode control register (DITCTL) controls DIT operations of the McASP. The DITCTL is shown in Figure B–123 and described in Table B–133.

Figure B–123. DIT Mode Control Register (DITCTL) [Offset 0050h]

31	4	3	2	1	0
Reserved†		VB	VA	—†	DITEN
R-0		R/W-0	R/W-0	R-0	R/W-0

Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.

Table B–133. DIT Mode Control Register (DITCTL) Field Descriptions

Bit	***field***†	***symval***†	Value	Description
31–4	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
3	VB			Valid bit for odd time slots (DIT right subframe).
		ZERO	0	V bit is 0 during odd DIT subframes.
		ONE	1	V bit is 1 during odd DIT subframes.
2	VA			Valid bit for even time slots (DIT left subframe).
		ZERO	0	V bit is 0 during even DIT subframes.
		ONE	1	V bit is 1 during even DIT subframes.
1	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
0	DITEN			DIT mode enable bit. DITEN should only be changed while XSMRST in GBLCTL is in reset (and for startup, XSRCLR also in reset). However, it is not necessary to reset XCLKRST or XHCLKRST in GBLCTL to change DITEN.
		TDM	0	DIT mode is disabled. Transmitter operates in TDM or burst mode.
		DIT	1	DIT mode is enabled. Transmitter operates in DIT encoded mode.

† For CSL implementation, use the notation MCASP_DITCTL_*field_symval*

B.10.13 Receiver Global Control Register (RGBLCTL)

Alias of the global control register (GBLCTL). Writing to the receiver global control register (RGBLCTL) affects only the receive bits of GBLCTL (bits 4–0). Reads from RGBLCTL return the value of GBLCTL. RGBLCTL allows the receiver to be reset independently from the transmitter. The RGBLCTL is shown in Figure B–124 and described in Table B–134. See section B.10.9 for a detailed description of GBLCTL.

Figure B–124. Receiver Global Control Register (RGBLCTL) [Offset 0060h]

31	16
Reserved†	
R-0	

15	13	12	11	10	9	8
Reserved†		XFRST	XSMRST	XSRCLR	XHCLKRST	XCLKRST
R-0		R-0	R-0	R-0	R-0	R-0

7	5	4	3	2	1	0
Reserved†		RFRST	RSMRST	RSRCLR	RHCLKRST	RCLKRST
R-0		R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.

Table B–134. Receiver Global Control Register (RGBLCTL) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31–13	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
12	XFRST	–	x	Transmit frame sync generator reset enable bit. A read of this bit returns the XFRST bit value of GBLCTL. Writes have no effect.
11	XSMRST	–	x	Transmit state machine reset enable bit. A read of this bit returns the XSMRST bit value of GBLCTL. Writes have no effect.
10	XSRCLR	–	x	Transmit serializer clear enable bit. A read of this bit returns the XSRCLR bit value of GBLCTL. Writes have no effect.

† For CSL implementation, use the notation MCASP_RGBLCTL_*field_symval*

Table B–134. Receiver Global Control Register (RGBLCTL) Field Descriptions (Continued)

Bit	*field*†	*symval*†	Value	Description
9	XHCLKRST	–	x	Transmit high-frequency clock divider reset enable bit. A read of this bit returns the XHCLKRST bit value of GBLCTL. Writes have no effect.
8	XCLKRST	–	x	Transmit clock divider reset enable bit. a read of this bit returns the XCLKRST bit value of GBLCTL. Writes have no effect.
7–5	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
4	RFRST			Receive frame sync generator reset enable bit. A write to this bit affects the RFRST bit of GBLCTL.
		RESET	0	Receive frame sync generator is reset.
		ACTIVE	1	Receive frame sync generator is active.
3	RSMRST			Receive state machine reset enable bit. A write to this bit affects the RSMRST bit of GBLCTL.
		RESET	0	Receive state machine is held in reset.
		ACTIVE	1	Receive state machine is released from reset.
2	RSRCLR			Receive serializer clear enable bit. A write to this bit affects the RSRCLR bit of GBLCTL.
		CLEAR	0	Receive serializers are cleared.
		ACTIVE	1	Receive serializers are active.
1	RHCLKRST			Receive high-frequency clock divider reset enable bit. A write to this bit affects the RHCLKRST bit of GBLCTL.
		RESET	0	Receive high-frequency clock divider is held in reset.
		ACTIVE	1	Receive high-frequency clock divider is running.
0	RCLKRST			Receive clock divider reset enable bit. A write to this bit affects the RCLKRST bit of GBLCTL.
		RESET	0	Receive clock divider is held in reset.
		ACTIVE	1	Receive clock divider is running.

† For CSL implementation, use the notation MCASP_RGBLCTL_*field_symval*

B.10.14 Receive Format Unit Bit Mask Register (RMASK)

The receive format unit bit mask register (RMASK) determines which bits of the received data are masked off and padded with a known value before being read by the CPU or EDMA. The RMASK is shown in Figure B–125 and described in Table B–135.

Figure B–125. Receive Format Unit Bit Mask Register (RMASK) [Offset 0064h]

31	30	29	28	27	26	25	24
RMASK31	RMASK30	RMASK29	RMASK28	RMASK27	RMASK26	RMASK25	RMASK24
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
23	22	21	20	19	18	17	16
RMASK23	RMASK22	RMASK21	RMASK20	RMASK19	RMASK18	RMASK17	RMASK16
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
15	14	13	12	11	10	9	8
RMASK15	RMASK14	RMASK13	RMASK12	RMASK11	RMASK10	RMASK9	RMASK8
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
7	6	5	4	3	2	1	0
RMASK7	RMASK6	RMASK5	RMASK4	RMASK3	RMASK2	RMASK1	RMASK0
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

Legend: R/W = Read/Write; -*n* = value after reset

Table B–135. Receive Format Unit Bit Mask Register (RMASK) Field Descriptions

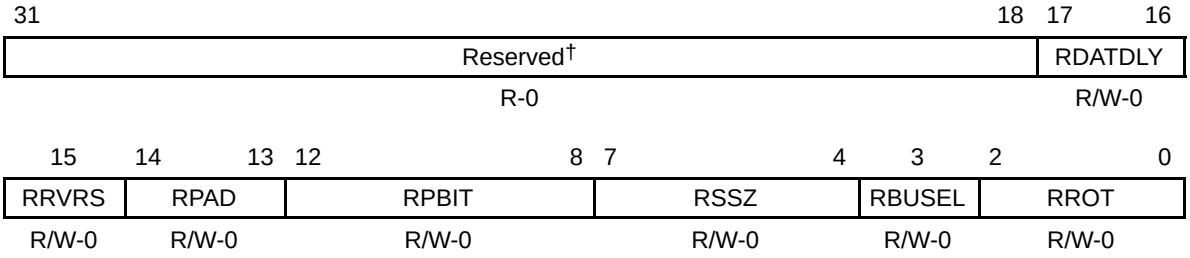
Bit	*field*†	*symval*†	Value	Description
31–0	RMASK[31–0]			Receive data mask enable bit.
		USEMASK	0	Corresponding bit of receive data (after passing through reverse and rotate units) is masked out and then padded with the selected bit pad value (RPAD and RPBIT bits in RFMT).
		NOMASK	1	Corresponding bit of receive data (after passing through reverse and rotate units) is returned to CPU or EDMA.

† For CSL implementation, use the notation MCASP_RMASK_RMASK*n_symval*

B.10.15 Receive Bit Stream Format Register (RFMT)

The receive bit stream format register (RFMT) configures the receive data format. The RFMT is shown in Figure B–126 and described in Table B–136.

Figure B–126. Receive Bit Stream Format Register (RFMT) [Offset 0068h]



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.

Table B–136. Receive Bit Stream Format Register (RFMT) Field Descriptions

Bit	***field***†	***symval***†	Value	Description
31–18	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
17–16	RDATDLY			Receive bit delay.
		0BIT	0	0-bit delay. The first receive data bit, AXR[n], occurs in the same ACLKR cycle as the receive frame sync (AFSR).
		1BIT	1h	1-bit delay. The first receive data bit, AXR[n], occurs one ACLKR cycle after the receive frame sync (AFSR).
		2BIT	2h	2-bit delay. The first receive data bit, AXR[n], occurs two ACLKR cycles after the receive frame sync (AFSR).
			3h	Reserved
15	RRVRS			Receive serial bitstream order.
		LSBFIRST	0	Bitstream is LSB first. No bit reversal is performed in receive format bit reverse unit.
		MSBFIRST	1	Bitstream is MSB first. Bit reversal is performed in receive format bit reverse unit.

† For CSL implementation, use the notation MCASP_RFMT_*field_symval*

Table B–136. Receive Bit Stream Format Register (RFMT) Field Descriptions (Continued)

Bit	*field*†	*symval*†	Value	Description
14–13	RPAD			Pad value for extra bits in slot not belonging to the word. This field only applies to bits when RMASK[n] = 0.
		ZERO	0	Pad extra bits with 0.
		ONE	1h	Pad extra bits with 1.
		RPBIT	2h	Pad extra bits with one of the bits from the word as specified by RPBIT bits.
			3h	Reserved
12–8	RPBIT	OF(*value*)		RPBIT value determines which bit (as read by the CPU or EDMA from RBUF[n]) is used to pad the extra bits. This field only applies when RPAD = 2h.
			0	Pad with bit 0 value.
			1h–1Fh	Pad with bit 1 to bit 31 value.
7–4	RSSZ			Receive slot size.
			0–2h	Reserved
		8BITS	3h	Slot size is 8 bits.
			4h	Reserved
		12BITS	5h	Slot size is 12 bits.
			6h	Reserved
		16BITS	7h	Slot size is 16 bits.
			8h	Reserved
		20BITS	9h	Slot size is 20 bits.
			Ah	Reserved
		24BITS	Bh	Slot size is 24 bits.
			Ch	Reserved
		28BITS	Dh	Slot size is 28 bits.
			Eh	Reserved
		32BITS	Fh	Slot size is 32 bits.

† For CSL implementation, use the notation MCASP_RFMT_*field_symval*

Table B–136. Receive Bit Stream Format Register (RFMT) Field Descriptions (Continued)

Bit	*field*†	*symval*†	Value	Description
3	RBUSEL			Selects whether reads from serializer buffer XRBUF[n] originate from the configuration bus (CFG) or the data (DAT) port.
		DAT	0	Reads from XRBUF[n] originate on data port. Reads from XRBUF[n] on configuration bus are ignored.
		CFG	1	Reads from XRBUF[n] originate on configuration bus. Reads from XRBUF[n] on data port are ignored.
2–0	RROT			Right-rotation value for receive rotate right format unit.
		NONE	0	Rotate right by 0 (no rotation).
		4BITS	1h	Rotate right by 4 bit positions.
		8BITS	2h	Rotate right by 8 bit positions.
		12BITS	3h	Rotate right by 12 bit positions.
		16BITS	4h	Rotate right by 16 bit positions.
		20BITS	5h	Rotate right by 20 bit positions.
		24BITS	6h	Rotate right by 24 bit positions.
		28BITS	7h	Rotate right by 28 bit positions.

† For CSL implementation, use the notation MCASP_RFMT_*field_symval*

B.10.16 Receive Frame Sync Control Register (AFSRCTL)

The receive frame sync control register (AFSRCTL) configures the receive frame sync (AFSR). The AFSRCTL is shown in Figure B–127 and described in Table B–137.

Figure B–127. Receive Frame Sync Control Register (AFSRCTL) [Offset 006Ch]

31	16
Reserved†	
R-0	

15	7	6	5	4	3	2	1	0
RMOD		Reserved†		FRWID	Reserved†		FSRM	FSRP
R/W-0		R-0		R/W-0	R-0		R/W-0	R/W-0

Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.

Table B–137. Receive Frame Sync Control Register (AFSRCTL) Field Descriptions

Bit	***field***†	***symval***†	Value	Description
31–16	Reserved	–	0	value for future device compatibility.
15–7	RMOD	OF(*value*)		Receive frame sync mode select bits.
		BURST	0	Burst mode
			1h	Reserved
			2h–20h	2-slot TDM (I2S mode) to 32-slot TDM
			21h–17Fh	Reserved
			180h	384-slot TDM (external DIR IC inputting 384-slot DIR frames to McASP over I2S interface)
6–5	Reserved	–	0	value for future device compatibility.
4	FRWID			Receive frame sync width select bit indicates the width of the receive frame sync (AFSR) during its active period.
		BIT	0	Single bit
		WORD	1	Single word
3–2	Reserved	–	0	value for future device compatibility.

† For CSL implementation, use the notation MCASP_AFSRCTL_*field_symval*

Table B–137. Receive Frame Sync Control Register (AFSRCTL) Field Descriptions (Continued)

Bit	*field*†	*symval*†	Value	Description
1	FSRM			Receive frame sync generation select bit.
		EXTERNAL	0	Externally-generated receive frame sync
		INTERNAL	1	Internally-generated receive frame sync
0	FSRP			Receive frame sync polarity select bit.
		ACTIVEHIGH	0	A rising edge on receive frame sync (AFSR) indicates the beginning of a frame.
		ACTIVELOW	1	A falling edge on receive frame sync (AFSR) indicates the beginning of a frame.

† For CSL implementation, use the notation MCASP_AFSRCTL_*field_symval*

B.10.17 Receive Clock Control Register (ACLKRCTL)

The receive clock control register (ACLKRCTL) configures the receive bit clock (ACLKR) and the receive clock generator. The ACLKRCTL is shown in Figure B–128 and described in Table B–138.

Figure B–128. Receive Clock Control Register (ACLKRCTL) [Offset 0070h]

31	8	7	6	5	4	0
Reserved†		CLKRP	—†	CLKRM	CLKRDIV	
R-0		R/W-0	R-0	R/W-1	R/W-0	

Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.

Table B–138. Receive Clock Control Register (ACLKRCTL) Field Descriptions

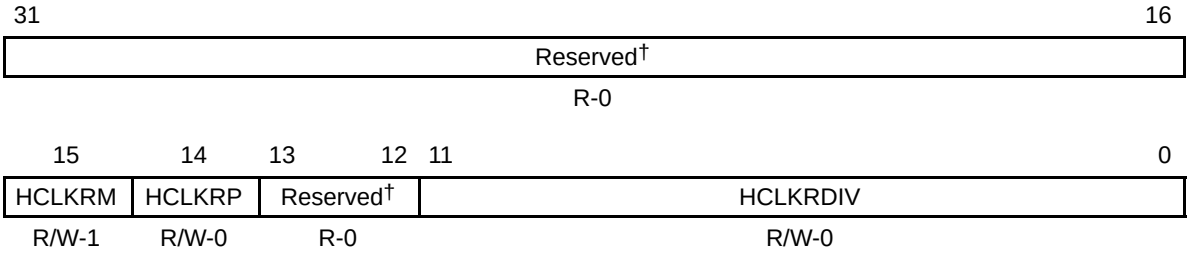
Bit	*field*	*symval*†	Value	Description
31–8	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
7	CLKRP			Receive bitstream clock polarity select bit.
		FALLING	0	Falling edge. Receiver samples data on the falling edge of the serial clock, so the external transmitter driving this receiver must shift data out on the rising edge of the serial clock.
		RISING	1	Rising edge. Receiver samples data on the rising edge of the serial clock, so the external transmitter driving this receiver must shift data out on the falling edge of the serial clock.
6	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
5	CLKRM			Receive bit clock source bit.
		EXTERNAL	0	External receive clock source from ACLKR pin.
		INTERNAL	1	Internal receive clock source from output of programmable bit clock divider.
4–0	CLKRDIV	OF(*value*)		Receive bit clock divide ratio bits determine the divide-down ratio from AHCLKR to ACLKR.
			0	Divide-by-1
			1h	Divide-by-2
			2h–1Fh	Divide-by-3 to divide-by-32

† For CSL implementation, use the notation MCASP_ACLKRCTL_*field_symval*

B.10.18 Receive High-Frequency Clock Control Register (AHCLKRCTL)

The receive high-frequency clock control register (AHCLKRCTL) configures the receive high-frequency master clock (AHCLKR) and the receive clock generator. The AHCLKRCTL is shown in Figure B–129 and described in Table B–139.

Figure B–129. Receive High-Frequency Clock Control Register (AHCLKRCTL) [Offset 0074h]



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.

Table B–139. Receive High-Frequency Clock Control Register (AHCLKRCTL) Field Descriptions

Bit	*field*	*symval*†	Value	Description
31–16	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
15	HCLKRM			Receive high-frequency clock source bit.
		EXTERNAL	0	External receive high-frequency clock source from AHCLKR pin.
		INTERNAL	1	Internal receive high-frequency clock source from output of programmable high clock divider.

† For CSL implementation, use the notation MCASP_AHCLKRCTL_*field_symval*

Table B–139. Receive High-Frequency Clock Control Register (AHCLKRCTL) Field Descriptions (Continued)

Bit	*field*	*symval*†	Value	Description
14	HCLKRP			Receive bitstream high-frequency clock polarity select bit.
		RISING	0	Rising edge. AHCLKR is not inverted before programmable bit clock divider. In the special case where the receive bit clock (ACLKR) is internally generated and the programmable bit clock divider is set to divide-by-1 (CLKRDIV = 0 in ACLKRCTL), AHCLKR is directly passed through to the ACLKR pin.
		FALLING	1	Falling edge. AHCLKR is inverted before programmable bit clock divider. In the special case where the receive bit clock (ACLKR) is internally generated and the programmable bit clock divider is set to divide-by-1 (CLKRDIV = 0 in ACLKRCTL), AHCLKR is directly passed through to the ACLKR pin.
13–12	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
11–0	HCLKRDIV	OF(*value*)		Receive high-frequency clock divide ratio bits determine the divide-down ratio from AUXCLK to AHCLKR.
			0	Divide-by-1
			1h	Divide-by-2
			2h–FFFh	Divide-by-3 to divide-by-4096

† For CSL implementation, use the notation MCASP_AHCLKRCTL_*field_symval*

B.10.19 Receive TDM Time Slot Register (RTDM)

The receive TDM time slot register (RTDM) specifies which TDM time slot the receiver is active. The RTDM is shown in Figure B–130 and described in Table B–140.

Figure B–130. Receive TDM Time Slot Register (RTDM) [Offset 0078h]

31	30	29	28	27	26	25	24
RTDMS31	RTDMS30	RTDMS29	RTDMS28	RTDMS27	RTDMS26	RTDMS25	RTDMS24
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
23	22	21	20	19	18	17	16
RTDMS23	RTDMS22	RTDMS21	RTDMS20	RTDMS19	RTDMS18	RTDMS17	RTDMS16
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
15	14	13	12	11	10	9	8
RTDMS15	RTDMS14	RTDMS13	RTDMS12	RTDMS11	RTDMS10	RTDMS9	RTDMS8
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
7	6	5	4	3	2	1	0
RTDMS7	RTDMS6	RTDMS5	RTDMS4	RTDMS3	RTDMS2	RTDMS1	RTDMS0
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

Legend: R/W = Read/Write; *-n* = value after reset

Table B–140. Receive TDM Time Slot Register (RTDM) Field Descriptions

Bit	***field***†	***symval***†	Value	Description
31–0	RTDMS[31–0]			Receiver mode during TDM time slot *n*.
		INACTIVE	0	Receive TDM time slot *n* is inactive. The receive serializer does not shift in data during this slot.
		ACTIVE	1	Receive TDM time slot *n* is active. The receive serializer shifts in data during this slot.

† For CSL implementation, use the notation MCASP_RTDM_RTDMS*n*_*symval*

B.10.20 Receiver Interrupt Control Register (RINTCTL)

The receiver interrupt control register (RINTCTL) controls generation of the McASP receive interrupt (RINT). When the register bit(s) is set to 1, the occurrence of the enabled McASP condition(s) generates RINT. The RINTCTL is shown in Figure B–131 and described in Table B–141. See section B.10.21 for a description of the interrupt conditions.

Figure B–131. Receiver Interrupt Control Register (RINTCTL) [Offset 007Ch]

31	30	29	28	27	26	25	24
Reserved†							
R-0							

23	22	21	20	19	18	17	16
Reserved†							
R-0							

15	14	13	12	11	10	9	8
Reserved†							
R-0							

7	6	5	4	3	2	1	0
RSTAFRM	Reserved†	RDATA	RLAST	RDMAERR	RCKFAIL	RSYNCERR	ROVRN
R/W-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.

Table B–141. Receiver Interrupt Control Register (RINTCTL) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31–8	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
7	RSTAFRM			Receive start of frame interrupt enable bit.
		DISABLE	0	Interrupt is disabled. A receive start of frame interrupt does not generate a McASP receive interrupt (RINT).
		ENABLE	1	Interrupt is enabled. A receive start of frame interrupt generates a McASP receive interrupt (RINT).
6	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.

† For CSL implementation, use the notation MCASP_RINTCTL_*field_symval*

Table B–141. Receiver Interrupt Control Register (RINTCTL) Field Descriptions (Continued)

Bit	*field*†	*symval*†	Value	Description
5	RDATA			Receive data ready interrupt enable bit.
		DISABLE	0	Interrupt is disabled. A receive data ready interrupt does not generate a McASP receive interrupt (RINT).
		ENABLE	1	Interrupt is enabled. A receive data ready interrupt generates a McASP receive interrupt (RINT).
4	RLAST			Receive last slot interrupt enable bit.
		DISABLE	0	Interrupt is disabled. A receive last slot interrupt does not generate a McASP receive interrupt (RINT).
		ENABLE	1	Interrupt is enabled. A receive last slot interrupt generates a McASP receive interrupt (RINT).
3	RDMAERR			Receive EDMA error interrupt enable bit.
		DISABLE	0	Interrupt is disabled. A receive EDMA error interrupt does not generate a McASP receive interrupt (RINT).
		ENABLE	1	Interrupt is enabled. A receive EDMA error interrupt generates a McASP receive interrupt (RINT).
2	RCKFAIL			Receive clock failure interrupt enable bit.
		DISABLE	0	Interrupt is disabled. A receive clock failure interrupt does not generate a McASP receive interrupt (RINT).
		ENABLE	1	Interrupt is enabled. A receive clock failure interrupt generates a McASP receive interrupt (RINT).
1	RSYNCERR			Unexpected receive frame sync interrupt enable bit.
		DISABLE	0	Interrupt is disabled. An unexpected receive frame sync interrupt does not generate a McASP receive interrupt (RINT).
		ENABLE	1	Interrupt is enabled. An unexpected receive frame sync interrupt generates a McASP receive interrupt (RINT).
0	ROVRN			Receiver overrun interrupt enable bit.
		DISABLE	0	Interrupt is disabled. A receiver overrun interrupt does not generate a McASP receive interrupt (RINT).
		ENABLE	1	Interrupt is enabled. A receiver overrun interrupt generates a McASP receive interrupt (RINT).

† For CSL implementation, use the notation MCASP_RINTCTL_*field_symval*

B.10.21 Receiver Status Register (RSTAT)

The receiver status register (RSTAT) provides the receiver status and receive TDM time slot number. If the McASP logic attempts to set an interrupt flag in the same cycle that the CPU writes to the flag to clear it, the McASP logic has priority and the flag remains set. This also causes a new interrupt request to be generated. The RSTAT is shown in Figure B–132 and described in Table B–142.

Figure B–132. Receiver Status Register (RSTAT) [Offset 0080h]

31						9	8
Reserved†							RERR
R-0							R/W-0
7	6	5	4	3	2	1	0
RDMAERR	RSTAFRM	RDATA	RLAST	RTDMSLOT	RCKFAIL	RSYNCERR	ROVRN
R/W-0	R/W-0	R/W-0	R/W-0	R-0	R/W-0	R/W-0	R/W-0

Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.

Table B–142. Receiver Status Register (RSTAT) Field Descriptions

Bit	***field*†**	***symval*†**	Value	Description
31–9	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
8	RERR	OF(*value*)		RERR bit always returns a logic-OR of: ROVRN | RSYNCERR | RCKFAIL | RDMAERR Allows a single bit to be checked to determine if a receiver error interrupt has occurred.
			0	No errors have occurred.
			1	An error has occurred.

† For CSL implementation, use the notation MCASP_RSTAT_*field_symval*

Table B–142. Receiver Status Register (RSTAT) Field Descriptions (Continued)

Bit	*field*†	*symval*†	Value	Description
7	RDMAERR	OF(*value*)		Receive EDMA error flag. RDMAERR is set when the CPU or EDMA reads more serializers through the data port in a given time slot than were programmed as receivers. Causes a receive interrupt (RINT), if this bit is set and RDMAERR in RINTCTL is set. This bit is cleared by writing a 1 to this bit. Writing a 0 to this bit has no effect.
			0	Receive EDMA error did not occur.
			1	Receive EDMA error did occur.
6	RSTAFRM			Receive start of frame flag. Causes a receive interrupt (RINT), if this bit is set and RSTAFRM in RINTCTL is set. This bit is cleared by writing a 1 to this bit. Writing a 0 to this bit has no effect.
		NO	0	No new receive frame sync (AFSR) is detected.
		YES	1	A new receive frame sync (AFSR) is detected.
5	RDATA			Receive data ready flag. Causes a receive interrupt (RINT), if this bit is set and RDATA in RINTCTL is set. This bit is cleared by writing a 1 to this bit. Writing a 0 to this bit has no effect.
		NO	0	No new data in RBUF.
		YES	1	Data is transferred from XRSR to RBUF and ready to be serviced by the CPU or EDMA. When RDATA is set, it always causes an EDMA event (AREVT).
4	RLAST			Receive last slot flag. RLAST is set along with RDATA, if the current slot is the last slot in a frame. Causes a receive interrupt (RINT), if this bit is set and RLAST in RINTCTL is set. This bit is cleared by writing a 1 to this bit. Writing a 0 to this bit has no effect.
		NO	0	Current slot is not the last slot in a frame.
		YES	1	Current slot is the last slot in a frame. RDATA is also set.
3	RTDMSLOT	OF(*value*)		Returns the LSB of RSLOT. Allows a single read of RSTAT to determine whether the current TDM time slot is even or odd.
			0	Current TDM time slot is odd.
			1	Current TDM time slot is even.

† For CSL implementation, use the notation MCASP_RSTAT_*field_symval*

Table B–142. Receiver Status Register (RSTAT) Field Descriptions (Continued)

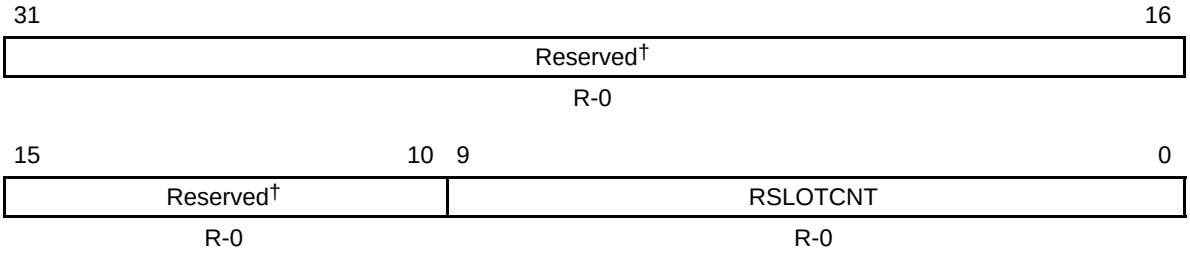
Bit	*field*†	*symval*†	Value	Description
2	RCKFAIL			Receive clock failure flag. RCKFAIL is set when the receive clock failure detection circuit reports an error. Causes a receive interrupt (RINT), if this bit is set and RCKFAIL in RINTCTL is set. This bit is cleared by writing a 1 to this bit. Writing a 0 to this bit has no effect.
		NO	0	Receive clock failure did not occur.
		YES	1	Receive clock failure did occur.
1	RSYNCERR			Unexpected receive frame sync flag. RSYNCERR is set when a new receive frame sync (AFSR) occurs before it is expected. Causes a receive interrupt (RINT), if this bit is set and RSYNCERR in RINTCTL is set. This bit is cleared by writing a 1 to this bit. Writing a 0 to this bit has no effect.
		NO	0	Unexpected receive frame sync did not occur.
		YES	1	Unexpected receive frame sync did occur.
0	ROVRN			Receiver overrun flag. ROVRN is set when the receive serializer is instructed to transfer data from XRSR to RBUF, but the former data in RBUF has not yet been read by the CPU or EDMA. Causes a receive interrupt (RINT), if this bit is set and ROVRN in RINTCTL is set. This bit is cleared by writing a 1 to this bit. Writing a 0 to this bit has no effect.
		NO	0	Receiver overrun did not occur.
		YES	1	Receiver overrun did occur.

† For CSL implementation, use the notation MCASP_RSTAT_*field_symval*

B.10.22 Current Receive TDM Time Slot Registers (RSLOT)

The current receive TDM time slot register (RSLOT) indicates the current time slot for the receive data frame. The RSLOT is shown in Figure B–133 and described in Table B–143.

Figure B–133. Current Receive TDM Time Slot Register (RSLOT) [Offset 0084h]



Legend: R = Read only; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.

Table B–143. Current Receive TDM Time Slot Register (RSLOT) Field Descriptions

Bit	Field	*symval*†	Value	Description
31–10	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
9–0	RSLOTCNT	OF(*value*)	0–17Fh	Current receive time slot count. Legal values: 0 to 383. TDM function is not supported for > 32 time slots. However, TDM time slot counter may count to 383 when used to receive a DIR block (transferred over TDM format).

† For CSL implementation, use the notation MCASP_RSLOT_RSLOTCNT_*symval*

B.10.23 Receive Clock Check Control Register (RCLKCHK)

The receive clock check control register (RCLKCHK) configures the receive clock failure detection circuit. The RCLKCHK is shown in Figure B–134 and described in Table B–144.

Figure B–134. Receive Clock Check Control Register (RCLKCHK) [Offset 0088h]

31–24	23–16	15–8	7–4	3–0
RCNT	RMAX	RMIN	Reserved†	RPS
R-0	R/W-0	R/W-0	R-0	R/W-0

Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.

Table B–144. Receive Clock Check Control Register (RCLKCHK) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31–24	RCNT	OF(*value*)	0–FFh	Receive clock count value (from previous measurement). The clock circuit continually counts the number of DSP system clocks for every 32 receive high-frequency master clock (AHCLKR) signals, and stores the count in RCNT until the next measurement is taken.
23–16	RMAX	OF(*value*)	0–FFh	Receive clock maximum boundary. This 8-bit unsigned value sets the maximum allowed boundary for the clock check counter after 32 receive high-frequency master clock (AHCLKR) signals have been received. If the current counter value is greater than RMAX after counting 32 AHCLKR signals, RCKFAIL in RSTAT is set. The comparison is performed using unsigned arithmetic.
15–8	RMIN	OF(*value*)	0–FFh	Receive clock minimum boundary. This 8-bit unsigned value sets the minimum allowed boundary for the clock check counter after 32 receive high-frequency master clock (AHCLKR) signals have been received. If RCNT is less than RMIN after counting 32 AHCLKR signals, RCKFAIL in RSTAT is set. The comparison is performed using unsigned arithmetic.

† For CSL implementation, use the notation MCASP_RCLKCHK_*field_symval*

Table B–144. Receive Clock Check Control Register (RCLKCHK) Field Descriptions (Continued)

Bit	*field*†	*symval*†	Value	Description
7–4	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
3–0	RPS			Receive clock check prescaler value.
		DIVBY1	0	McASP system clock divided by 1
		DIVBY2	1h	McASP system clock divided by 2
		DIVBY4	2h	McASP system clock divided by 4
		DIVBY8	3h	McASP system clock divided by 8
		DIVBY16	4h	McASP system clock divided by 16
		DIVBY32	5h	McASP system clock divided by 32
		DIVBY64	6h	McASP system clock divided by 64
		DIVBY128	7h	McASP system clock divided by 128
		DIVBY256	8h	McASP system clock divided by 256
			9h–Fh	Reserved

† For CSL implementation, use the notation MCASP_RCLKCHK_*field_symval*

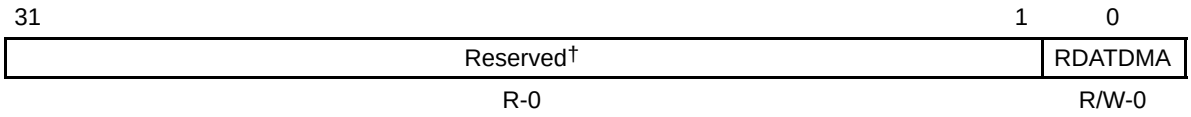
B.10.24 Receiver DMA Event Control Register (REVTCTL)

The receiver DMA event control register (REVTCTL) contains a disable bit for the receiver DMA event. The REVTCTL is shown in Figure B–135 and described in Table B–145.

DSP specific registers

Accessing REVTCTL not implemented on a specific DSP may cause improper device operation.

Figure B–135. Receiver DMA Event Control Register (REVTCTL) [Offset 008Ch]



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.

Table B–145. Receiver DMA Event Control Register (REVTCTL) Field Values

Bit	*field*	*symval*†	Value	Description
31–1	Reserved			Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
0	RDATDMA			Receive data DMA request enable bit.
		ENABLE	0	Receive data DMA request is enabled.
		DISABLE	1	Receive data DMA request is disabled.

† For CSL implementation, use the notation MCASP_REVTCTL_RDATDMA_*symval*

B.10.25 Transmitter Global Control Register (XGBLCTL)

Alias of the global control register (GBLCTL). Writing to the transmitter global control register (XGBLCTL) affects only the transmit bits of GBLCTL (bits 12–8). Reads from XGBLCTL return the value of GBLCTL. XGBLCTL allows the transmitter to be reset independently from the receiver. The XGBLCTL is shown in Figure B–136 and described in Table B–146. See section B.10.9 for a detailed description of GBLCTL.

Figure B–136. Transmitter Global Control Register (XGBLCTL) [Offset 00A0h]

31	16
Reserved†	
R-0	

15	13	12	11	10	9	8
Reserved†		XFRST	XSMRST	XSRCLR	XHCLKRST	XCLKRST
R-0		R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

7	5	4	3	2	1	0
Reserved†		RFRST	RSMRST	RSRCLR	RHCLKRST	RCLKRST
R-0		R-0	R-0	R-0	R-0	R-0

Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.

Table B–146. Transmitter Global Control Register (XGBLCTL) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31–13	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
12	XFRST			Transmit frame sync generator reset enable bit. A write to this bit affects the XFRST bit of GBLCTL.
		RESET	0	Transmit frame sync generator is reset.
		ACTIVE	1	Transmit frame sync generator is active.
11	XSMRST			Transmit state machine reset enable bit. A write to this bit affects the XSMRST bit of GBLCTL.
		RESET	0	Transmit state machine is held in reset.
		ACTIVE	1	Transmit state machine is released from reset.

† For CSL implementation, use the notation MCASP_XGBLCTL_*field_symval*

Table B–146. Transmitter Global Control Register (XGBLCTL) Field Descriptions (Continued)

Bit	*field*†	*symval*†	Value	Description
10	XSRCLR			Transmit serializer clear enable bit. A write to this bit affects the XSRCLR bit of GBLCTL.
		CLEAR	0	Transmit serializers are cleared.
		ACTIVE	1	Transmit serializers are active.
9	XHCLKRST			Transmit high-frequency clock divider reset enable bit. A write to this bit affects the XHCLKRST bit of GBLCTL.
		RESET	0	Transmit high-frequency clock divider is held in reset.
		ACTIVE	1	Transmit high-frequency clock divider is running.
8	XCLKRST			Transmit clock divider reset enable bit. A write to this bit affects the XCLKRST bit of GBLCTL.
		RESET	0	Transmit clock divider is held in reset.
		ACTIVE	1	Transmit clock divider is running.
7–5	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
4	RFRST	–	x	Receive frame sync generator reset enable bit. A read of this bit returns the RFRST bit value of GBLCTL. Writes have no effect.
3	RSMRST	–	x	Receive state machine reset enable bit. A read of this bit returns the RSMRST bit value of GBLCTL. Writes have no effect.
2	RSRCLR	–	x	Receive serializer clear enable bit. A read of this bit returns the RSRSCLR bit value of GBLCTL. Writes have no effect.
1	RHCLKRST	–	x	Receive high-frequency clock divider reset enable bit. A read of this bit returns the RHCLKRST bit value of GBLCTL. Writes have no effect.
0	RCLKRST	–	x	Receive clock divider reset enable bit. A read of this bit returns the RCLKRST bit value of GBLCTL. Writes have no effect.

† For CSL implementation, use the notation MCASP_XGBLCTL_*field_symval*

B.10.26 Transmit Format Unit Bit Mask Register (XMASK)

The transmit format unit bit mask register (XMASK) determines which bits of the transmitted data are masked off and padded with a known value before being shifted out the McASP. The XMASK is shown in Figure B–137 and described in Table B–147.

Figure B–137. Transmit Format Unit Bit Mask Register (XMASK) [Offset 00A4h]

31	30	29	28	27	26	25	24
XMASK31	XMASK30	XMASK29	XMASK28	XMASK27	XMASK26	XMASK25	XMASK24
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
23	22	21	20	19	18	17	16
XMASK23	XMASK22	XMASK21	XMASK20	XMASK19	XMASK18	XMASK17	XMASK16
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
15	14	13	12	11	10	9	8
XMASK15	XMASK14	XMASK13	XMASK12	XMASK11	XMASK10	XMASK9	XMASK8
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
7	6	5	4	3	2	1	0
XMASK7	XMASK6	XMASK5	XMASK4	XMASK3	XMASK2	XMASK1	XMASK0
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

Legend: R/W = Read/Write; *-n* = value after reset

Table B–147. Transmit Format Unit Bit Mask Register (XMASK) Field Descriptions

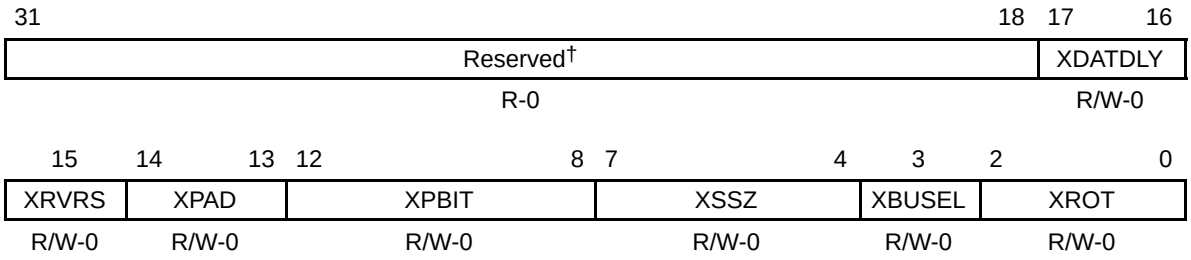
Bit	*field*†	*symval*†	Value	Description
31–0	XMASK[31–0]			Transmit data mask enable bit.
		USEMASK	0	Corresponding bit of transmit data (before passing through reverse and rotate units) is masked out and then padded with the selected bit pad value (XPAD and XPBIT bits in XFMT), which is transmitted out the McASP in place of the original bit.
		NOMASK	1	Corresponding bit of transmit data (before passing through reverse and rotate units) is transmitted out the McASP.

† For CSL implementation, use the notation MCASP_XMASK_XMASK*n_symval*

B.10.27 Transmit Bit Stream Format Register (XFMT)

The transmit bit stream format register (XFMT) configures the transmit data format. The XFMT is shown in Figure B–138 and described in Table B–148.

Figure B–138. Transmit Bit Stream Format Register (XFMT) [Offset 00A8h]



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.

Table B–148. Transmit Bit Stream Format Register (XFMT) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31–18	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
17–16	XDATDLY			Transmit sync bit delay.
		0BIT	0	0-bit delay. The first transmit data bit, AXR[n], occurs in the same ACLKX cycle as the transmit frame sync (AFSX).
		1BIT	1h	1-bit delay. The first transmit data bit, AXR[n], occurs one ACLKX cycle after the transmit frame sync (AFSX).
		2BIT	2h	2-bit delay. The first transmit data bit, AXR[n], occurs two ACLKX cycles after the transmit frame sync (AFSX).
			3h	Reserved
15	XRVRS			Transmit serial bitstream order.
		LSBFIRST	0	Bitstream is LSB first. No bit reversal is performed in transmit format bit reverse unit.
		MSBFIRST	1	Bitstream is MSB first. Bit reversal is performed in transmit format bit reverse unit.

† For CSL implementation, use the notation MCASP_XFMT_*field_symval*

Table B–148. Transmit Bit Stream Format Register (XFMT) Field Descriptions (Continued)

Bit	*field*†	*symval*†	Value	Description
14–13	XPAD			Pad value for extra bits in slot not belonging to word defined by XMASK. This field only applies to bits when XMASK[n] = 0.
		ZERO	0	Pad extra bits with 0.
		ONE	1h	Pad extra bits with 1.
		XPBIT	2h	Pad extra bits with one of the bits from the word as specified by XPBIT bits.
			3h	Reserved
12–8	XPBIT	OF(*value*)		XPBIT value determines which bit (as written by the CPU or EDMA to XBUF[n]) is used to pad the extra bits before shifting. This field only applies when XPAD = 2h.
			0	Pad with bit 0 value.
			1–1Fh	Pad with bit 1 to bit 31 value.
7–4	XSSZ			Transmit slot size.
			0–2h	Reserved
		8BITS	3h	Slot size is 8 bits.
			4h	Reserved
		12BITS	5h	Slot size is 12 bits.
			6h	Reserved
		16BITS	7h	Slot size is 16 bits.
			8h	Reserved
		20BITS	9h	Slot size is 20 bits.
			Ah	Reserved
		24BITS	Bh	Slot size is 24 bits.
			Ch	Reserved
		28BITS	Dh	Slot size is 28 bits.
			Eh	Reserved
		32BITS	Fh	Slot size is 32 bits.

† For CSL implementation, use the notation MCASP_XFMT_*field_symval*

Table B–148. Transmit Bit Stream Format Register (XFMT) Field Descriptions (Continued)

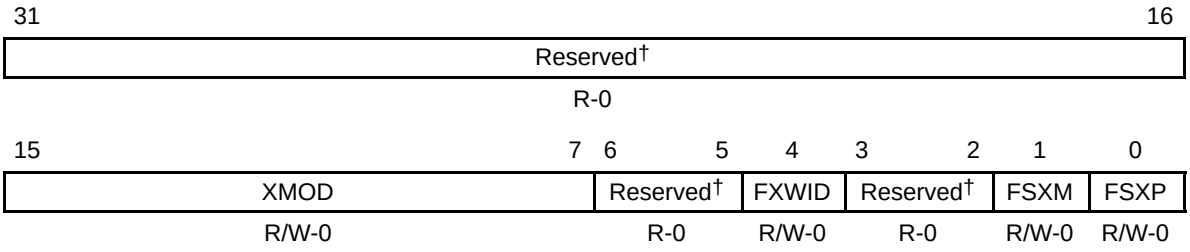
Bit	*field*†	*symval*†	Value	Description
3	XBUSEL			Selects whether writes to serializer buffer XRBUF[n] originate from the configuration bus (CFG) or the data (DAT) port.
		DAT	0	Writes to XRBUF[n] originate from the data port. Writes to XRBUF[n] from the configuration bus are ignored with no effect to the McASP.
		CFG	1	Writes to XRBUF[n] originate from the configuration bus. Writes to XRBUF[n] from the data port are ignored with no effect to the McASP.
2–0	XROT			Right-rotation value for transmit rotate right format unit.
		NONE	0	Rotate right by 0 (no rotation).
		4BITS	1h	Rotate right by 4 bit positions.
		8BITS	2h	Rotate right by 8 bit positions.
		12BITS	3h	Rotate right by 12 bit positions.
		16BITS	4h	Rotate right by 16 bit positions.
		20BITS	5h	Rotate right by 20 bit positions.
		24BITS	6h	Rotate right by 24 bit positions.
		28BITS	7h	Rotate right by 28 bit positions.

† For CSL implementation, use the notation MCASP_XFMT_*field_symval*

B.10.28 Transmit Frame Sync Control Register (AFSXCTL)

The transmit frame sync control register (AFSXCTL) configures the transmit frame sync (AFSX). The AFSXCTL is shown in Figure B–139 and described in Table B–149.

Figure B–139. Transmit Frame Sync Control Register (AFSXCTL) [Offset 00ACh]



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.

Table B–149. Transmit Frame Sync Control Register (AFSXCTL) Field Descriptions

Bit	***field***†	***symval***†	Value	Description
31–16	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
15–7	XMOD	OF(*value*)		Transmit frame sync mode select bits.
		BURST	0	Burst mode
			1h	Reserved
			2h–20h	2-slot TDM (I2S mode) to 32-slot TDM
			21h–17Fh	Reserved
			180h	384-slot DIT mode
6–5	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.

† For CSL implementation, use the notation MCASP_AFSXCTL_*field_symval*

Table B–149. Transmit Frame Sync Control Register (AFSXCTL) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
4	FXWID			Transmit frame sync width select bit indicates the width of the transmit frame sync (AFSX) during its active period.
		BIT	0	Single bit
		WORD	1	Single word
3–2	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
1	FSXM			Transmit frame sync generation select bit.
		EXTERNAL	0	Externally-generated transmit frame sync
		INTERNAL	1	Internally-generated transmit frame sync
0	FSXP			Transmit frame sync polarity select bit.
		ACTIVEHIGH	0	A rising edge on transmit frame sync (AFSX) indicates the beginning of a frame.
		ACTIVELOW	1	A falling edge on transmit frame sync (AFSX) indicates the beginning of a frame.

† For CSL implementation, use the notation MCASP_AFSXCTL_*field_symval*

B.10.29 Transmit Clock Control Register (ACLKXCTL)

The transmit clock control register (ACLKXCTL) configures the transmit bit clock (ACLKX) and the transmit clock generator. The ACLKXCTL is shown in Figure B–140 and described in Table B–150.

Figure B–140. Transmit Clock Control Register (ACLKXCTL) [Offset 00B0h]

31	8	7	6	5	4	0
Reserved†		CLKXP	ASYNC	CLKXM	CLKXDIV	
R-0		R/W-0	R/W-1	R/W-1	R/W-0	

Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.

Table B–150. Transmit Clock Control Register (ACLKXCTL) Field Descriptions

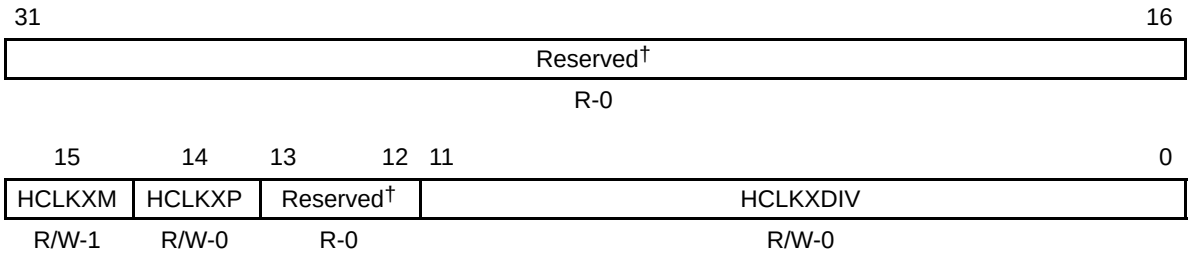
Bit	*field*	*symval*†	Value	Description
31–8	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
7	CLKXP			Transmit bitstream clock polarity select bit.
		RISING	0	Rising edge. External receiver samples data on the falling edge of the serial clock, so the transmitter must shift data out on the rising edge of the serial clock.
		FALLING	1	Falling edge. External receiver samples data on the rising edge of the serial clock, so the transmitter must shift data out on the falling edge of the serial clock.
6	ASYNC			Transmit/receive operation asynchronous enable bit.
		SYNC	0	Synchronous. Transmit clock and frame sync provides the source for both the transmit and receive sections.
		ASYNC	1	Asynchronous. Separate clock and frame sync used by transmit and receive sections.
5	CLKXM			Transmit bit clock source bit.
		EXTERNAL	0	External transmit clock source from ACLKX pin.
		INTERNAL	1	Internal transmit clock source from output of programmable bit clock divider.
4–0	CLKXDIV	OF(*value*)		Transmit bit clock divide ratio bits determine the divide-down ratio from AHCLKX to ACLKX.
			0	Divide-by-1
			1h	Divide-by-2
			2h–1Fh	Divide-by-3 to divide-by-32

† For CSL implementation, use the notation MCASP_ACLKXCTL_*field_symval*

B.10.30 Transmit High-Frequency Clock Control Register (AHCLKXCTL)

The transmit high-frequency clock control register (AHCLKXCTL) configures the transmit high-frequency master clock (AHCLKX) and the transmit clock generator. The AHCLKXCTL is shown in Figure B–141 and described in Table B–151.

Figure B–141. Transmit High Frequency Clock Control Register (AHCLKXCTL) [Offset 00B4h]



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.

Table B–151. Transmit High-Frequency Clock Control Register (AHCLKXCTL) Field Descriptions

Bit	*field*	*symval*†	Value	Description
31–16	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
15	HCLKXM			Transmit high-frequency clock source bit.
		EXTERNAL	0	External transmit high-frequency clock source from AHCLKX pin.
		INTERNAL	1	Internal transmit high-frequency clock source from output of programmable high clock divider.

† For CSL implementation, use the notation MCASP_AHCLKXCTL_*field_symval*

Table B–151. Transmit High-Frequency Clock Control Register (AHCLKXCTL) Field Descriptions (Continued)

Bit	*field*	*symval*†	Value	Description
14	HCLKXP			Transmit bitstream high-frequency clock polarity select bit.
		RISING	0	Rising edge. AHCLKX is not inverted before programmable bit clock divider. In the special case where the transmit bit clock (ACLKX) is internally generated and the programmable bit clock divider is set to divide-by-1 (CLKXDIV = 0 in ACLKXCTL), AHCLKX is directly passed through to the ACLKX pin.
		FALLING	1	Falling edge. AHCLKX is inverted before programmable bit clock divider. In the special case where the transmit bit clock (ACLKX) is internally generated and the programmable bit clock divider is set to divide-by-1 (CLKXDIV = 0 in ACLKXCTL), AHCLKX is directly passed through to the ACLKX pin.
13–12	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
11–0	HCLKXDIV	OF(*value*)		Transmit high-frequency clock divide ratio bits determine the divide-down ratio from AUXCLK to AHCLKX.
			0	Divide-by-1
			1h	Divide-by-2
			2h–FFFh	Divide-by-3 to divide-by-4096

† For CSL implementation, use the notation MCASP_AHCLKXCTL_*field_symval*

B.10.31 Transmit TDM Time Slot Register (XTDM)

The transmit TDM time slot register (XTDM) specifies in which TDM time slot the transmitter is active. TDM time slot counter range is extended to 384 slots (to support SPDIF blocks of 384 subframes). XTDM operates modulo 32, that is, XTDMS specifies the TDM activity for time slots 0, 32, 64, 96, 128, etc. The XTDM is shown in Figure B–142 and described in Table B–152.

Figure B–142. Transmit TDM Time Slot Register (XTDM) [Offset 00B8h]

31	30	29	28	27	26	25	24
XTDMS31	XTDMS30	XTDMS29	XTDMS28	XTDMS27	XTDMS26	XTDMS25	XTDMS24
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
23	22	21	20	19	18	17	16
XTDMS23	XTDMS22	XTDMS21	XTDMS20	XTDMS19	XTDMS18	XTDMS17	XTDMS16
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
15	14	13	12	11	10	9	8
XTDMS15	XTDMS14	XTDMS13	XTDMS12	XTDMS11	XTDMS10	XTDMS9	XTDMS8
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
7	6	5	4	3	2	1	0
XTDMS7	XTDMS6	XTDMS5	XTDMS4	XTDMS3	XTDMS2	XTDMS1	XTDMS0
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

Legend: R/W = Read/Write; -*n* = value after reset

Table B–152. Transmit TDM Time Slot Register (XTDM) Field Descriptions

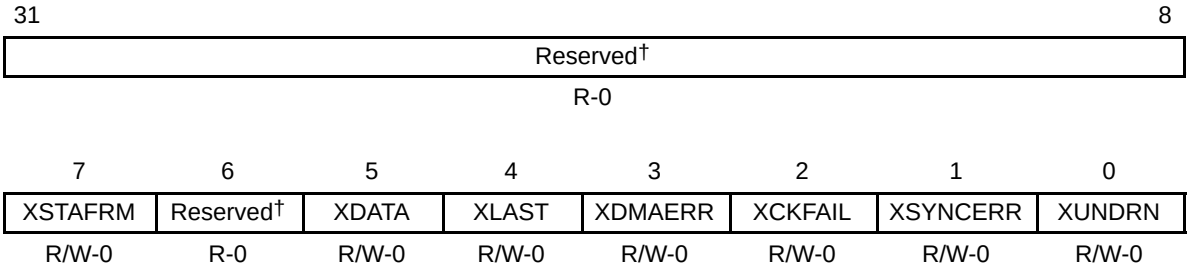
Bit	***field***†	***symval***†	Value	Description
31–0	XTDMS[31–0]			Transmitter mode during TDM time slot *n*.
		INACTIVE	0	Transmit TDM time slot *n* is inactive. The transmit serializer does not shift out data during this slot.
		ACTIVE	1	Transmit TDM time slot *n* is active. The transmit serializer shifts out data during this slot according to the serializer control register (SRCTL).

† For CSL implementation, use the notation MCASP_XTDM_XTDMS*n_symval*

B.10.32 Transmitter Interrupt Control Register (XINTCTL)

The transmitter interrupt control register (XINTCTL) controls generation of the McASP transmit interrupt (XINT). When the register bit(s) is set to 1, the occurrence of the enabled McASP condition(s) generates XINT. The XINTCTL is shown in Figure B–143 and described in Table B–153. See section B.10.33 for a description of the interrupt conditions.

Figure B–143. Transmitter Interrupt Control Register (XINTCTL) [Offset 00BCh]



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.

Table B–153. Transmitter Interrupt Control Register (XINTCTL) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31–8	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
7	XSTAFRM			Transmit start of frame interrupt enable bit.
		DISABLE	0	Interrupt is disabled. A transmit start of frame interrupt does not generate a McASP transmit interrupt (XINT).
		ENABLE	1	Interrupt is enabled. A transmit start of frame interrupt generates a McASP transmit interrupt (XINT).
6	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.

† For CSL implementation, use the notation MCASP_XINTCTL_*field_symval*

Table B–153. Transmitter Interrupt Control Register (XINTCTL) Field Descriptions (Continued)

Bit	*field*†	*symval*†	Value	Description
5	XDATA			Transmit data ready interrupt enable bit.
		DISABLE	0	Interrupt is disabled. A transmit data ready interrupt does not generate a McASP transmit interrupt (XINT).
		ENABLE	1	Interrupt is enabled. A transmit data ready interrupt generates a McASP transmit interrupt (XINT).
4	XLAST			Transmit last slot interrupt enable bit.
		DISABLE	0	Interrupt is disabled. A transmit last slot interrupt does not generate a McASP transmit interrupt (XINT).
		ENABLE	1	Interrupt is enabled. A transmit last slot interrupt generates a McASP transmit interrupt (XINT).
3	XDMAERR			Transmit EDMA error interrupt enable bit.
		DISABLE	0	Interrupt is disabled. A transmit EDMA error interrupt does not generate a McASP transmit interrupt (XINT).
		ENABLE	1	Interrupt is enabled. A transmit EDMA error interrupt generates a McASP transmit interrupt (XINT).
2	XCKFAIL			Transmit clock failure interrupt enable bit.
		DISABLE	0	Interrupt is disabled. A transmit clock failure interrupt does not generate a McASP transmit interrupt (XINT).
		ENABLE	1	Interrupt is enabled. A transmit clock failure interrupt generates a McASP transmit interrupt (XINT).
1	XSYNCERR			Unexpected transmit frame sync interrupt enable bit.
		DISABLE	0	Interrupt is disabled. An unexpected transmit frame sync interrupt does not generate a McASP transmit interrupt (XINT).
		ENABLE	1	Interrupt is enabled. An unexpected transmit frame sync interrupt generates a McASP transmit interrupt (XINT).
0	XUNDRN			Transmitter underrun interrupt enable bit.
		DISABLE	0	Interrupt is disabled.
				Interrupt is disabled. A transmitter underrun interrupt does not generate a McASP transmit interrupt (XINT).
		ENABLE	1	Interrupt is enabled. A transmitter underrun interrupt generates a McASP transmit interrupt (XINT).

† For CSL implementation, use the notation MCASP_XINTCTL_*field_symval*

B.10.33 Transmitter Status Register (XSTAT)

The transmitter status register (XSTAT) provides the transmitter status and transmit TDM time slot number. If the McASP logic attempts to set an interrupt flag in the same cycle that the CPU writes to the flag to clear it, the McASP logic has priority and the flag remains set. This also causes a new interrupt request to be generated. The XSTAT is shown in Figure B–144 and described in Table B–154.

Figure B–144. Transmitter Status Register (XSTAT) [Offset 00C0h]

31	9	8
Reserved†		XERR
R-0		R/W-0

7	6	5	4	3	2	1	0
XDMAERR	XSTAFRM	XDATA	XLAST	XTDMSLOT	XCKFAIL	XSYNCERR	XUNDRN
R/W-0	R/W-0	R/W-0	R/W-0	R-0	R/W-0	R/W-0	R/W-0

Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.

Table B–154. Transmitter Status Register (XSTAT) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31–9	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
8	XERR	OF(*value*)		XERR bit always returns a logic-OR of: XUNDRN | XSYNCERR | XCKFAIL | XDMAERR Allows a single bit to be checked to determine if a transmitter error interrupt has occurred.
			0	No errors have occurred.
			1	An error has occurred.

† For CSL implementation, use the notation MCASP_XSTAT_*field_symval*

Table B–154. Transmitter Status Register (XSTAT) Field Descriptions (Continued)

Bit	*field*†	*symval*†	Value	Description
7	XDMAERR	OF(*value*)		Transmit EDMA error flag. XDMAERR is set when the CPU or EDMA writes more serializers through the data port in a given time slot than were programmed as transmitters. Causes a transmit interrupt (XINT), if this bit is set and XDMAERR in XINTCTL is set. This bit is cleared by writing a 1 to this bit. Writing a 0 has no effect.
			0	Transmit EDMA error did not occur.
			1	Transmit EDMA error did occur.
6	XSTAFRM			Transmit start of frame flag. Causes a transmit interrupt (XINT), if this bit is set and XSTAFRM in XINTCTL is set. This bit is cleared by writing a 1 to this bit. Writing a 0 has no effect.
		NO	0	No new transmit frame sync (AFSX) is detected.
		YES	1	A new transmit frame sync (AFSX) is detected.
5	XDATA			Transmit data ready flag. Causes a transmit interrupt (XINT), if this bit is set and XDATA in XINTCTL is set. This bit is cleared by writing a 1 to this bit. Writing a 0 has no effect.
		NO	0	XBUF is written and is full.
		YES	1	Data is copied from XBUF to XRSR. XBUF is empty and ready to be written. XDATA is also set when the transmit serializers are taken out of reset. When XDATA is set, it always causes an EDMA event (AXEVT).
4	XLAST			Transmit last slot flag. XLAST is set along with XDATA, if the current slot is the last slot in a frame. Causes a transmit interrupt (XINT), if this bit is set and XLAST in XINTCTL is set. This bit is cleared by writing a 1 to this bit. Writing a 0 has no effect.
		NO	0	Current slot is not the last slot in a frame.
		YES	1	Current slot is the last slot in a frame. XDATA is also set.
3	XTDMSLOT	OF(*value*)		Returns the LSB of XSLOT. Allows a single read of XSTAT to determine whether the current TDM time slot is even or odd.
			0	Current TDM time slot is odd.
			1	Current TDM time slot is even.

† For CSL implementation, use the notation MCASP_XSTAT_*field_symval*

Table B–154. Transmitter Status Register (XSTAT) Field Descriptions (Continued)

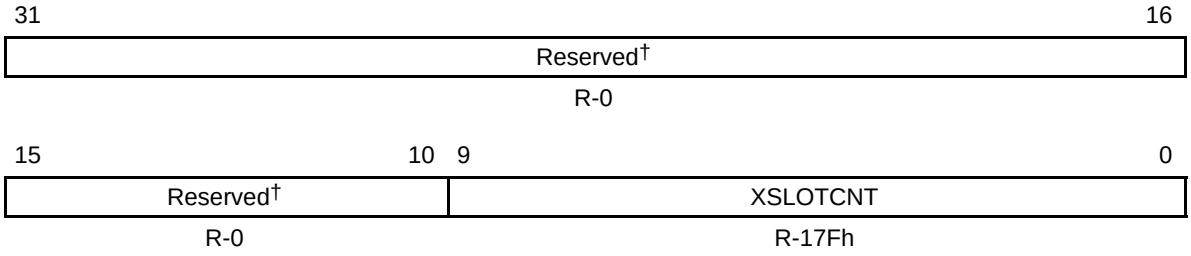
Bit	*field*†	*symval*†	Value	Description
2	XCKFAIL			Transmit clock failure flag. XCKFAIL is set when the transmit clock failure detection circuit reports an error. Causes a transmit interrupt (XINT), if this bit is set and XCKFAIL in XINTCTL is set. This bit is cleared by writing a 1 to this bit. Writing a 0 has no effect.
		NO	0	Transmit clock failure did not occur.
		YES	1	Transmit clock failure did occur.
1	XSYNCERR			Unexpected transmit frame sync flag. XSYNCERR is set when a new transmit frame sync (AFSX) occurs before it is expected. Causes a transmit interrupt (XINT), if this bit is set and XSYNCERR in XINTCTL is set. This bit is cleared by writing a 1 to this bit. Writing a 0 has no effect.
		NO	0	Unexpected transmit frame sync did not occur.
		YES	1	Unexpected transmit frame sync did occur.
0	XUNDRN			Transmitter underrun flag. XUNDRN is set when the transmit serializer is instructed to transfer data from XBUF to XRSR, but XBUF has not yet been serviced with new data since the last transfer. Causes a transmit interrupt (XINT), if this bit is set and XUNDRN in XINTCTL is set. This bit is cleared by writing a 1 to this bit. Writing a 0 has no effect.
		NO	0	Transmitter underrun did not occur.
		YES	1	Transmitter underrun did occur.

† For CSL implementation, use the notation MCASP_XSTAT_*field_symval*

B.10.34 Current Transmit TDM Time Slot Register (XSLOT)

The current transmit TDM time slot register (XSLOT) indicates the current time slot for the transmit data frame. The XSLOT is shown in Figure B–145 and described in Table B–155.

Figure B–145. Current Transmit TDM Time Slot Register (XSLOT) [Offset 00C4h]



Legend: R = Read only; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.

Table B–155. Current Transmit TDM Time Slot Register (XSLOT) Field Descriptions

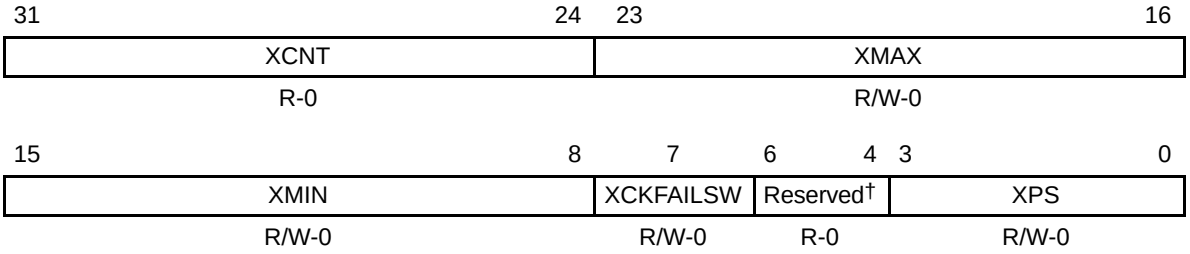
Bit	Field	*symval*†	Value	Description
31–10	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
9–0	XSLOTCNT	OF(*value*)	0–17Fh	Current transmit time slot count. Legal values: 0 to 383. During reset, this counter value is 383 so the next count value, which is used to encode the first DIT group of data, will be 0 and encodes the B preamble. TDM function is not supported for >32 time slots. However, TDM time slot counter may count to 383 when used to transmit a DIT block.

† For CSL implementation, use the notation MCASP_XSLOT_XSLOTCNT_*symval*

B.10.35 Transmit Clock Check Control Register (XCLKCHK)

The transmit clock check control register (XCLKCHK) configures the transmit clock failure detection circuit. The XCLKCHK is shown in Figure B–146 and described in Table B–156.

Figure B–146. Transmit Clock Check Control Register (XCLKCHK) [Offset 00C8h]



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.

Table B–156. Transmit Clock Check Control Register (XCLKCHK) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31–24	XCNT	OF(*value*)	0–FFh	Transmit clock count value (from previous measurement). The clock circuit continually counts the number of DSP system clocks for every 32 transmit high-frequency master clock (AHCLKX) signals, and stores the count in XCNT until the next measurement is taken.
23–16	XMAX	OF(*value*)	0–FFh	Transmit clock maximum boundary. This 8-bit unsigned value sets the maximum allowed boundary for the clock check counter after 32 transmit high-frequency master clock (AHCLKX) signals have been received. If the current counter value is greater than XMAX after counting 32 AHCLKX signals, XCKFAIL in XSTAT is set. The comparison is performed using unsigned arithmetic.
15–8	XMIN	OF(*value*)	0–FFh	Transmit clock minimum boundary. This 8-bit unsigned value sets the minimum allowed boundary for the clock check counter after 32 transmit high-frequency master clock (AHCLKX) signals have been received. If XCNT is less than XMIN after counting 32 AHCLKX signals, XCKFAIL in XSTAT is set. The comparison is performed using unsigned arithmetic.

† For CSL implementation, use the notation MCASP_XCLKCHK_*field_symval*

Table B–156. Transmit Clock Check Control Register (XCLKCHK) Field Descriptions (Continued)

Bit	*field*†	*symval*†	Value	Description
7	XCKFAILSW			Transmit clock failure detect autoswitch enable bit.
		DISABLE	0	Transmit clock failure detect autoswitch is disabled.
		ENABLE	1	Transmit clock failure detect autoswitch is enabled.
6–4	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
3–0	XPS			Transmit clock check prescaler value.
		DIVBY1	0	McASP system clock divided by 1
		DIVBY2	1h	McASP system clock divided by 2
		DIVBY4	2h	McASP system clock divided by 4
		DIVBY8	3h	McASP system clock divided by 8
		DIVBY16	4h	McASP system clock divided by 16
		DIVBY32	5h	McASP system clock divided by 32
		DIVBY64	6h	McASP system clock divided by 64
		DIVBY128	7h	McASP system clock divided by 128
		DIVBY256	8h	McASP system clock divided by 256
			9h–Fh	Reserved

† For CSL implementation, use the notation MCASP_XCLKCHK_*field_symval*

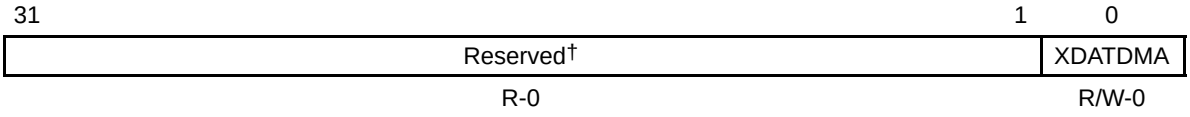
B.10.36 Transmitter DMA Event Control Register (XEVTCTL)

The transmitter DMA event control register (XEVTCTL) contains a disable bit for the transmit DMA event. The XEVTCTL is shown in Figure B–147 and described in Table B–157.

DSP specific registers

Accessing XEVTCTL not implemented on a specific DSP may cause improper device operation.

Figure B–147. Transmitter DMA Event Control Register (XEVTCTL) [Offset 00CCh]



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.

Table B–157. Transmitter DMA Event Control Register (XEVTCTL) Field Values

Bit	*field*	*symval*†	Value	Description
31–1	Reserved			Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
0	XDATDMA			Transmit data DMA request enable bit.
		ENABLE	0	Transmit data DMA request is enabled.
		DISABLE	1	Transmit data DMA request is disabled.

† For CSL implementation, use the notation MCASP_XEVTCTL_XDATDMA_*symval*

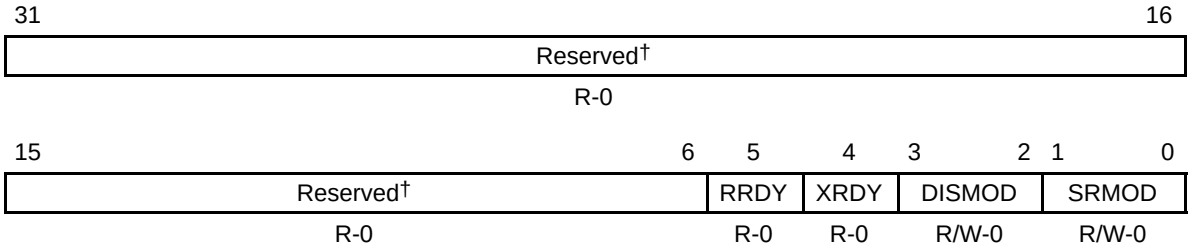
B.10.37 Serializer Control Registers (SRCTLn)

Each serializer on the McASP has a serializer control register (SRCTL). There are up to 16 serializers per McASP. The SRCTL is shown in Figure B–148 and described in Table B–158.

DSP specific registers

Accessing SRCTLn not implemented on a specific DSP may cause improper device operation.

Figure B–148. Serializer Control Registers (SRCTLn) [Offset 0180h–01BCh]



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.

Table B–158. Serializer Control Registers (SRCTLn) Field Descriptions

Bit	*field*	*symval*†	Value	Description
31–6	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
5	RRDY	OF(*value*)		Receive buffer ready bit. RRDY indicates the current receive buffer state. Always reads 0 when programmed as a transmitter or as inactive. If SRMOD bit is set to receive (2h), RRDY switches from 0 to 1 whenever data is transferred from XRSR to RBUF.
			0	Receive buffer (RBUF) is empty.
			1	Receive buffer (RBUF) contains data and needs to be read before the start of the next time slot or a receiver overrun occurs.

† For CSL implementation, use the notation MCASP_SRCTL_*field_symval*

Table B–158. Serializer Control Registers (SRCTLn) Field Descriptions (Continued)

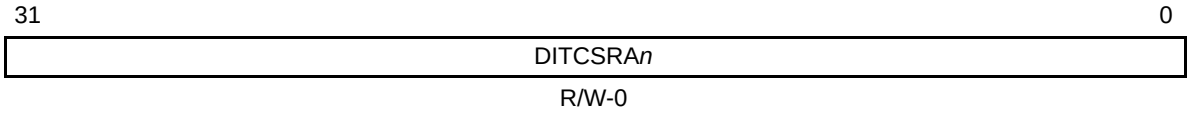
Bit	*field*	*symval*†	Value	Description
4	XRDY	OF(*value*)		Transmit buffer ready bit. XRDY indicates the current transmit buffer state. Always reads 0 when programmed as a receiver or as inactive. If SRMOD bit is set to transmit (1h), XRDY switches from 0 to 1 when XSRCLR in GBLCTL is switched from 0 to 1 to indicate an empty transmitter. XRDY remains set until XSRCLR is forced to 0, data is written to the corresponding transmit buffer, or SRMOD bit is changed to receive (2h) or inactive (0).
			0	Transmit buffer (XBUF) contains data.
			1	Transmit buffer (XBUF) is empty and needs to be written before the start of the next time slot or a transmit underrun occurs.
3–2	DISMOD			Serializer pin drive mode bit. Drive on pin when in inactive TDM slot of transmit mode or when serializer is inactive. This field only applies if the pin is configured as a McASP pin (PFUNC = 0).
		3STATE	0	Drive on pin is 3-state.
			1h	Reserved
		LOW	2h	Drive on pin is logic low.
		HIGH	3h	Drive on pin is logic high.
1–0	SRMOD			Serializer mode bit.
		INACTIVE	0	Serializer is inactive.
		XMT	1h	Serializer is transmitter.
		RCV	2h	Serializer is receiver.
			3h	Reserved

† For CSL implementation, use the notation MCASP_SRCTL_*field_symval*

B.10.38 DIT Left Channel Status Registers (DITCSRA0–DITCSRA5)

The DIT left channel status registers (DITCSRA) provide the status of each left channel (even TDM time slot). Each of the six 32-bit registers (Figure B–149) can store 192 bits of channel status data for a complete block of transmission. The DIT reuses the same data for the next block. It is your responsibility to update the register file in time, if a different set of data need to be sent.

Figure B–149. DIT Left Channel Status Registers (DITCSRA0–DITCSRA5) [Offset 0100h–0114h]

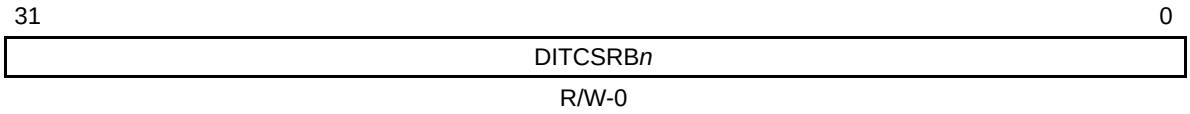


Legend: R/W = Read/Write; -*n* = value after reset

B.10.39 DIT Right Channel Status Registers (DITCSRB0–DITCSRB5)

The DIT right channel status registers (DITCSRB) provide the status of each right channel (odd TDM time slot). Each of the six 32-bit registers (Figure B–150) can store 192 bits of channel status data for a complete block of transmission. The DIT reuses the same data for the next block. It is your responsibility to update the register file in time, if a different set of data need to be sent.

Figure B–150. DIT Right Channel Status Registers (DITCSRB0–DITCSRB5) [Offset 0118h–012Ch]

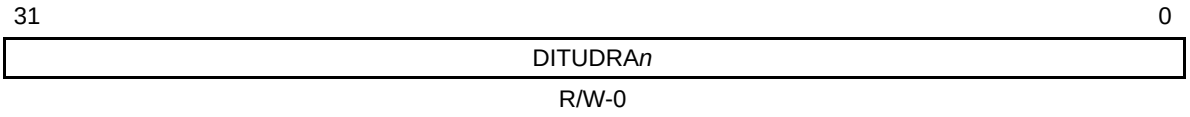


Legend: R/W = Read/Write; -*n* = value after reset

B.10.40 DIT Left Channel User Data Registers (DITUDRA0–DITUDRA5)

The DIT left channel user data registers (DITUDRA) provides the user data of each left channel (even TDM time slot). Each of the six 32-bit registers (Figure B–151) can store 192 bits of user data for a complete block of transmission. The DIT reuses the same data for the next block. It is your responsibility to update the register in time, if a different set of data need to be sent.

Figure B–151. DIT Left Channel User Data Registers (DITUDRA0–DITUDRA5) [Offset 0130h–0144h]

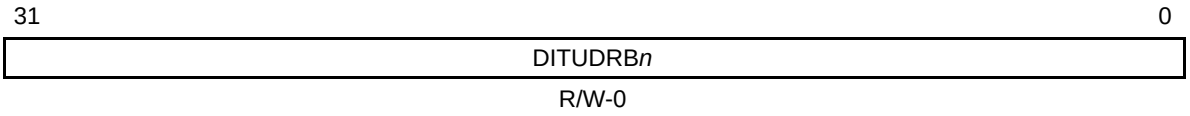


Legend: R/W = Read/Write; -*n* = value after reset

B.10.41 DIT Right Channel User Data Registers (DITUDRB0–DITUDRB5)

The DIT right channel user data registers (DITUDRB) provides the user data of each right channel (odd TDM time slot). Each of the six 32-bit registers (Figure B–152) can store 192 bits of user data for a complete block of transmission. The DIT reuses the same data for the next block. It is your responsibility to update the register in time, if a different set of data need to be sent.

Figure B–152. DIT Right Channel User Data Registers (DITUDRB0–DITUDRB5) [Offset 0148h–015Ch]



Legend: R/W = Read/Write; -*n* = value after reset

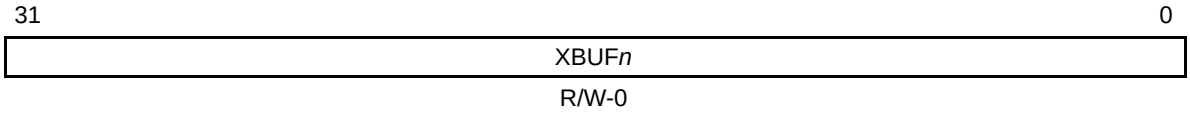
B.10.42 Transmit Buffer Registers (XBUFn)

The transmit buffers for the serializers (XBUF) hold data from the transmit format unit. For transmit operations, the XBUF (Figure B–153) is an alias of the XRBUF in the serializer. The XBUF can be accessed through the configuration bus (Table B–188) or through the data port (Table B–121).

DSP specific registers

Accessing XBUF registers not implemented on a specific DSP may cause improper device operation.

Figure B–153. Transmit Buffer Registers (XBUFn) [Offset 0200h–021Ch]



Legend: R/W = Read/Write; -*n* = value after reset

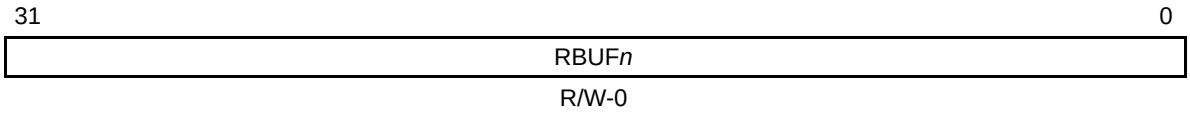
B.10.43 Receive Buffer Registers (RBUFn)

The receive buffers for the serializers (RBUF) hold data from the serializer before the data goes to the receive format unit. For receive operations, the RBUF (Figure B–154) is an alias of the XRBUF in the serializer. The RBUF can be accessed through the configuration bus (Table B–188) or through the data port (Table B–121).

DSP specific registers

Accessing RBUF registers not implemented on a specific DSP may cause improper device operation.

Figure B–154. Receive Buffer Registers (RBUFn) [Offset 0280h–02BCh]



Legend: R/W = Read/Write; -*n* = value after reset

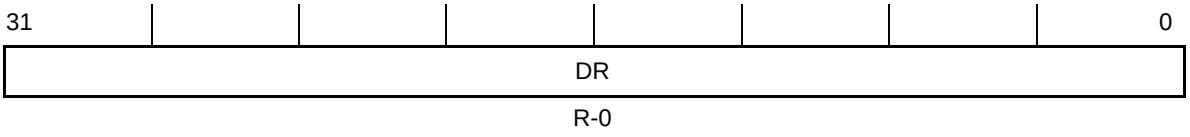
B.11 Multichannel Buffered Serial Port (MCBSP) Registers

Table B–159. McBSP Registers

Acronym	Register Name	Section
DRR	Data receive register	B.11.1
DXR	Data transmit register	B.11.2
SPCR	Serial port control register	B.11.3
PCR	Pin control register	B.11.4
RCR	Receive control register	B.11.5
XCR	Transmit control register	B.11.6
SRGR	Sample rate generator register	B.11.7
MCR	Multichannel control register	B.11.8
RCER	Receive channel enable register	B.11.9
XCER	Transmit channel enable register	B.11.10

B.11.1 Data Receive Register (DRR)

Figure B–155. Data Receive Register (DRR)



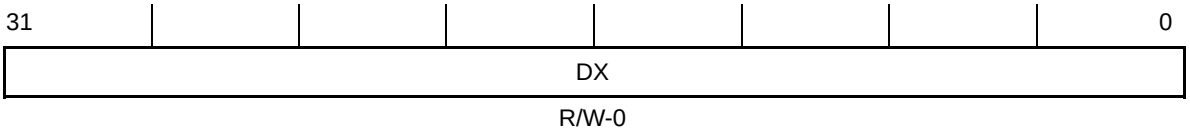
Legend: R/W-x = Read/Write-Reset value

Table B–160. Data Receive Register (DRR) Field Values (MCBSP_DRR_field_symval)

Bit	*field*	*symval*	Value	Description
31–0	DR	OF(*value*)	0–FFFF FFFFh	

B.11.2 Data Transmit Register (DXR)

Figure B–156. Data Transmit Register (DXR)



Legend: R/W-x = Read/Write-Reset value

Table B–161. Data Transmit Register (DXR) Field Values (MCBSP_DXR_field_symval)

Bit	*field*	*symval*	Value	Description
31–0	DX	OF(*value*)	0–FFFF FFFFh	

B.11.3 Serial Port Control Register (SPCR)

Figure B–157. Serial Port Control Register (SPCR)

31						26	25	24
Reserved							FREE†	SOFT†
R-0							R/W-0	R/W-0
23	22	21			20	19	18	17	16
FRST	GRST	XINTM				XSYNCERR‡	XEMPTY	XRDY	XRST
R/W-0	R/W-0	R/W-0				R/W-0	R-0	R-0	R/W-0
15	14			13	12	11	10					8
DLB	RJUST			CLKSTP		Reserved					
R/W-0	R/W-0			R/W-0		R-0					
7	6	5			4	3	2	1	0
DXENA†	Reserved	RINTM				RSYNCERR‡	RFULL	RRDY	RRST
R/W-0	R-0	R/W-0				R/W-0	R-0	R-0	R/W-0

† Available in the C621x/C671x/C64x only.

‡ Writing a 1 to XSYNCERR or RSYNCERR sets the error condition when the transmitter or receiver (XRST=1 or RRST=1), respectively, are enabled. Thus, it is used mainly for testing purposes or if this operation is desired.

Legend: R/W-x = Read/Write-Reset value

Table B–162. Serial Port Control Register (SPCR) Field Values (MCBSP_SPCR_field_symval)

Bit	*field*	*symval*	Value	Function
31–26	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
25	FREE			
		NO	0	Soft bit control
		YES	1	Clock running during emulation halt
24	SOFT			
		NO	0	Aborted transmissions

Table B–162. Serial Port Control Register (SPCR) Field Values (MCBSP_SPCR_field_symval) (Continued)

Bit	*field*	*symval*	Value	Function
		YES	1	Completed transmissions
23	FRST			Frame-sync generator reset.
		YES	0	Frame-synchronization logic is reset. Frame-sync signal (FSG) is not generated by the sample-rate generator.
		NO	1	Frame-sync signal (FSG) is generated after (FPER + 1) number of CLKG clocks; that is, all frame counters are loaded with their programmed values.
22	GRST			Sample-rate generator reset.
		YES	0	Sample-rate generator is reset.
		NO	1	Sample-rate generator is taken out of reset. CLKG is driven as per programmed value in sample-rate generator register (SRGR).
21–20	XINTM			Transmit interrupt (XINT) mode bit.
		XRDY	00	XINT is driven by XRDY (end-of-word) and end-of-frame in A-bis mode.
		EOS	01	XINT is generated by end-of-block or end-of-frame in multi-channel operation.
		FRM	10	XINT is generated by a new frame synchronization.
		XSYNCERR	11	XINT is generated by XSYNCERR.
19	XSYNCERR			Transmit synchronization error bit.
		NO	0	No synchronization error is detected.
		YES	1	Synchronization error is detected.
18	XEMPTY			Transmit shift register empty bit.
		YES	0	XSR is empty.
		NO	1	XSR is not empty.
17	XRDY			Transmitter ready bit.
		NO	0	Transmitter is not ready.
		YES	1	Transmitter is ready for new data in DXR.

Table B–162. Serial Port Control Register (SPCR) Field Values (MCBSP_SPCR_field_symval) (Continued)

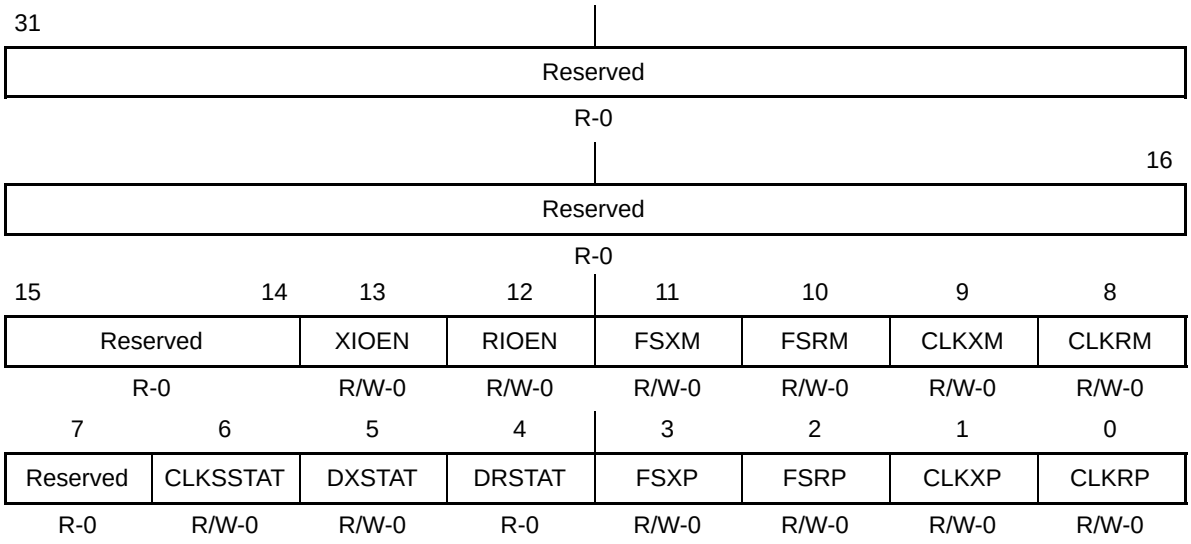
Bit	*field*	*symval*	Value	Function
16	XRST			Transmitter reset bit resets or enables the transmitter.
		YES	0	Serial port transmitter is disabled and in reset state.
		NO	1	Serial port transmitter is enabled.
15	DLB			Digital loop back mode enable bit.
		OFF	0	Digital loop back mode is disabled.
		ON	1	Digital loop back mode is enabled.
14–13	RJUST			Receive sign-extension and justification mode bit.
		RZF	00	Right-justify and zero-fill MSBs in DRR.
		RSE	01	Right-justify and sign-extend MSBs in DRR.
		LZF	10	Left-justify and zero-fill LSBs in DRR.
			11	Reserved
12–11	CLKSTP			Clock stop mode bit. In SPI mode, operates in conjunction with CLKXP bit of Pin Control Register (PCR).
		DISABLE	0x	Clock stop mode is disabled. Normal clocking for non-SPI mode.
				In SPI mode with data sampled on rising edge (CLKXP = 0):
		NODELAY	10	Clock starts with rising edge without delay.
		DELAY	11	Clock starts with rising edge with delay.
				In SPI mode with data sampled on falling edge (CLKXP = 1):
		NODELAY	10	Clock starts with falling edge without delay.
		DELAY	11	Clock starts with falling edge with delay.
10–8	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
7	DXENA			DX enabler bit.
		OFF	0	DX enabler is off.
		ON	1	DX enabler is on.

Table B–162. Serial Port Control Register (SPCR) Field Values (MCBSP_SPCR_field_symval) (Continued)

Bit	*field*	*symval*	Value	Function
6	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
5–4	RINTM			Receive interrupt (RINT) mode bit.
		RRDY	00	RINT is driven by RRDY (end-of-word) and end-of-frame in A-bis mode.
		EOS	01	RINT is generated by end-of-block or end-of-frame in multichannel operation.
		FRM	10	RINT is generated by a new frame synchronization.
		RSYNCERR	11	RINT is generated by RSYNCERR.
3	RSYNCERR			Receive synchronization error bit.
		NO	0	No synchronization error is detected.
		YES	1	Synchronization error is detected.
2	RFULL			Receive shift register full bit.
		NO	0	RBR is not in overrun condition.
		YES	1	DRR is not read, RBR is full, and RSR is also full with new word.
1	RRDY			Receiver ready bit.
		NO	0	Receiver is not ready.
		YES	1	Receiver is ready with data to be read from DRR.
0	RRST			Receiver reset bit resets or enables the receiver.
		YES	0	The serial port receiver is disabled and in reset state.
		NO	1	The serial port receiver is enabled.

B.11.4 Pin Control Register (PCR)

Figure B–158. Pin Control Register (PCR)



Legend: R/W-x = Read/Write-Reset value

Table B–163. Pin Control Register (PCR) Field Values (MCBSP_PCR_field_symval)

No.	*field*	*symval*	Value	Function
31–14	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
13	XIOEN			Transmit general-purpose I/O mode only when transmitter is disabled (XRST = 0 in SPCR).
		SP	0	DX, FSX, and CLKX pins are configured as serial port pins and do not function as general-purpose I/O pins.
		GPIO	1	DX pin is configured as general-purpose output pin; FSX and CLKX pins are configured as general-purpose I/O pins. These serial port pins do not perform serial port operations.
12	RIOEN			Receive general-purpose I/O mode only when receiver is disabled (RRST = 0 in SPCR).
		SP	0	DR, FSR, CLKR, and CLKS pins are configured as serial port pins and do not function as general-purpose I/O pins.
		GPIO	1	DR and CLKS pins are configured as general-purpose input pins; FSR and CLKR pins are configured as general-purpose I/O pins. These serial port pins do not perform serial port operations.

Table B–163. Pin Control Register (PCR) Field Values (MCBSP_PCR_field_symval)

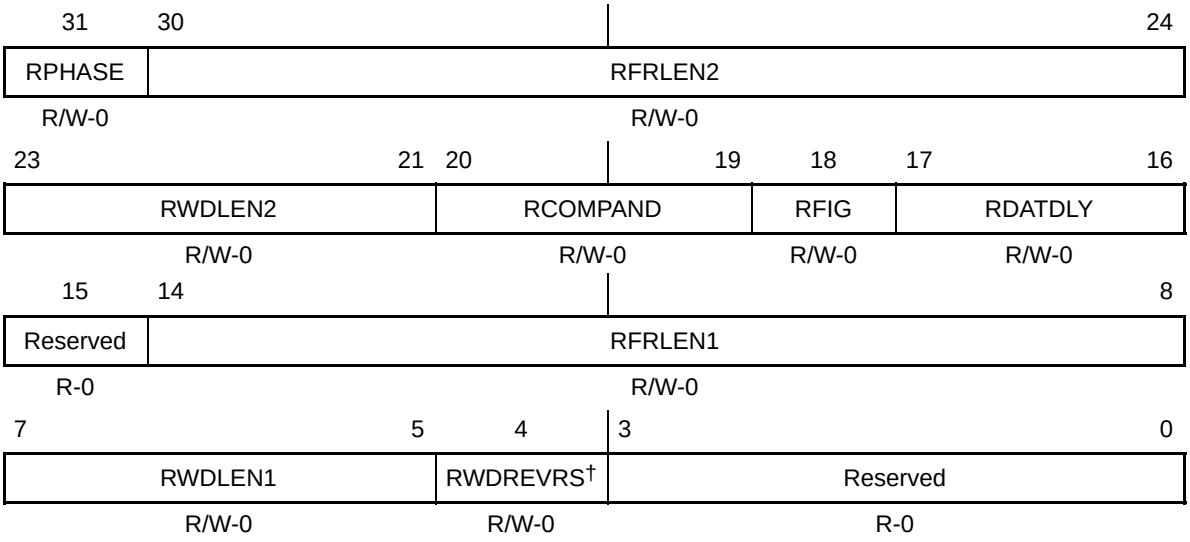
No.	*field*	*symval*	Value	Function
11	FSXM			Transmit frame-synchronization mode bit.
		EXTERNAL	0	Frame-synchronization signal is derived from an external source.
		INTERNAL	1	Frame-synchronization signal is determined by FSGM bit in SRGR.
10	FSRM			Receive frame-synchronization mode bit.
		EXTERNAL	0	Frame-synchronization signal is derived from an external source. FSR is an input pin.
		INTERNAL	1	Frame-synchronization signal is generated internally by the sample-rate generator. FSR is an output pin, except when GSYNC = 1 in SRGR.
9	CLKXM			Transmitter clock mode bit.
		INPUT	0	CLKX is an input pin and is driven by an external clock.
		OUTPUT	1	CLKX is an output pin and is driven by the internal sample-rate generator.
				In SPI mode when CLKSTP in SPCR is a non-zero value:
		INPUT	0	MCBSP is a slave and clock (CLKX) is driven by the SPI master in the system. CLKR is internally driven by CLKX.
		OUTPUT	1	MCBSP is a master and generates the clock (CLKX) to drive its receive clock (CLKR) and the shift clock of the SPI-compliant slaves in the system.
8	CLKRM			Receiver clock mode bit.
				Digital loop back mode is disabled (DLB = 0 in SPCR):
		INPUT	0	CLKR is an input pin and is driven by an external clock.
		OUTPUT	1	CLKR is an output pin and is driven by the internal sample-rate generator.
				Digital loop back mode is enabled (DLB = 1 in SPCR):
		INPUT	0	Receive clock (not the CLKR pin) is driven by transmit clock (CLKX) that is based on CLKXM bit. CLKR pin is in high-impedance state.
		OUTPUT	1	CLKR is an output pin and is driven by the transmit clock. The transmit clock is based on CLKXM bit.

Table B–163. Pin Control Register (PCR) Field Values (MCBSP_PCR_field_symval)

No.	*field*	*symval*	Value	Function
7	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
6	CLKSSTAT			CLKS pin status reflects value on CLKS pin when configured as a general-purpose input pin.
		0	0	CLKS pin reflects a logic low.
		1	1	CLKS pin reflects a logic high.
5	DXSTAT			DX pin status reflects value driven to DX pin when configured as a general-purpose output pin.
		0	0	DX pin reflects a logic low.
		1	1	DX pin reflects a logic high.
4	DRSTAT			DR pin status reflects value on DR pin when configured as a general-purpose input pin.
		0	0	DR pin reflects a logic low.
		1	1	DR pin reflects a logic high.
3	FSXP			Transmit frame-synchronization polarity bit.
		ACTIVEHIGH	0	Transmit frame-synchronization pulse is active high.
		ACTIVELOW	1	Transmit frame-synchronization pulse is active low.
2	FSRP			Receive frame-synchronization polarity bit.
		ACTIVEHIGH	0	Receive frame-synchronization pulse is active high.
		ACTIVELOW	1	Receive frame-synchronization pulse is active low.
1	CLKXP			Transmit clock polarity bit.
		RISING	0	Transmit data sampled on rising edge of CLKX.
		FALLING	1	Transmit data sampled on falling edge of CLKX.
0	CLKRP			Receive clock polarity bit.
		FALLING	0	Receive data sampled on falling edge of CLKR.
		RISING	1	Receive data sampled on rising edge of CLKR.

B.11.5 Receive Control Register (RCR)

Figure B–159. Receive Control Register (RCR)



Legend: R/W-x = Read/Write-Reset value

Table B–164. Receive Control Register (RCR) Field Values (MCBSP_RCR_field_symval)

Bit	*field*	*symval*	Value	Function
31	RPHASE			Receive phases bit.
		SINGLE	0	Single-phase frame
		DUAL	1	Dual-phase frame
30–24	RFRLEN2	OF(*value*)	0–127	Specifies the number of words (length) in the receive frame.
23–21	RWDLEN2			Specifies the number of bits (length) in the receive word.
		8BIT	000	Receive word length is 8 bits.
		12BIT	001	Receive word length is 12 bits.
		16BIT	010	Receive word length is 16 bits.
		20BIT	011	Receive word length is 20 bits.
		24BIT	100	Receive word length is 24 bits.
		32BIT	101	Receive word length is 32 bits.
			110	Reserved
			111	Reserved

Table B–164. Receive Control Register (RCR) Field Values (MCBSP_RCR_field_symval)

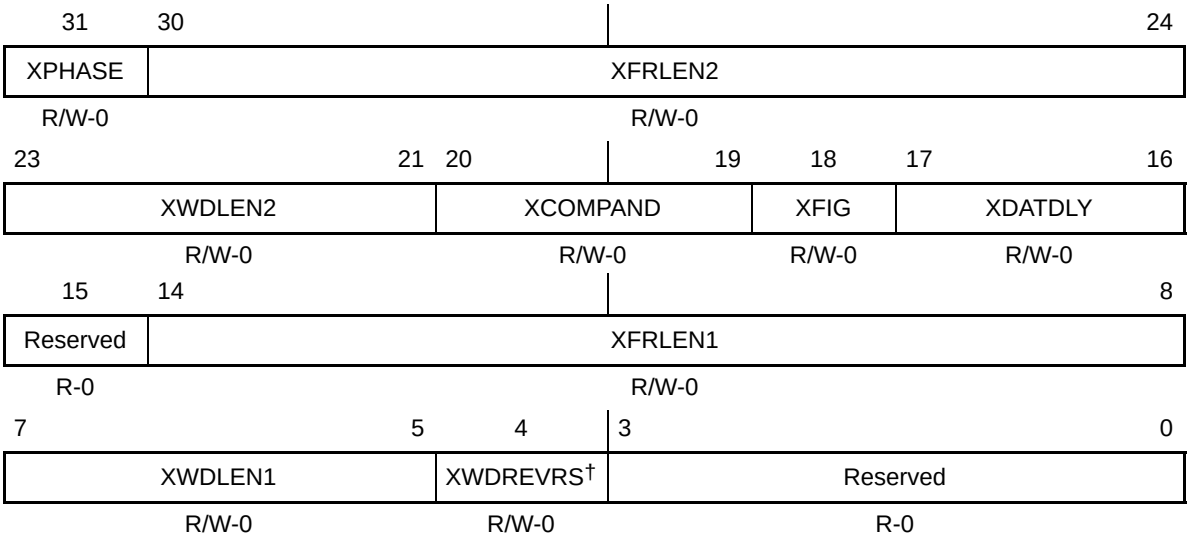
Bit	*field*	*symval*	Value	Function
20–19	RCOMPAND			Receive companding mode. Modes other than 00 are only enabled when RWDLEN[1, 2] bit is 000 (indicating 8-bit data).
		MSB	00	No companding, data transfer starts with MSB first.
		8BITLSB	01	No companding, 8-bit data transfer starts with LSB first.
		ULAW	10	Compand using μ-law for receive data.
		ALAW	11	Compand using A-law for receive data.
18	RFIG			Receive frame ignore bit.
		NO	0	Receive frame-synchronization pulses after the first pulse restarts the transfer.
		YES	1	Receive frame-synchronization pulses after the first pulse are ignored.
17–16	RDATDLY			Receive data delay bit.
		0BIT	00	0-bit data delay
		1BIT	01	1-bit data delay
		2BIT	10	2-bit data delay
			11	Reserved
15	Reserved			
14–8	RFRLEN1	OF(*value*)	0–127	Specifies the number of words (length) in the receive frame.
7–5	RWDLEN1			Specifies the number of bits (length) in the receive word.
		8BIT	000	Receive word length is 8 bits.
		12BIT	001	Receive word length is 12 bits.
		16BIT	010	Receive word length is 16 bits.
		20BIT	011	Receive word length is 20 bits.
		24BIT	100	Receive word length is 24 bits.
		32BIT	101	Receive word length is 32 bits.
			110	Reserved
			111	Reserved

Table B–164. Receive Control Register (RCR) Field Values (MCBSP_RCR_field_symval)

Bit	*field*	*symval*	Value	Function
4	RWDREVRS			Receive 32-bit bit reversal enable bit.
		DISABLE	0	32-bit reversal disabled
		ENABLE	1	32-bit reversal enabled. 32-bit data is received LSB first. RWDLEN should be set for 32-bit operation; RCOMPAND should be set to 01b; else operation is undefined
3–0	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.

B.11.6 Transmit Control Register (XCR)

Figure B–160. Transmit Control Register (XCR)



Legend: R/W-x = Read/Write-Reset value

Table B–165. Transmit Control Register (XCR) Field Values (MCBSP_XCR_field_symval)

Bit	*field*	*symval*	Value	Description
31	XPHASE			Transmit phases bit.
		SINGLE	0	Single-phase frame
		DUAL	1	Dual-phase frame
30–24	XFRLEN2	OF(*value*)	0–127	Specifies the number of words (length) in the transmit frame.

Table B–165. Transmit Control Register (XCR) Field Values (MCBSP_XCR_field_symval)

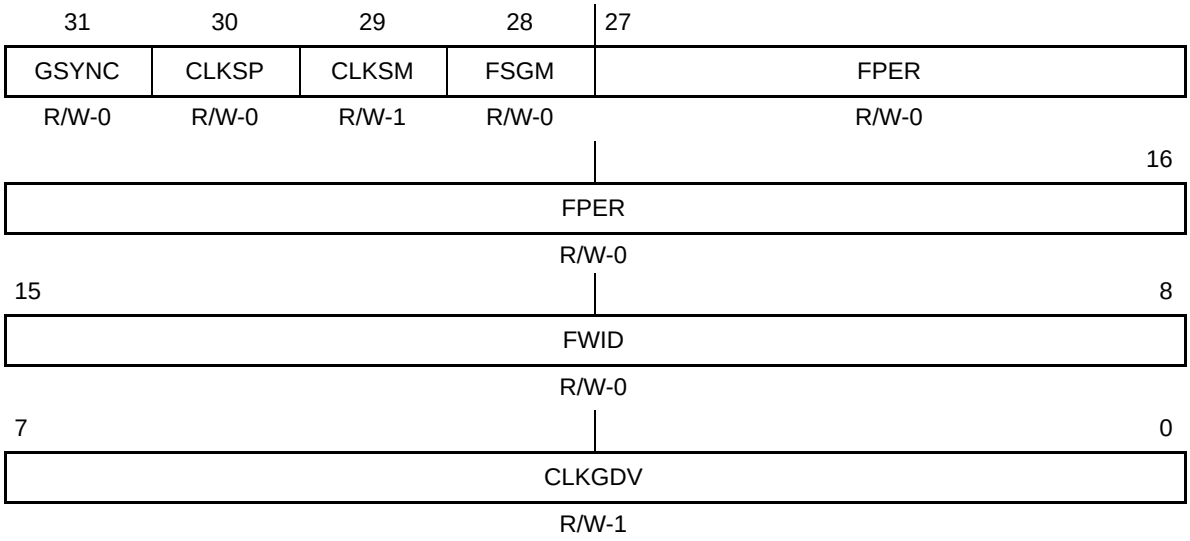
Bit	*field*	*symval*	Value	Description
23–21	XWDLEN2			Specifies the number of bits (length) in the transmit word.
		8BIT	000	Transmit word length is 8 bits.
		12BIT	001	Transmit word length is 12 bits.
		16BIT	010	Transmit word length is 16 bits.
		20BIT	011	Transmit word length is 20 bits.
		24BIT	100	Transmit word length is 24 bits.
		32BIT	101	Transmit word length is 32 bits.
			110	Reserved
			111	Reserved
20–19	XCOMPAND			Transmit companding mode. Modes other than 00 are only enabled when XWDLEN[1, 2] bit is 000 (indicating 8-bit data).
		MSB	00	No companding, data transfer starts with MSB first.
		8BITLSB	01	No companding, 8-bit data transfer starts with LSB first.
		ULAW	10	Compand using μ-law for transmit data.
		ALAW	11	Compand using A-law for transmit data.
18	XFIG			Transmit frame ignore bit.
		NO	0	Transmit frame-synchronization pulses after the first pulse restarts the transfer.
		YES	1	Transmit frame-synchronization pulses after the first pulse are ignored.
17–16	XDATDLY			Transmit data delay bit.
		0BIT	00	0-bit data delay
		1BIT	01	1-bit data delay
		2BIT	10	2-bit data delay
			11	Reserved
15	Reserved			

Table B–165. Transmit Control Register (XCR) Field Values (MCBSP_XCR_field_symval)

Bit	*field*	*symval*	Value	Description
14–8	XFRLEN1	OF(*value*)	0–127	Specifies the number of words (length) in the transmit frame.
7–5	XWDLEN1			Specifies the number of bits (length) in the transmit word.
		8BIT	000	Transmit word length is 8 bits.
		12BIT	001	Transmit word length is 12 bits.
		16BIT	010	Transmit word length is 16 bits.
		20BIT	011	Transmit word length is 20 bits.
		24BIT	100	Transmit word length is 24 bits.
		32BIT	101	Transmit word length is 32 bits.
			110	Reserved
			111	Reserved
4	XWDREVRS			Transmit 32-bit bit reversal feature enable bit.
		DISABLE	0	32-bit reversal disabled
		ENABLE	1	32-bit reversal enabled. 32-bit data is transmitted LSB first. XWDLEN should be set for 32-bit operation; XCOMPAND should be set to 01b; else operation is undefined.
3–0	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.

B.11.7 Sample Rate Generator Register (SRGR)

Figure B–161. Sample Rate Generator Register (SRGR)



Legend: R/W-x = Read/Write-Reset value

Table B–166. Sample Rate Generator Register (SRGR) Field Values (MCBSP_SRGR_field_symval)

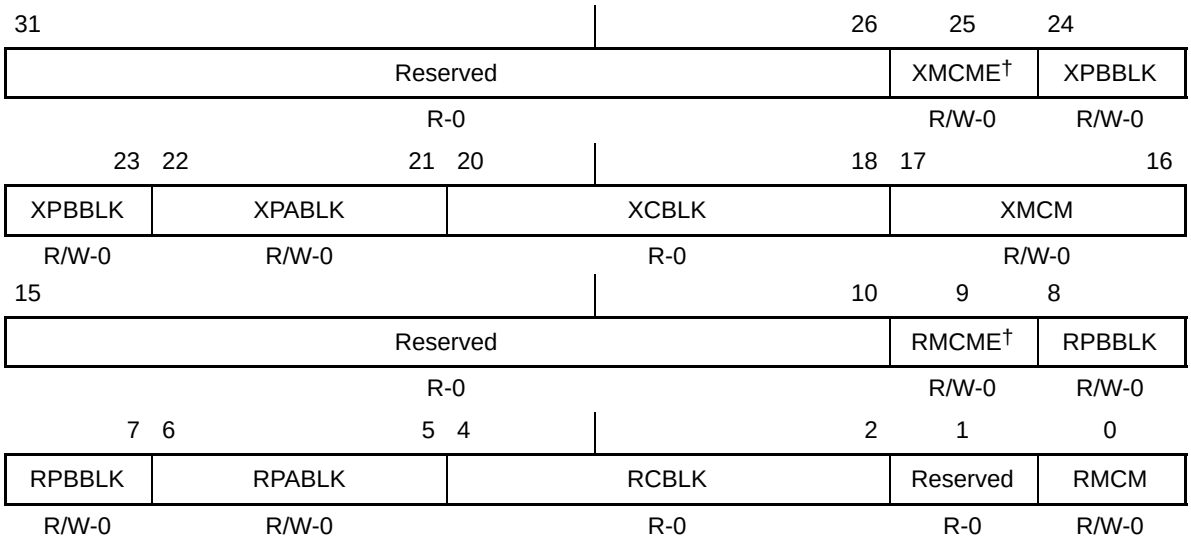
Bit	*field*	*symval*	Value	Description
31	GSYNC			Sample-rate generator clock synchronization bit only used when the external clock (CLKS) drives the sample-rate generator clock (CLKSM = 0).
		FREE	0	The sample-rate generator clock (CLKG) is free running.
		SYNC	1	The sample-rate generator clock (CLKG) is running; however, CLKG is resynchronized and frame-sync signal (FSG) is generated only after detecting the receive frame-synchronization signal (FSR). Also, frame period (FPER) is a don't care because the period is dictated by the external frame-sync pulse.
30	CLKSP			CLKS polarity clock edge select bit only used when the external clock (CLKS) drives the sample-rate generator clock (CLKSM = 0).
		RISING	0	Rising edge of CLKS generates CLKG and FSG.
		FALLING	1	Falling edge of CLKS generates CLKG and FSG.

Table B–166. Sample Rate Generator Register (SRGR) Field Values (MCBSP_SRGR_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
29	CLKSM			MCBSP sample-rate generator clock mode bit.
		CLKS	0	Sample-rate generator clock derived from the CLKS pin.
		INTERNAL	1	Sample-rate generator clock derived from CPU clock.
28	FSGM			Sample-rate generator transmit frame-synchronization mode bit used when FSXM = 1 in PCR.
		DXR2XSR	0	Transmit frame-sync signal (FSX) due to DXR-to-XSR copy. When FSGM = 0, FWID bit and FPER bit are ignored.
		FSG	1	Transmit frame-sync signal (FSX) driven by the sample-rate generator frame-sync signal (FSG).
27–16	FPER	OF(*value*)	0–4095	The value plus 1 specifies when the next frame-sync signal becomes active. Range: 1 to 4096 sample-rate generator clock (CLKG) periods.
15–8	FWID	OF(*value*)	0–255	The value plus 1 specifies the width of the frame-sync pulse (FSG) during its active period.
7–0	CLKGDV	OF(*value*)	0–255	The value is used as the divide-down number to generate the required sample-rate generator clock frequency.

B.11.8 Multichannel Control Register (MCR)

Figure B–162. Multichannel Control Register (MCR)



† XMCME and RMCME are only available on C64x devices. These bit fields are Reserved (R-0) on all other C6000 devices.

Legend: R/W-x = Read/Write-Reset value

Table B–167. Multichannel Control Register (MCR) Field Values (MCBSP_MCR_field_symval)

Bit	*field*	*symval*	Value	Description
31–26	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
25	XMCME			**For devices with 128-channel selection capability:** Transmit 128-channel selection enable bit.
		NORMAL	0	Normal 32-channel selection is enabled.
		ENHANCED	1	Six additional registers (XCERC–XCERH) are used to enable 128-channel selection.

† DX is masked or driven to a high-impedance state during (a) interpacket intervals, (b) when a channel is masked regardless of whether it is enabled, or (c) when a channel is disabled.

Table B–167. Multichannel Control Register (MCR) Field Values (MCBSP_MCR_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
24–23	XPBBLK			Transmit partition B block bit. Enables 16 contiguous channels in each block.
		SF1	00	Block 1. Channel 16 to channel 31
		SF3	01	Block 3. Channel 48 to channel 63
		SF5	10	Block 5. Channel 80 to channel 95
		SF7	11	Block 7. Channel 112 to channel 127
22–21	XPABLK			Transmit partition A block bit. Enables 16 contiguous channels in each block.
		SF0	00	Block 0. Channel 0 to channel 15
		SF2	01	Block 2. Channel 32 to channel 47
		SF4	10	Block 4. Channel 64 to channel 79
		SF6	11	Block 6. Channel 96 to channel 111
20–18	XCBLK			Transmit current block bit.
		SF0	000	Block 0. Channel 0 to channel 15
		SF1	001	Block 1. Channel 16 to channel 31
		SF2	010	Block 2. Channel 32 to channel 47
		SF3	011	Block 3. Channel 48 to channel 63
		SF4	100	Block 4. Channel 64 to channel 79
		SF5	101	Block 5. Channel 80 to channel 95
		SF6	110	Block 6. Channel 96 to channel 111
		SF7	111	Block 7. Channel 112 to channel 127

† DX is masked or driven to a high-impedance state during (a) interpacket intervals, (b) when a channel is masked regardless of whether it is enabled, or (c) when a channel is disabled.

Table B–167. Multichannel Control Register (MCR) Field Values (MCBSP_MCR_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
17–16	XMCM			Transmit multichannel selection enable bit.
		ENNOMASK	00	All channels enabled without masking (DX is always driven during transmission of data†).
		DISXP	01	All channels disabled and, therefore, masked by default. Required channels are selected by enabling XP[A, B]BLK and XCER[A, B] appropriately. Also, these selected channels are not masked and, therefore, DX is always driven.
		ENMASK	10	All channels enabled, but masked. Selected channels enabled using XP[A, B]BLK and XCER[A, B] are unmasked.
		DISRP	11	All channels disabled and, therefore, masked by default. Required channels are selected by enabling RP[A, B]BLK and RCER[A, B] appropriately. Selected channels can be unmasked by RP[A, B]BLK and XCER[A, B]. This mode is used for symmetric transmit and receive operation.
15–10	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
9	RMCME			**For devices with 128-channel selection capability:** Receive 128-channel selection enable bit.
		NORMAL	0	Normal 32-channel selection is enabled.
		ENHANCED	1	Six additional registers (RCERC–RCERH) are used to enable 128-channel selection.
8–7	RPBBLK			Receive partition B block bit. Enables 16 contiguous channels in each block.
		SF1	00	Block 1. Channel 16 to channel 31
		SF3	01	Block 3. Channel 48 to channel 63
		SF5	10	Block 5. Channel 80 to channel 95
		SF7	11	Block 7. Channel 112 to channel 127

† DX is masked or driven to a high-impedance state during (a) interpacket intervals, (b) when a channel is masked regardless of whether it is enabled, or (c) when a channel is disabled.

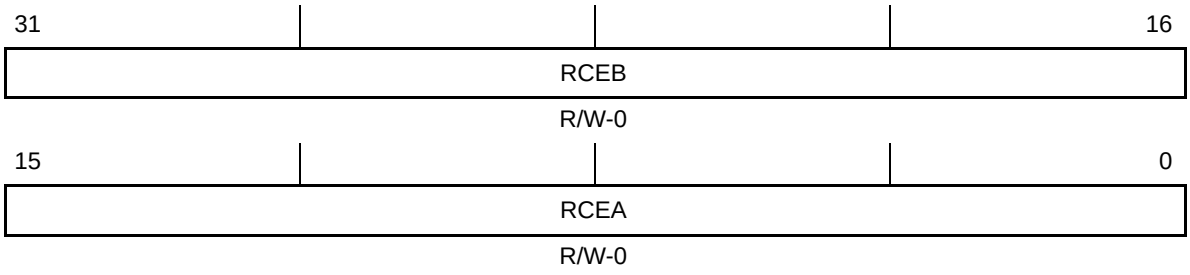
Table B–167. Multichannel Control Register (MCR) Field Values (MCBSP_MCR_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
6–5	RPABLK			Receive partition A block bit. Enables 16 contiguous channels in each block.
		SF0	00	Block 0. Channel 0 to channel 15
		SF2	01	Block 2. Channel 32 to channel 47
		SF4	10	Block 4. Channel 64 to channel 79
		SF6	11	Block 6. Channel 96 to channel 111
4–2	RCBLK			Receive current block bit.
		SF0	000	Block 0. Channel 0 to channel 15
		SF1	001	Block 1. Channel 16 to channel 31
		SF2	010	Block 2. Channel 32 to channel 47
		SF3	011	Block 3. Channel 48 to channel 63
		SF4	100	Block 4. Channel 64 to channel 79
		SF5	101	Block 5. Channel 80 to channel 95
		SF6	110	Block 6. Channel 96 to channel 111
		SF7	111	Block 7. Channel 112 to channel 127
1	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
0	RMCM			Receive multichannel selection enable bit.
		CHENABLE	0	All 128 channels enabled.
		ELDISABLE	1	All channels disabled by default. Required channels are selected by enabling RP[A, B]BLK and RCER[A, B] appropriately.

† DX is masked or driven to a high-impedance state during (a) interpacket intervals, (b) when a channel is masked regardless of whether it is enabled, or (c) when a channel is disabled.

B.11.9 Receive Channel Enable Register (RCER)

Figure B–163. Receive Channel Enable Register (RCER)



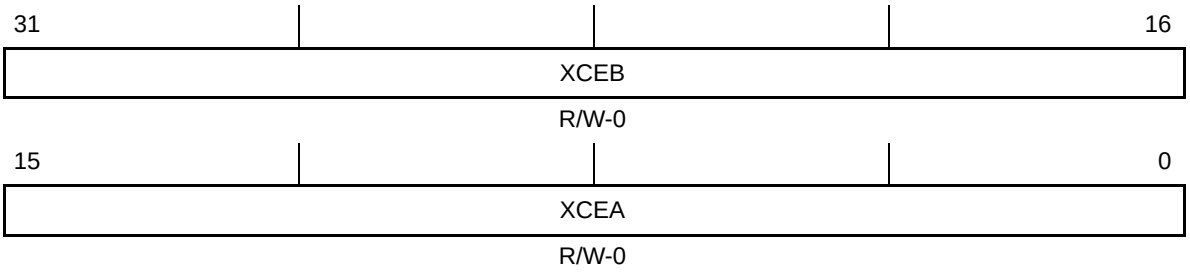
Legend: R/W-x = Read/Write-Reset value

Table B–168. Receive Channel Enable Register (RCER) Field Values (MCBSP_RCER_field_symval)

Bit	*field*	*symval*	Value	Description
31–16	RCEB	OF(*value*)	0–FFFFh	A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) reception of the *n*th channel within the 16-channel-wide block in partition B. The 16-channel-wide block is selected by the RPBBLK bit in MCR.
15–0	RCEA	OF(*value*)	0–FFFFh	A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) reception of the *n*th channel within the 16-channel-wide block in partition A. The 16-channel-wide block is selected by the RPABLK bit in MCR.

B.11.10 Transmit Channel Enable Register (XCER)

Figure B–164. Transmit Channel Enable Register (XCER)



Legend: R/W-x = Read/Write-Reset value

Table B–169. Transmit Channel Enable Register (XCER) Field Values (MCBSP_XCER_field_symval)

Bit	*field*	*symval*	Value	Description
31–16	XCEB	OF(*value*)	0–FFFFh	A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) transmission of the *n*th channel within the 16-channel-wide block in partition B. The 16-channel-wide block is selected by the XPBBLK bit in MCR.
15–0	XCEA	OF(*value*)	0–FFFFh	A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) transmission of the *n*th channel within the 16-channel-wide block in partition A. The 16-channel-wide block is selected by the XPABLK bit in MCR.

B.12 Peripheral Component Interconnect (PCI) Registers

Table B–170. PCI Memory-Mapped Registers

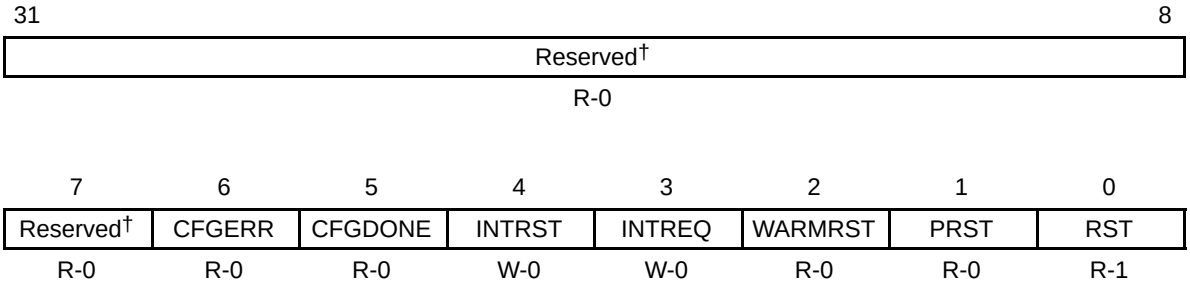
Acronym	Register Name	Section
RSTSRC	DSP reset source/status register	B.12.1
PMDCSR†	Power management DSP control/status register	B.12.2
PCIIS	PCI interrupt source register	B.12.3
PCIIEN	PCI interrupt enable register	B.12.4
DSPMA	DSP master address register	B.12.5
PCIMA	PCI master address register	B.12.6
PCIMC	PCI master control register	B.12.7
CDSPA	Current DSP address register	B.12.8
CPCIA	Current PCI address register	B.12.9
CCNT	Current byte count register	B.12.10
EEADD	EEPROM address register	B.12.11
EEDAT	EEPROM data register	B.12.12
EECTL	EEPROM control register	B.12.13
HALT1	PCI transfer halt register	B.12.14
TRCTL‡	PCI transfer request control register	B.12.15

† This register only applies to C62x/C67x DSP.
‡ TRCTL register only applies to C64x DSP.

B.12.1 DSP Reset Source/Status Register (RSTSRC)

The DSP reset source/status register (RSTSRC) shows the reset status of the DSP. It gives the DSP visibility to which reset source caused the last reset. The RSTSRC is shown in Figure B–165 and described in Table B–171. The RST, PRST, and WARMRST bits are cleared by a read of RSTSRC.

Figure B–165. DSP Reset Source/Status Register (RSTSRC)



Legend: R = Read only; W = Write only; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.

Table B–171. DSP Reset Source/Status Register (RSTSRC) Field Descriptions

Bits	***field***†	***symval***†	Value	Description
31–7	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
6	CFGERR	OF(*value*)		Configuration error bit. An error occurred when trying to load the configuration registers from EEPROM (Checksum failure). Read-only bit, writes have no effect.
			0	No configuration error.
			1	Checksum error during EEPROM autoinitialization.
5	CFGDONE	OF(*value*)		Configuration hold bit. EEPROM has finished loading the PCI configuration registers. Read-only bit, writes have no effect.
			0	Configuration registers have not been loaded.
			1	Configuration registers load from EEPROM is complete.

† For CSL implementation, use the notation PCI_RSTSRC_*field_symval*

Table B–171. DSP Reset Source/Status Register (RSTSRC) Field Descriptions (Continued)

Bits	*field*†	*symval*†	Value	Description
4	INTRST			$\overline{\text{PINTA}}$ reset bit. This bit must be asserted before another host interrupt can be generated. Write-only bit, reads return 0.
		NO	0	Writes of 0 have no effect.
		YES	1	When a 1 is written to this bit, $\overline{\text{PINTA}}$ is deasserted and the interrupt logic is reset to enable future interrupts.
3	INTREQ			Request a DSP-to-PCI interrupt when written with a 1. Write-only bit, reads return 0.
		NO	0	Writes of 0 have no effect.
		YES	1	Causes assertion of $\overline{\text{PINTA}}$ if the INTAM bit in the host status register (HSR) is 0.
2	WARMRST	OF(*value*)		A host software reset of the DSP or a power management warm reset occurred since the last RSTSRC read or last $\overline{\text{RESET}}$. Read-only bit, writes have no effect. This bit is set by a host write of 0 to the WARMRESET bit in the host-to-DSP control register (HDCR) or a power management request from D2 or D3. Cleared by a read of RSTSRC or $\overline{\text{RESET}}$ assertion.
			0	No warm reset since last RSTSRC read or $\overline{\text{RESET}}$.
			1	Warm reset since last RSTSRC read or $\overline{\text{RESET}}$.
1	PRST	OF(*value*)		Indicates occurrence of a $\overline{\text{PRST}}$ reset since the last RSTSRC read or $\overline{\text{RESET}}$ assertion. Read-only bit, writes have no effect. Cleared by a read of RSTSRC or $\overline{\text{RESET}}$ active. When PRST is held active (low), this bit always reads as 1.
			0	No $\overline{\text{PRST}}$ reset since last RSTSRC read.
			1	$\overline{\text{PRST}}$ reset has occurred since last RSTSRC read.
0	RST	OF(*value*)		Indicates a device reset ($\overline{\text{RESET}}$) occurred since the last RSTSRC read. Read-only bit, writes have no effect. Cleared by a read of RSTSRC.
			0	No device reset ($\overline{\text{RESET}}$) since last RSTSRC read
			1	Device reset ($\overline{\text{RESET}}$) has occurred since last RSTSRC read

† For CSL implementation, use the notation PCI_RSTSRC_*field_symval*

B.12.2 Power Management DSP Control/Status Register (PMDCSR) (C62x/C67x DSP only)

The power management DSP control/status register (PMDCSR) allows power management control. The PMDCSR is shown in Figure B–166 and described in Table B–172.

B.12.2.1 3.3 V_{aux} Presence Detect Status Bit (AUXDETECT)

The 3.3 V_{aux}DET pin is used to indicate the presence of 3.3 V_{aux} when V_{DDcore} is removed. The DSP can monitor this pin by reading the AUXDETECT bit in PMDCSR. The PMEEN bit in the power management control/status register (PMCSR) is held clear by the 3.3 V_{aux}DET pin being low.

B.12.2.2 PCI Port Response to $\overline{\text{PWR\_WKP}}$ and PME Generation

The PCI port responds differently to an active $\overline{\text{PWR\_WKP}}$ input, depending on whether V_{DDcore} is alive when 3.3 V_{aux} is alive. The PCI port response to $\overline{\text{PWR\_WKP}}$ is powered by 3.3 V_{aux}.

When V_{DDcore} is alive and 3.3 V_{aux} is alive (that is, all device power states but $D3_{cold}$), bits are set in the PCI interrupt source register (PCIIS) for the detection of the $\overline{\text{PWR\_WKP}}$ high-to-low and low-to-high transition. The $\overline{\text{PWR\_WKP}}$ signal is directly connected to the DSP PCI_WAKEUP interrupt.

When V_{DDcore} is shut down and 3.3 V_{aux} is alive (in $D3_{cold}$), a $\overline{\text{PWR\_WKP}}$ transition causes the PMESTAT bit in PMCSR to be set (regardless of the PMEEN bit value). If the PMEEN bit is set, $\overline{\text{PWR\_WKP}}$ activity also causes the PME pin to be asserted and held active.

The PCI port can also generate PME depending on the HWPMECTL bits in PMDCSR. PME can be generated from any state or on transition to any state on an active $\overline{\text{PWR\_WKP}}$ signal, if the corresponding bit in the HWPMECTL bits is set.

Transitions on the $\overline{\text{PWR\_WKP}}$ pin can cause a CPU interrupt (PCI_WAKEUP). The PWRHL and PWRLH bits in PCIIS indicate a high-to-low or low-to-high transition on the $\overline{\text{PWR\_WKP}}$ pin. If the corresponding interrupts are enabled in the PCI interrupt enable register (PCIIEN), a PCI_WAKEUP interrupt is generated to the CPU.

If 3.3 V_{aux} is not powered, the PME pin is in a high-impedance state. Once PME is driven active by the DSP, it is only deasserted when the PMESTAT bit in PMCSR is written with a 1 or the PMEEN bit is written with a 0. Neither $\overline{\text{PRST}}$, $\overline{\text{RESET}}$, or warm reset active can cause PME to go into a high-impedance state if it was already asserted before the reset.

Figure B–166. Power Management DSP Control/Status Register (PMDCSR)

31–19	18–11	10	9	8
Reserved†	HWPMECTL	D3WARMONWKP	D2WARMONWKP	PMEEN
R-0	R/W-1000 1000	R-x	R-x	R/W-x

7	6	5	4	3–2	1–0
PWRWKP	PMESTAT	PMEDRVN	AUXDETECT	CURSTATE	REQSTATE
R-x	R-x/W-0	R-0	R-x	R/W-0	R-0

Legend: R = Read only; R/W = Read/Write; -*n* = value after reset; -x = value is indeterminate after reset
† If writing to this field, always write the default value for future device compatibility.

Table B–172. Power Management DSP Control/Status Register (PMDCSR) Field Descriptions

Bits	*field*†	*symval*†	Value	Description
31–19	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
18–11	HWPMECTL		0–FFh	Hardware PME control. Allows PME to be generated automatically by hardware on active PWR_WKP if the corresponding bit is set.
			0	Reserved
		REQD0	1h	Requested state = 00
		REQD1	2h	Requested state = 01
		REQD2	3h	Requested state = 10
		REQD3	4h	Requested state = 11
			5h–FFh	Reserved

† For CSL implementation, use the notation PCI_PMDCSR_*field_symval*

Table B–172. Power Management DSP Control/Status Register (PMDCSR) Field Descriptions (Continued)

Bits	*field*†	*symval*†	Value	Description
10	D3WARMONWKP	OF(*value*)		Warm reset from D3. Read-only bit, writes have no effect. Warm resets are only generated from $\overline{\text{PWR\_WKP}}$ if the following conditions are true: • $\overline{\text{PRST}}$ (PCI reset) is deasserted • PCLK is active.
			0	No warm reset is generated on $\overline{\text{PWR\_WKP}}$ asserted (low).
			1	Warm reset is generated on $\overline{\text{PWR\_WKP}}$ asserted if the current state is D3.
9	D2WARMONWKP	OF(*value*)		Warm reset from D2. Read-only bit, writes have no effect. Warm resets are only generated from $\overline{\text{PWR\_WKP}}$ if the following conditions are true: • $\overline{\text{PRST}}$ (PCI reset) is deasserted • PCLK is active.
			0	No warm reset is generated on $\overline{\text{PWR\_WKP}}$ asserted (low).
			1	Warm reset is generated on $\overline{\text{PWR\_WKP}}$ asserted if the current state is D2.
8	PMEEN			PME assertion enable bit. Reads return current value of PMEEN bit in the power management control/status register (PMCSR). Writes of 1 clear both the PMEEN and PMESTAT bits in PMCSR, writes of 0 have no effect.
			0	PMEEN bit in PMCSR is 0; PME assertion is disabled.
		CLR	1	PMEEN bit in PMCSR is 1; PME assertion is enabled.
7	PWRWKP	OF(*value*)		$\overline{\text{PWRWKP}}$ pin value. Read-only bit, writes have no effect.
			0	$\overline{\text{PWR\_WKP}}$ pin is low.
			1	$\overline{\text{PWR\_WKP}}$ pin is high.

† For CSL implementation, use the notation PCI_PMDCSR_*field_symval*

Table B–172. Power Management DSP Control/Status Register (PMDCSR) Field Descriptions (Continued)

Bits	*field*†	*symval*†	Value	Description
6	PMESTAT			PMESTAT sticky bit value. Reads return the current status of the PMESTAT bit in the power management control/status register (PMCSR). If the PMESTAT and PMEEN bits are written with a 1 at the same time, the PMEEN and PMESTAT bits are cleared. Writes of 0 have no effect.
			0	No effect.
		SET	1	Forces the PMESTAT bit in PMCSR to 1.
5	PMEDRVN	OF(*value*)		PME driven high. The DSP has driven the PME pin active high. Read-only bit, writes have no effect.
			0	DSP read of PMDCSR, but bit would be set if the PMEEN and PMESTAT bits are both still high.
			1	PMEEN and PMESTAT bits in the power management control/status register (PMCSR) are high.
4	AUXDETECT	OF(*value*)		$3.3V_{aux}$DET pin value. Read-only bit, writes have no effect.
			0	3.3 V_{aux}DET is low.
			1	3.3 V_{aux}DET is high.
3–2	CURSTATE		0–3h	Current power state. Reflects the current power management state of the device. On changing state, the device must change the CURSTATE bits. The value written here is used for PCI reads of the PWRSTATE bits in the power management control/status register (PMCSR).
		D0	0	Current state = 00
		D1	1h	Current state = 01
		D2	2h	Current state = 10
		D3	3h	Current state = 11
1–0	REQSTATE	OF(*value*)	0–3h	Last requested power state. Last value written by the host to the PCI PWRSTATE bits in the power management control/status register (PMCSR). Cleared to 00b on RESET or PRST. Read-only bit, writes have no effect.

† For CSL implementation, use the notation PCI_PMDCSR_*field_symval*

B.12.3 PCI Interrupt Source Register (PCIIS)

The PCI interrupt source register (PCIIS) shows the status of the interrupt sources. Writing a 1 to the bit(s) clears the condition. Writes of 0 to, and reads from, the bit(s) have no effect. The PCIIS is shown in Figure B–167 and described in Table B–173.

Figure B–167. PCI Interrupt Source Register (PCIIS)

31	16
Reserved†	
R-0	

15	13	12	11	10	9	8
Reserved†		DMAHALTED‡	PRST	Reserved†	EERDY	CFGERR
R-0		R/W-0	R/W-0	R-0	R/W-0	R/W-0

7	6	5	4	3	2	1	0
CFGDONE	MASTEROK	PWRHL	PWRLH	HOSTSW	PCIMASTER	PCITARGET	PWRMGMT
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.
‡ This bit is reserved on C64x DSP.

Table B–173. PCI Interrupt Source Register (PCIIS) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31–13	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
12	DMAHALTED			DMA transfers halted enable bit. (C62x/C67x DSP only)
		CLR	0	Auxiliary DMA transfers are not halted.
			1	Auxiliary DMA transfers have stopped.
11	PRST			PCI reset change state bit.
		NOCHG	0	No change of state on PCI reset.
		CHGSTATE	1	PCI reset changed state.

† For CSL implementation, use the notation PCI_PCIIS_*field_symval*

Table B–173. PCI Interrupt Source Register (PCIIS) Field Descriptions (Continued)

Bit	*field*†	*symval*†	Value	Description
10	Reserved			Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
9	EERDY			EEPROM ready bit.
			0	EEPROM is not ready to accept a new command.
		CLR	1	EEPROM is ready to accept a new command and the data register can be read.
8	CFGERR			Configuration error bit.
			0	No checksum failure during PCI autoinitialization.
		CLR	1	Checksum failed during PCI autoinitialization. Set after an initialization due to PRST asserted and checksum error. Set after WARM if initialization has been done, but had checksum error.
7	CFGDONE			Configuration hold bit.
			0	Configuration of PCI configuration registers is not complete.
		CLR	1	Configuration of PCI configuration registers is complete. Set after an initialization due to PRST asserted. Set after WARM if initialization has been done.
6	MASTEROK			PCI master transaction completes bit.
			0	No PCI master transaction completes interrupt.
		CLR	1	PCI master transaction completes interrupt.
5	PWRHL			High-to-low transition on PWRWKP bit.
			0	No high-to-low transition on PWRWKP.
		CLR	1	High-to-low transition on PWRWKP.
4	PWRLH			Low-to-high transition on PWRWKP bit.
			0	No low-to-high transition on PWRWKP.
		CLR	1	Low-to-high transition on PWRWKP.

† For CSL implementation, use the notation PCI_PCIIS_*field_symval*

Table B–173. PCI Interrupt Source Register (PCIIS) Field Descriptions (Continued)

Bit	*field*†	*symval*†	Value	Description
3	HOSTSW			Host software requested bit.
			0	No host software requested interrupt.
		CLR	1	Host software requested interrupt (this bit must be set after boot from PCI to wake up DSP).
2	PCIMASTER			Master abort received bit.
			0	No master abort received.
		CLR	1	Master abort received.
1	PCITARGET			Target abort received bit.
			0	No target abort received.
		CLR	1	Target abort received.
0	PWRMGMT			Power management state transition bit.
			0	No power management state transition interrupt.
		CLR	1	Power management state transition interrupt (is not set if the DSP clocks are not running).

† For CSL implementation, use the notation PCI_PCIIS_*field_symval*

B.12.4 PCI Interrupt Enable Register (PCIIEN)

The PCI interrupt enable register (PCIIEN) enables the PCI interrupts. For the DSP to monitor the interrupts, the DSP software must also set the appropriate bits in the CPU control status register (CSR) and CPU interrupt enable register (IER).

The only interrupt enabled after device reset ($\overline{\text{RESET}}$) is the HOSTSW interrupt. In this way, the PCI host can wake up the DSP by writing the DSPINT bit in the host-to-DSP control register (HDCR). The PCIIEN is shown in Figure B–168 and described in Table B–174.

Figure B–168. PCI Interrupt Enable Register (PCIIEN)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved†															
R-0															

15	14	13	12	11	10	9	8
Reserved†			Reserved‡	PRST	Reserved‡	EERDY	CFGERR
R-0			R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

7	6	5	4	3	2	1	0
CFGDONE	MASTEROK	PWRHL	PWRLH	HOSTSW	PCIMASTER	PCITARGET	PWRMGMT
R/W-0	R/W-0	R/W-0	R/W-0	R/W-1	R/W-0	R/W-0	R/W-0

Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.
‡ These reserved bits must always be written with 0, writing 1 to these bits result in an undefined operation.

Table B–174. PCI Interrupt Enable Register (PCIIEN) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31–13	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
12	Reserved	–	0	Reserved. The reserved bit location is always read as 0. This reserved bit must always be written with a 0. Writing 1 to this bit results in an undefined operation.

† For CSL implementation, use the notation PCI_PCIIEN_*field_symval*

Table B–174. PCI Interrupt Enable Register (PCIIEN) Field Descriptions (Continued)

Bit	*field*†	*symval*†	Value	Description
11	PRST			$\overline{\text{PRST}}$ transition interrupts enable bit.
		DISABLE	0	$\overline{\text{PRST}}$ transition interrupts are not enabled.
		ENABLE	1	$\overline{\text{PRST}}$ transition interrupts are enabled.
10	Reserved	–	0	Reserved. The reserved bit location is always read as 0. This reserved bit must always be written with a 0. Writing 1 to this bit results in an undefined operation.
9	EERDY			EEPROM ready interrupts enable bit.
		DISABLE	0	EEPROM ready interrupts are not enabled.
		ENABLE	1	EEPROM ready interrupts are enabled.
8	CFGERR			Configuration error interrupts enable bit.
		DISABLE	0	Configuration error interrupts are not enabled.
		ENABLE	1	Configuration error interrupts are enabled.
7	CFGDONE			Configuration complete interrupts enable bit.
		DISABLE	0	Configuration complete interrupts are not enabled.
		ENABLE	1	Configuration complete interrupts are enabled.
6	MASTEROK			PCI master transaction complete interrupts enable bit.
		DISABLE	0	PCI master transaction complete interrupts are not enabled.
		ENABLE	1	PCI master transaction complete interrupts are enabled.
5	PWRHL			High-to-low PWRWKP interrupts enable bit.
		DISABLE	0	High-to-low PWRWKP interrupts are not enabled.
		ENABLE	1	High-to-low PWRWKP interrupts are enabled.
4	PWRLH			Low-to-high PWRKWP interrupts enable bit.
		DISABLE	0	Low-to-high PWRWKP interrupts are not enabled.
		ENABLE	1	Low-to-high PWRKWP interrupts are enabled.

† For CSL implementation, use the notation PCI_PCIIEN_*field_symval*

Table B–174. PCI Interrupt Enable Register (PCIIEN) Field Descriptions (Continued)

Bit	*field*†	*symval*†	Value	Description
3	HOSTSW			Host software requested interrupt enable bit.
		DISABLE	0	Host software requested interrupts are not enabled.
		ENABLE	1	Host software requested interrupt are enabled.
2	PCIMASTER			PCI master abort interrupt enable bit.
		DISABLE	0	PCI master abort interrupt is not enabled.
		ENABLE	1	PCI master abort interrupt is enabled.
1	PCITARGET			PCI target abort interrupt enable bit.
		DISABLE	0	PCI target abort interrupt is not enabled.
		ENABLE	1	PCI target abort interrupt is enabled.
0	PWRMGMT			Power management state transition interrupt enable bit.
		DISABLE	0	Power management state transition interrupt is not enabled.
		ENABLE	1	Power management state transition interrupt is enabled.

† For CSL implementation, use the notation PCI_PCIIEN_*field_symval*

B.12.5 DSP Master Address Register (DSPMA)

The DSP master address register (DSPMA) contains the DSP address location of the destination data for DSP master reads, or the address location of source data for DSP master writes. DSPMA also contains bits to control the address modification. DSPMA is doubleword aligned on the C64x DSP and word aligned on the C6205 DSP. The DSPMA is shown in Figure B–169 and described in Table B–175.

Figure B–169. DSP Master Address Register (DSPMA)

31	2	1	0
ADDRMA		AINC†	—‡
R/W-0		R/W-0	R-0

Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† This bit is valid on C6205 DSP only; on C64x DSP, this bit is reserved and must be written with a 0.
‡ If writing to this field, always write the default value for future device compatibility.

Table B–175. DSP Master Address Register (DSPMA) Field Descriptions

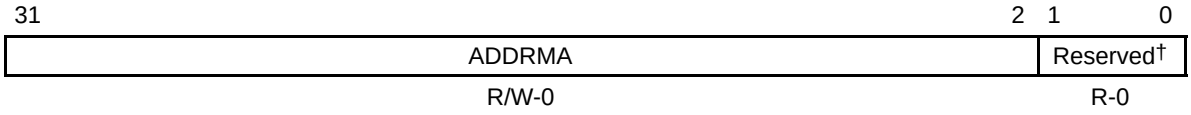
Bit	***field***†	***symval***†	Value	Description
31–2	ADDRMA	OF(*value*)	0–3FFF FFFFh	DSP word address for PCI master transactions.
1	AINC			Autoincrement mode of DSP master address (C6205 DSP only). Autoincrement only affects the lower 24 bits of DSPMA. As a result, autoincrement does not cross 16M-byte boundaries and wraps around if incrementing past the boundary. On the C64x DSP, this bit is reserved and must be written with a 0. The PCI port on the C64x DSP does not support fixed addressing for master PCI transfers. All transfers are issued to linear incrementing addresses in DSP memory.
		DISABLE	0	ADDRMA autoincrement is disabled.
		ENABLE	1	ADDRMA autoincrement is enabled.
0	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.

† For CSL implementation, use the notation PCI_DSPMA_*field_symval*

B.12.6 PCI Master Address Register (PCIMA)

The PCI master address register (PCIMA) contains the PCI word address. For DSP master reads, PCIMA contains the source address;.for DSP master writes, PCIMA contains the destination address. The PCIMA is shown in Figure B–170 and described in Table B–176.

Figure B–170. PCI Master Address Register (PCIMA)



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.

Table B–176. PCI Master Address Register (PCIMA) Field Descriptions

Bit	Field	*symval*†	Value	Description
31–2	ADDRMA	OF(*value*)	0–3FFF FFFFh	PCI word address for PCI master transactions.
1–0	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.

† For CSL implementation, use the notation PCI_PCIMA_ADDRMA_*symval*

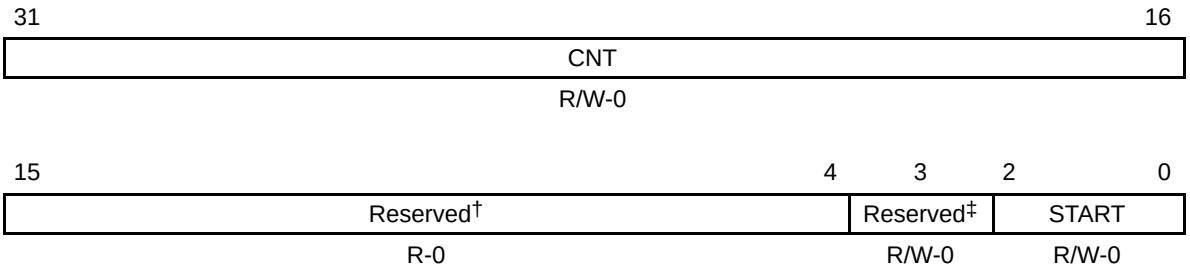
B.12.7 PCI Master Control Register (PCIMC)

The PCI master control register (PCIMC) contains:

- Start bits to initiate the transfer between the DSP and PCI.
- The transfer count, which specifies the number of bytes to transfer (65K bytes maximum).
- Reads indicate transfer status

The PCIMC is shown in Figure B–171 and described in Table B–177.

Figure B–171. PCI Master Control Register (PCIMC)



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.
‡ This reserved bit must always be written with 0, writing 1 to this bit results in an undefined operation.

Table B–177. PCI Master Control Register (PCIMC) Field Descriptions

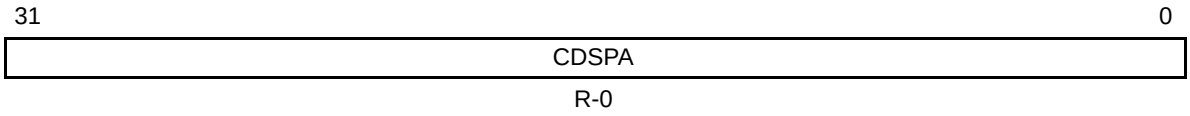
Bit	***field***†	***symval***†	Value	Description
31–16	CNT	OF(*value*)	0–FFFFh	Transfer count specifies the number of bytes to transfer.
15–4	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
3	Reserved	–	0	Reserved. The reserved bit location is always read as 0. This reserved bit must always be written with a 0. Writing 1 to this bit results in an undefined operation.
2–0	START		0–7h	Start the read or write master transaction. The START bits return to 000b when the transaction is complete. The START bits must not be written/changed during an active master transfer. If the PCI bus is reset during a transfer, the transfer stops and the FIFOs are flushed. (A CPU interrupt can be generated on a $\overline{\text{PRST}}$ transition.) The START bits only get set if the CNT bits are not 0000h.
		FLUSH	0	Transaction not started/flush current transaction.
		WRITE	1h	Start a master write transaction.
		READPREF	2h	Start a master read transaction to prefetchable memory.
		READNOPREF	3h	Start a master read transaction to nonprefetchable memory.
			4h	Start a configuration write.
			5h	Start a configuration read.
			6h	Start an I/O write.
			7h	Start an I/O read.

† For CSL implementation, use the notation PCI_PCIMC_*field_symval*

B.12.8 Current DSP Address Register (CDSPA)

The current DSP address register (CDSPA) contains the current DSP address for master transactions. The CDSPA is shown in Figure B–172 and described in Table B–178.

Figure B–172. Current DSP Address (CDSPA)



Legend: R = Read only; -*n* = value after reset

Table B–178. Current DSP Address (CDSPA) Field Descriptions

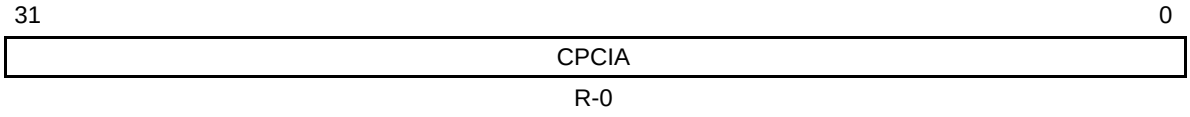
Bit	Field	*symval*†	Value	Description
31–0	CDSPA	OF(*value*)	0–FFFF FFFFh	The current DSP address for master transactions.

† For CSL implementation, use the notation PCI_CDSPA_CDSPA_*symval*

B.12.9 Current PCI Address Register (CPCIA)

The current PCI address register (CPCIA) contains the current PCI address for master transactions. The CPCIA is shown in Figure B–173 and described in Table B–179.

Figure B–173. Current PCI Address Register (CPCIA)



Legend: R = Read only; -*n* = value after reset

Table B–179. Current PCI Address Register (CPCIA) Field Descriptions

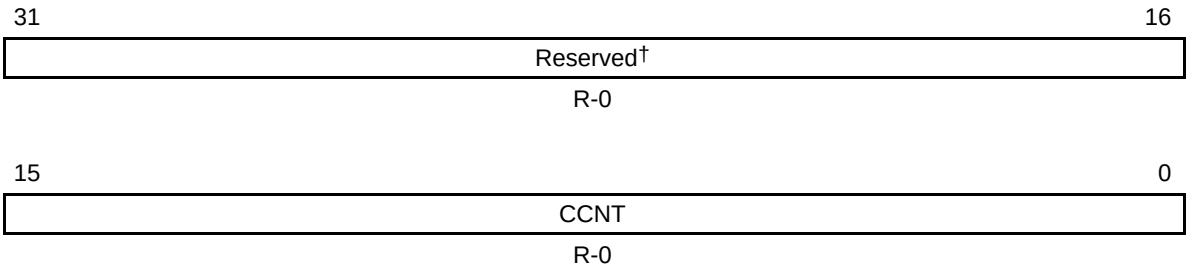
Bit	Field	*symval*†	Value	Description
31–0	CPCIA	OF(*value*)	0–FFFF FFFFh	The current PCI address for master transactions

† For CSL implementation, use the notation PCI_CPCIA_CPCIA_*symval*

B.12.10 Current Byte Count Register (CCNT)

The current byte count register (CCNT) contains the number of bytes left on the current master transaction. The CCNT is shown in Figure B–174 and described in Table B–180.

Figure B–174. Current Byte Count Register (CCNT)



Legend: R = Read only; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.

Table B–180. Current Byte Count Register (CCNT) Field Descriptions

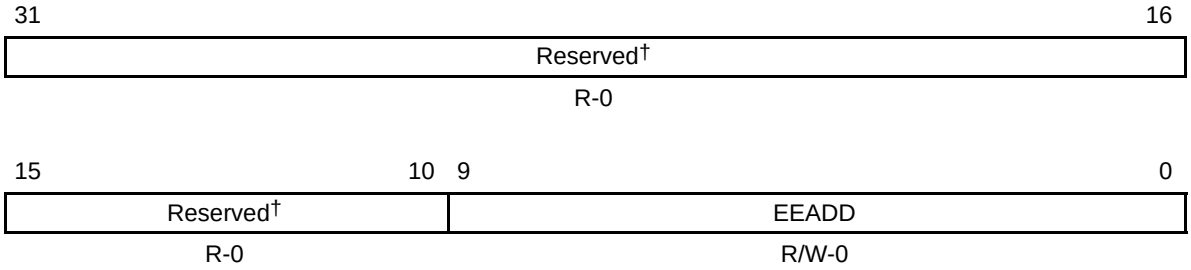
Bit	Field	*symval*†	Value	Description
31–16	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
15–0	CCNT	OF(*value*)	0–FFFFh	The number of bytes left on the master transaction.

† For CSL implementation, use the notation PCI_CCNT_CCNT_*symval*

B.12.11 EEPROM Address Register (EEADD)

The EEPROM address register (EEADD) contains the EEPROM address. The EEADD is shown in Figure B–175 and described in Table B–181.

Figure B–175. EEPROM Address Register (EEADD)



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.

Table B–181. EEPROM Address Register (EEADD) Field Descriptions

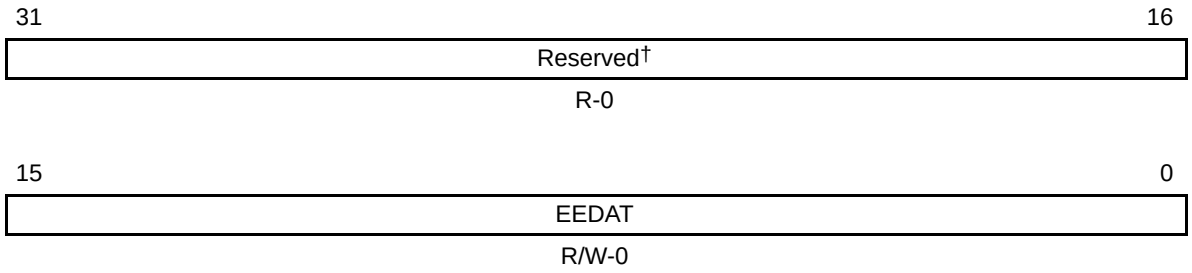
Bit	Field	*symval*†	Value	Description
31–10	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
9–0	EEADD	OF(*value*)	0–3FFh	EEPROM address.

† For CSL implementation, use the notation PCI_EEADD_EEADD_*symval*

B.12.12 EEPROM Data Register (EEDAT)

The EEPROM data register (EEDAT) is used to clock out user data to the EEPROM on writes and store EEPROM data on reads. For EEPROM writes, data written to EEDAT is immediately transferred to an internal register. A DSP read from EEDAT at this point does not return the value of the EEPROM data just written. The write data (stored in the internal register) is shifted out on the pins as soon as the two-bit op code is written to the EECNT bits in the EEPROM control register (EECTL). For EEPROM reads, data is available in EEDAT as soon as the READY bit in EECTL is set to 1. The EEDAT is shown in Figure B–176 and described in Table B–182.

Figure B–176. EEPROM Data Register (EEDAT)



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.

Table B–182. EEPROM Data Register (EEDAT) Field Descriptions

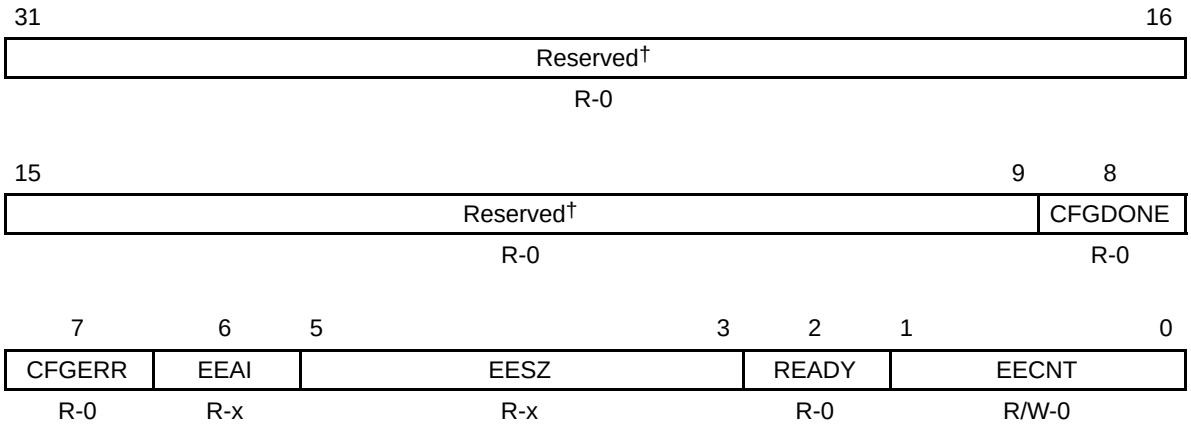
Bit	Field	*symval*†	Value	Description
31–16	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
15–0	EEDAT	OF(*value*)	0–FFFFh	EEPROM data.

† For CSL implementation, use the notation PCI_EEDAT_EEDAT_*symval*

B.12.13 EEPROM Control Register (EECTL)

The EEPROM control register (EECTL) has fields for the two-bit opcode (EECNT) and read-only bits that indicate the size of the EEPROM (EESZ latched from the EESZ[2–0] pins on power-on reset). The READY bit in EECTL indicates when the last operation is complete, and the EEPROM is ready for a new instruction. The READY bit is cleared when a new op code is written to the EECNT bits. An interrupt can also be generated on EEPROM command completion. The EERDY bit in the PCI interrupt source register (PCIIS) and in the PCI interrupt enable register (PCIIEN) control the operation of the interrupt. The EECTL is shown in Figure B–177 and described in Table B–183.

Figure B–177. EEPROM Control Register (EECTL)



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset; -x = value is indeterminate after reset
† If writing to this field, always write the default value for future device compatibility.

Table B–183. EEPROM Control Register (EECTL) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31–9	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
8	CFGDONE	OF(*value*)		Configuration done bit.
			0	Configuration is not done.
			1	Configuration is done.

† For CSL implementation, use the notation PCI_EECTL_*field_symval*

Table B–183. EEPROM Control Register (EECTL) Field Descriptions (Continued)

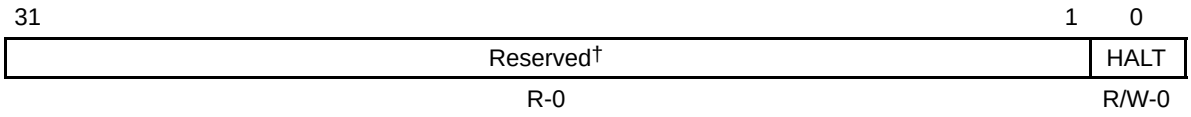
Bit	*field*†	*symval*†	Value	Description
7	CFGERR	OF(*value*)		Checksum failed error bit.
			0	No checksum error.
			1	Checksum error.
6	EEAI	OF(*value*)		EEAI pin state at power-on reset.
			0	PCI uses default values.
			1	Read PCI configuration register values from EEPROM.
5–3	EESZ	OF(*value*)		EESZ pins state at power-on reset.
			0	No EEPROM
			1h	1K bits (C6205 DSP only)
			2h	2K bits (C6205 DSP only)
			3h	4K bits
			4h	16K bits (C6205 DSP only)
			5h–7h	Reserved
2	READY	OF(*value*)		EEPROM is ready for a new command. Cleared on writes to the EECNT bit.
			0	EEPROM is not ready for a new command.
			1	EEPROM is ready for a new command.
1–0	EECNT			EEPROM op code. Writes to this field cause the serial operation to commence.
		EWEN	0	Write enable (address = 11xxxx)
		ERAL	0	Erases all memory locations (address = 10xxxx)
		WRAL	0	Writes all memory locations (address = 01xxxx)
		EWDS	0	Disables programming instructions (address = 00xxxx)
		WRITE	1h	Write memory at address
		READ	2h	Reads data at specified address
		ERASE	3h	Erase memory at address

† For CSL implementation, use the notation PCI_EECTL_*field_symval*

B.12.14 PCI Transfer Halt Register (HALT) (C62x/C67x DSP only)

The PCI transfer halt register (HALT) allows the C62x/C67x DSP to terminate internal transfer requests to the auxiliary DMA channel. The HALT is shown in Figure B–178 and described in Table B–184.

Figure B–178. PCI Transfer Halt Register (HALT)



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.

Table B–184. PCI Transfer Halt Register (HALT) Field Descriptions

Bit	Field	*symval*†	Value	Description
31–1	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
0	HALT			Halt internal transfer requests bit.
			0	No effect.
		SET	1	HALT prevents the PCI port from performing master/slave auxiliary DMA transfer requests.

† For CSL implementation, use the notation PCI_HALT_HALT_*symval*

B.12.15 PCI Transfer Request Control Register (TRCTL) (C64x DSP only)

The PCI transfer request control register (TRCTL) controls how the PCI submits its requests to the EDMA subsystem. The TRCTL is shown in Figure B–179 and described in Table B–185.

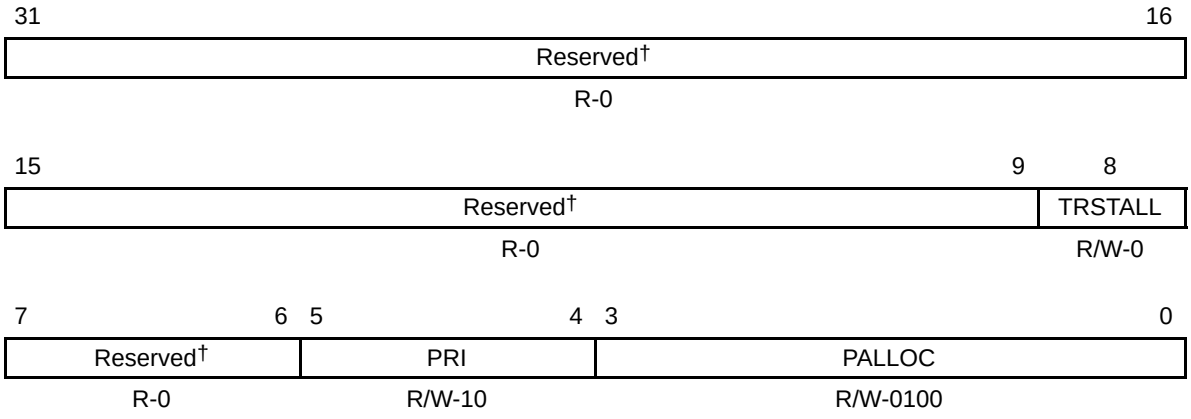
To safely change the PALLOC or PRI bits in TRCTL, the TRSTALL bit needs to be used to ensure a proper transition. The following procedure must be followed to change the PALLOC or PRI bits:

1) Set the TRSTALL bit to 1 to stop the PCI from submitting TR requests on the current PRI level. In the same write, the desired new PALLOC and PRI bits may be specified.
2) Clear all EDMA event enables (EER) corresponding to both old and new PRI levels to stop the EDMA from submitting TR requests on both PRI levels. Do not manually submit additional events via the EDMA.
3) Do not submit new QDMA requests on either old or new PRI level.
4) Stop L2 cache misses on either old or new PRI level. This can be done by forcing program execution or data accesses in internal memory. Another way is to have the CPU executing a tight loop that does not cause additional cache misses.
5) Poll the appropriate PQ bits in the priority queue status register (PQSR) of the EDMA until both queues are empty (see the *Enhanced DMA (EDMA) Controller Reference Guide*, SPRU234).
6) Clear the TRSTALL bit to 0 to allow the PCI to continue normal operation.

Requestors are halted on the old PCI PRI level so that memory ordering can be preserved. In this case, all pending requests corresponding to the old PRI level must be allowed to complete before PCI is released from stall state.

Requestors are halted on the new PRI level to ensure that at no time can the sum of all requestor allocations exceed the queue length. By halting all requestors at a given level, you can be free to modify the queue allocation counters of each requestor.

Figure B–179. PCI Transfer Request Control Register (TRCTL)



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset
† If writing to this field, always write the default value for future device compatibility.

Table B–185. PCI Transfer Request Control Register (TRCTL) Field Descriptions

Bit	***field***†	***symval***†	Value	Description
31–9	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
8	TRSTALL			Forces the PCI to stall all PCI requests to the EDMA. This bit allows the safe changing of the PALLOC and PRI fields.
			0	Allows PCI requests to be submitted to the EDMA.
			1	Halts the creation of new PCI requests to the EDMA.
7–6	Reserved	–	0	Reserved. The reserved bit location always returns the default value. A value written to this field has no effect. If writing to this field, always write the default value for future device compatibility.
5–4	PRI		0–3h	Controls the priority queue level that PCI requests are submitted to.
			0	Urgent priority
			1h	High priority
			2h	Medium priority
			3h	Low priority
3–0	PALLOC		0–Fh	Controls the total number of outstanding requests that can be submitted by the PCI to the EDMA. Valid values of PALLOC are 1 to 15, all other values are reserved. PCI may have the programmed number of outstanding requests.

† For CSL implementation, use the notation PCI_TRCTL_*field_symval*

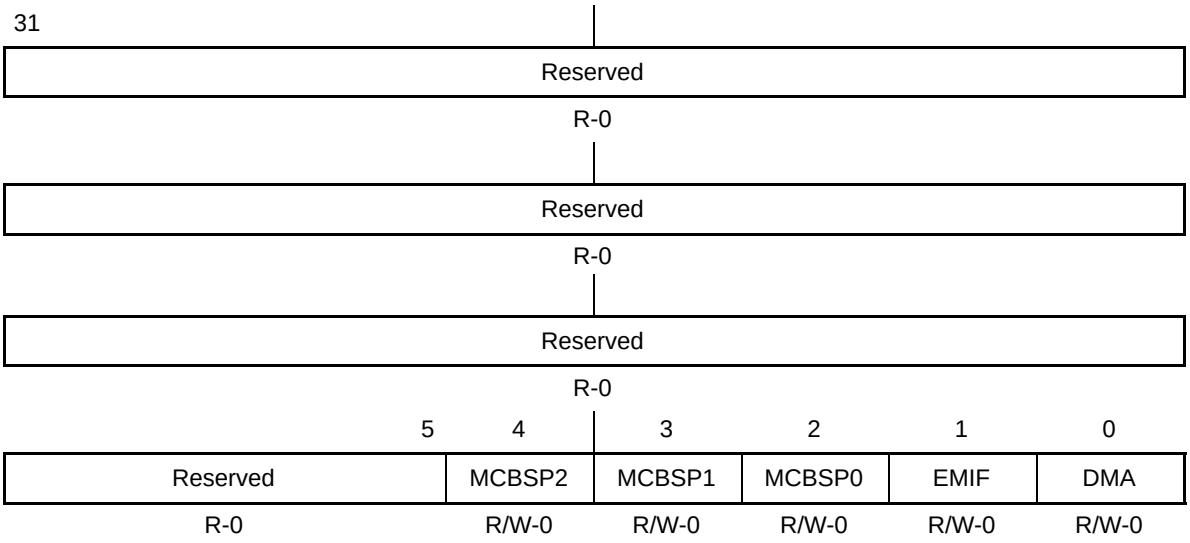
B.13 Power-Down Logic Register

Table B–186. Power-Down Logic Register

Acronym	Register Name	Section
PDCTL	Power down control register	B.13.1

B.13.1 Power-Down Control Register

Figure B–180. Power-Down Control Register (PDCTL)



Legend: R/W-x = Read/Write-Reset value

Table B–187. Power-Down Control Register (PDCTL) Field Values (PWR_PDCTL_field_symval)

Bit	*field*	*symval*	Value	Description
31–5	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
4	MCBSP2			Enable/disable internal McBSP2 clock
		CLKON	0	Internal McBSP2 clock enabled.
		CLKOFF	1	Internal McBSP2 clock disabled, McBSP2 is not functional
3	MCBSP1			Enable/disable internal McBSP1 clock
		CLKON	0	Internal McBSP1 clock enabled.
		CLKOFF	1	Internal McBSP1 clock disabled, McBSP1 is not functional.

Table B–187. Power-Down Control Register (PDCTL) Field Values (PWR_PDCTL_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
2	MCBSP0			Enable/disable internal McBSP0 clock
		CLKON	0	Internal McBSP0 clock enabled.
		CLKOFF	1	Internal McBSP0 clock disabled, McBSP1 is not functional.
1	EMIF			Enable/disable internal EMIF clock
		CLKON	0	Internal EMIF clock enabled.
		CLKOFF	1	Internal EMIF clock disabled. EMIF is not functional.
0	DMA			Enable/disable internal DMA clock
		CLKON	0	Internal DMA clock enabled.
		CLKOFF	1	Internal DMA clock disabled. DMA is not functional

B.14 Phase-Locked Loop (PLL) Registers

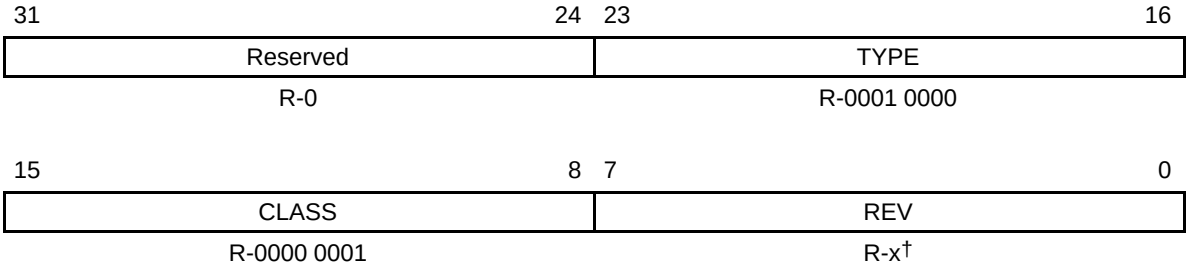
Table B–188. PLL Controller Registers

Acronym	Register Name	Section
PLLPID	PLL controller peripheral identification register	B.14.1
PLLCSR	PLL control/status register	B.14.2
PLLM	PLL multiplier control register	B.14.3
PLLDIV0–3	PLL controller divider registers	B.14.4
OSCDIV1	Oscillator divider 1 register	B.14.5

B.14.1 PLL Controller Peripheral Identification Register (PLLPID)

The PLL controller peripheral identification register (PLLPID) contains identification code for the PLL controller. PLLPID is shown in Figure B–93 and described in Table B–99.

Figure B–181. PLL Controller Peripheral Identification Register (PLLPID)



Legend: R = Read only; -x = value after reset
† See the device-specific datasheet for the default value of this field.

Table B–189. PLL Controller Peripheral Identification Register (PLLPID) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31–24	Reserved	–	0	These Reserved bit locations are always read as zeros. A value written to this field has no effect.
23–16	TYPE	OF(*value*)		Identifies type of peripheral.
			10h	PLL controller

† For CSL implementation, use the notation PLL_PID_*field_symval*

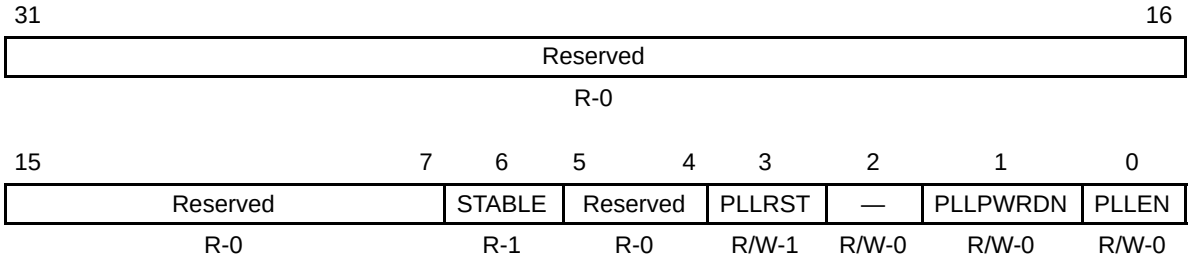
Bit	*field*†	*symval*†	Value	Description
15–8	CLASS	OF(*value*)		Identifies class of peripheral.
			1	Serial port
7–0	REV	OF(*value*)		Identifies revision of peripheral.
			x	See the device-specific datasheet for the value.

† For CSL implementation, use the notation PLL_PID_*field_symval*

B.14.2 PLL Control/Status Register (PLLCSR)

The PLL control/status register (PLLCSR) is shown in Figure B–182 and described in Table B–190.

Figure B–182. PLL Control/Status Register (PLLCSR)



Legend: R = Read only; R/W = Read/write; -*n* = value after reset

Table B–190. PLL Control/Status Register (PLLCSR) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31–7	Reserved	–	0	Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
6	STABLE	OF(*value*)		Oscillator input stable bit indicates if the OSCIN/CLKIN input has stabilized. The STABLE bit is set to 1 after the reset controller counts 4096 input clock cycles after the RESET signal is asserted high.
			0	OSCIN/CLKIN input is not yet stable. Oscillator counter is not finished counting.
			1	OSCIN/CLKIN input is stable.
5–4	Reserved	–	0	Reserved. The reserved bit location is always read as zero. Always write a 0 to this location.

† For CSL implementation, use the notation PLL_PLLCSR_*field_symval*

Table B–190. PLL Control/Status Register (PLLCSR) Field Descriptions (Continued)

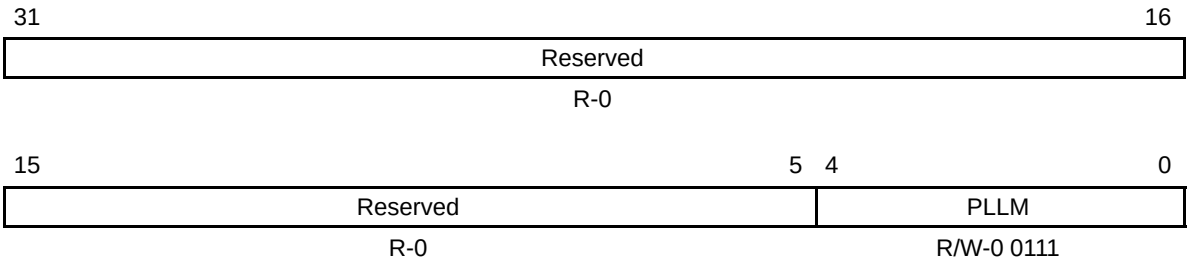
Bit	*field*†	*symval*†	Value	Description
3	PLLRST			PLL reset bit.
		0	0	PLL reset is released.
		1	1	PLL reset is asserted.
2	Reserved	–	0	Reserved. The reserved bit location is always read as zero. Always write a 0 to this location.
1	PLLPWRDN			PLL power-down mode select bit.
		NO	0	PLL is operational.
		YES	1	PLL is placed in power-down state.
0	PLLEN			PLL enable bit.
		BYPASS	0	Bypass mode. Divider D0 and PLL are bypassed. SYSCLK1/SYSCLK2/SYSCLK3 are divided down directly from input reference clock.
		ENABLE	1	PLL mode. PLL output path is enabled. Divider D0 and PLL are not bypassed. SYSCLK1/SYSCLK2/SYSCLK3 are divided down from PLL output.

† For CSL implementation, use the notation PLL_PLLCSR_*field_symval*

B.14.3 PLL Multiplier Control Register (PLLM)

The PLL multiplier control register (PLLM) is shown in Figure B–183 and described in Table B–191. The PLLM defines the input reference clock frequency multiplier in conjunction with the PLL divider ratio bits (RATIO) in the PLL controller divider 0 register (PLLDIV0).

Figure B–183. PLL Multiplier Control Register (PLLM)



Legend: R = Read only; R/W = Read/write; -*n* = value after reset

Table B–191. PLL Multiplier Control Register (PLLM) Field Descriptions

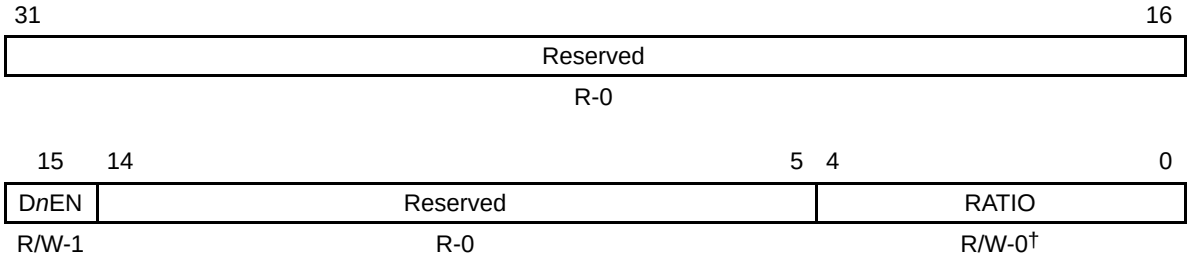
Bit	Field	*symval*†	Value	Description
31–5	Reserved	–	0	Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
4–0	PLLM	OF(*value*)	0–1Fh	PLL multiplier bits. Defines the frequency multiplier of the input reference clock in conjunction with the PLL divider ratio bits (RATIO) in PLLDIV0. See the device-specific datasheet for the PLL multiplier rates supported on your device.

† For CSL implementation, use the notation PLL_PLLM_PLLM_*symval*

B.14.4 PLL Controller Divider Registers (PLLDIV0–3)

The PLL controller divider register (PLLDIV) is shown in Figure B–184 and described in Table B–192.

Figure B–184. PLL Controller Divider Register (PLLDIV)



Legend: R = Read only; R/W = Read/write; -*n* = value after reset
† For PLLDIV0 and PLLDIV1; for PLLDIV2 and PLLDIV3, reset value is 0 0001.

Table B–192. PLL Controller Divider Register (PLLDIV) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31–16	Reserved	–	0	Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
15	D*n*EN			Divider D*n* enable bit.
		DISABLE	0	Divider *n* is disabled. No clock output.
		ENABLE	1	Divider *n* is enabled.
14–5	Reserved	–	0	Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.

† For CSL implementation, use the notation PLL_PLLDIV*n_field_symval*

Table B–192. PLL Controller Divider Register (PLLDIV) Field Descriptions (Continued)

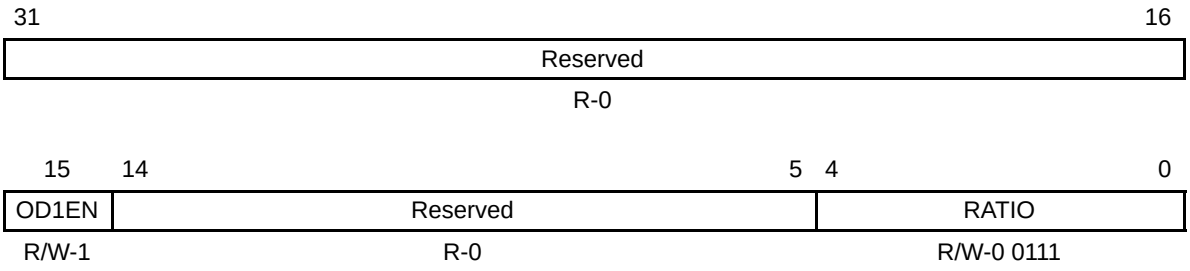
Bit	*field*†	*symval*†	Value	Description
4–0	RATIO	OF(*value*)		PLL divider ratio bits. For PLLDIV0, defines the input reference clock frequency multiplier in conjunction with the PLL multiplier bits (PLLM) in PLLM. For PLLDIV1–3, defines the PLL output clock frequency divider ratio.
			0	÷1. Divide frequency by 1.
			1h	÷2. Divide frequency by 2.
			2h–1Fh	÷3 to ÷32. Divide frequency by 3 to divide frequency by 32.

† For CSL implementation, use the notation PLL_PLLDIV*n_field_symval*

B.14.5 Oscillator Divider 1 Register (OSCDIV1)

The oscillator divider 1 register (OSCDIV1) is shown in Figure B–185 and described in Table B–193.

Figure B–185. Oscillator Divider 1 Register (OSCDIV1)



Legend: R = Read only; R/W = Read/write; -*n* = value after reset

Table B–193. Oscillator Divider 1 Register (OSCDIV1) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31–16	Reserved	–	0	Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
15	OD1EN			Oscillator divider enable bit.
		DISABLE	0	Oscillator divider is disabled. No clock output.
		ENABLE	1	Oscillator divider is enabled.

† For CSL implementation, use the notation PLL_OSCDIV1_*field_symval*

Table B–193. Oscillator Divider 1 Register (OSCDIV1) Field Descriptions (Continued)

Bit	*field*†	*symval*†	Value	Description
14–5	Reserved	–	0	Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
4–0	RATIO	OF(*value*)		Oscillator divider ratio bits. Defines the input reference clock frequency divider ratio for output clock CLKOUT3.
			0	÷1. Divide input reference clock frequency by 1.
			1h	÷2. Divide input reference clock frequency by 2.
			2h–1Fh	÷3 to ÷32. Divide input reference clock frequency by 3 to divide input reference clock frequency by 32.

† For CSL implementation, use the notation PLL_OSCDIV1_*field_symval*

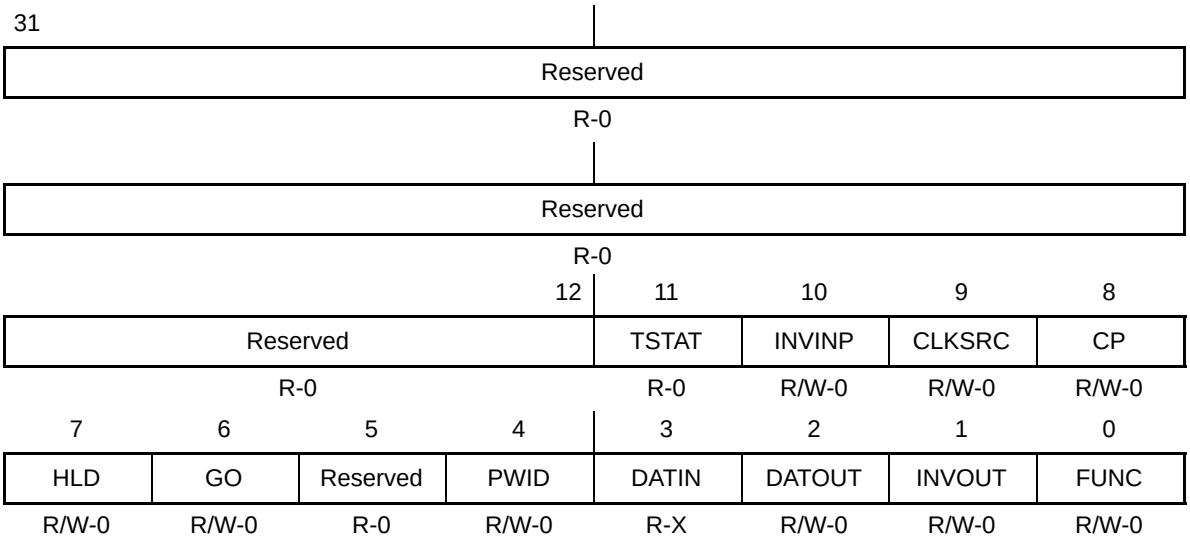
B.15 Timer Registers

Table B–194. Timer Registers

Acronym	Register Name	Section
CTL	Timer control register	B.15.1
PRD	Timer period register	B.15.2
CNT	Timer count register	B.15.3

B.15.1 Timer Control Register (CTL)

Figure B–186. Timer Control Register (CTL)



Legend: R/W-x = Read/Write-Reset value

Table B–195. Timer Control Register (CTL) Field Values (TIMER_CTL_field_symval)

Bit	*field*	*symval*	Value	Description
31–12	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
11	TSTAT			Timer status. Value of timer output.
		0	0	
		1	1	
10	INVINP			TINP inverter control. Only affects operation if CLKSRC = 0.

Table B–195. Timer Control Register (CTL) Field Values (TIMER_CTL_field_symval) (Continued)

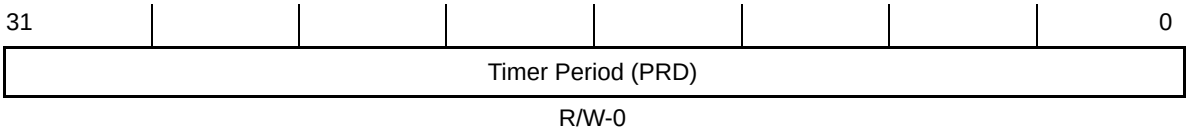
Bit	*field*	*symval*	Value	Description
		NO	0	Uninverted TINP drives timer.
		YES	1	Inverted TINP drives timer.
9	CLKSRC			Timer input clock source
		EXTERNAL	0	External clock source drives the TINP pin.
		CPUOVR4	1	Internal clock source. For C62x/C67x: CPU clock/4
		CPUOVR8	1	Internal clock source. For C64x: CPU clock/8
8	CP			Clock/pulse mode
		PULSE	0	Pulse mode. TSTAT is active one CPU clock after the timer reaches the timer period. PWID determines when it goes inactive.
		CLOCK	1	Clock mode. TSTAT has a 50% duty cycle with each high and low period one countdown period wide.
7	HLD			Hold. Counter may be read or written regardless of $\overline{\text{HLD}}$ value.
		YES	0	Counter is disabled and held in the current state.
		NO	1	Counter is allowed to count.
6	GO			GO bit. Resets and starts the timer counter.
		NO	0	No effect on the timers.
		YES	1	If HLD = 1, the counter register is zeroed and begins counting on the next clock.
5	Reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
4	PWID			Pulse width. Only used in pulse mode (CP = 0).
		ONE	0	TSTAT goes inactive one timer input clock cycle after the timer counter value equals the timer period value.
		TWO	1	TSTAT goes inactive two timer input clock cycles after the timer counter value equals the timer period value.
3	DATIN			Data in. Value on TINP pin.
2	DATOUT			Data output.
		0	0	DATOUT is driven on TOUT.

Table B–195. Timer Control Register (CTL) Field Values (TIMER_CTL_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
		1	1	TSTAT is driven on TOUT after inversion by INVOUT.
1	INVOUT			TOUT inverter control (used only if FUNC = 1).
		NO	0	Uninverted TSTAT drives TOUT.
		YES	1	Inverted TSTAT drives TOUT.
0	FUNC			Function of TOUT pin.
		GPIO	0	TOUT is a general-purpose output pin.
		TOUT	1	TOUT is a timer output pin.

B.15.2 Timer Period Register (PRD)

Figure B–187. Timer Period Register (PRD)



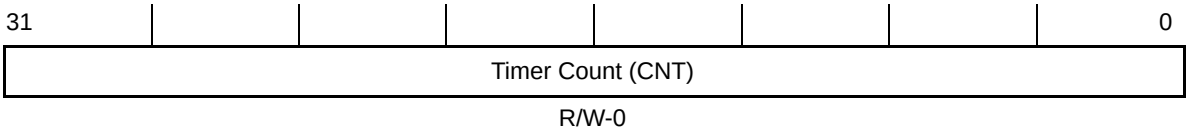
Legend: R/W-x = Read/Write-Reset value

Table B–196. Timer Period Register (PRD) Field Values (TIMER_PRD_field_symval)

Bit	*field*	*symval*	Value	Description
31–0	PRD	OF(*value*)	0–FFFFFFFFh	This 32-bit value is used to reload the timer count register (CNT).

B.15.3 Timer Count Register (CNT)

Figure B–188. Timer Count Register (CNT)



Legend: R/W-x = Read/Write-Reset value

Table B–197. Timer Count Register (CNT) Field Values (TIMER_CNT_field_symval)

Bit	*field*	*symval*	Value	Description
31–0	CNT	OF(*value*)	0–FFFFFFFFh	

B.16 VCP Registers

The VCP contains several memory-mapped registers accessible via CPU load and store instructions, the QDMA, and the EDMA. A peripheral-bus access is faster than an EDMA-bus access for isolated accesses (typically when accessing control registers). EDMA-bus accesses are intended to be used for EDMA transfers and are meant to provide maximum throughput to/from the VCP.

The memory map is listed in Table B–198. The branch metric and decision memories contents are not accessible and the memories can be regarded as FIFOs by the DSP, meaning you do not have to perform any indexing on the addresses.

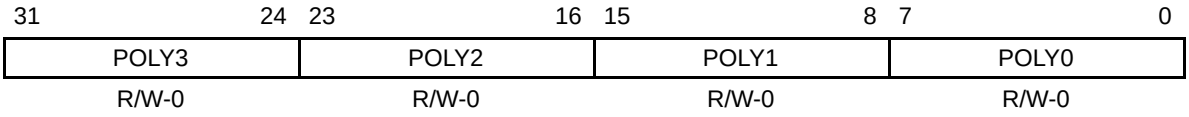
Table B–198. EDMA Bus Accesses Memory Map

Start Address (hex)				
EDMA bus	**Peripheral Bus**	**Acronym**	**Register Name**	**Section**
5000 0000	01B8 0000	VCPIC0	VCP Input Configuration Register 0	B.16.1
5000 0004	01B8 0004	VCPIC1	VCP Input Configuration Register 1	B.16.2
5000 0008	01B8 0008	VCPIC2	VCP Input Configuration Register 2	B.16.3
5000 000C	01B8 000C	VCPIC3	VCP Input Configuration Register 3	B.16.4
5000 0010	01B8 0010	VCPIC4	VCP Input Configuration Register 4	B.16.5
5000 0014	01B8 0014	VCPIC5	VCP Input Configuration Register 5	B.16.6
5000 0048	01B8 0048	VCPOUT0	VCP Output Register 0	B.16.7
5000 004C	01B8 004C	VCPOUT1	VCP Output Register 1	B.16.8
5000 0080	–	VCPWBM	VCP Branch Metrics Write Register	–
5000 0088	–	VCPRDECS	VCP Decisions Read Register	–
–	01B8 0018	VCPEXE	VCP Execution Register	B.16.9
–	01B8 0020	VCPEND	VCP Endian Mode Register	B.16.10
–	01B8 0040	VCPSTAT0	VCP Status Register 0	B.16.11
–	01B8 0044	VCPSTAT1	VCP Status Register 1	B.16.12
–	01B8 0050	VCPERR	VCP Error Register	B.16.13

B.16.1 VCP Input Configuration Register 0 (VCPIC0)

The VCP input configuration register 0 (VCPIC0) is shown in Figure B–189 and described in Table B–199.

Figure B–189. VCP Input Configuration Register 0 (VCPIC0)



Legend: R/W = Read/write; -*n* = value after reset

Table B–199. VCP Input Configuration Register 0 (VCPIC0) Bit Field Description

Bit	***field***†	***symval***†	Value	Description‡
31–24	POLY3	OF(*value*)	0–FFh	Polynomial generator G_3 .
23–16	POLY2	OF(*value*)	0–FFh	Polynomial generator G_2 .
15–8	POLY1	OF(*value*)	0–FFh	Polynomial generator G_1 .
7–0	POLY0	OF(*value*)	0–FFh	Polynomial generator G_0 .

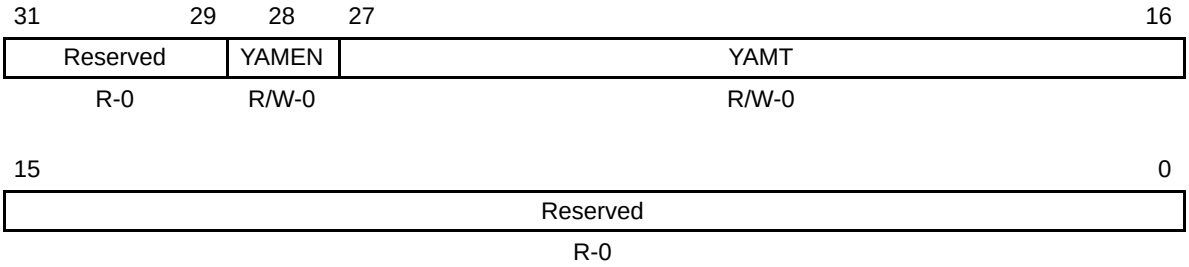
† For CSL implementation, use the notation VCP_IC0_POLY*n*_*symval*

‡ The polynomial generators are 9-bit values defined as $G(z) = b_8z^{-8} + b_7z^{-7} + b_6z^{-6} + b_5z^{-5} + b_4z^{-4} + b_3z^{-3} + b_2z^{-2} + b_1z^{-1} + b_0$, but only 8 bits are passed in the POLY*n* bitfields so that b_1 is the most significant bit and b_8 the least significant bit (b_0 is not passed but set to 1 by the internal VCP hardware).

B.16.2 VCP Input Configuration Register 1 (VCPIC1)

The VCP input configuration register 1 (VCPIC1) is shown in Figure B–190 and described in Table B–200.

Figure B–190. VCP Input Configuration Register 1 (VCPIC1)



Legend: R/W = Read/write; -*n* = value after reset

Table B–200. VCP Input Configuration Register 1 (VCPIC1) Bit Field Description

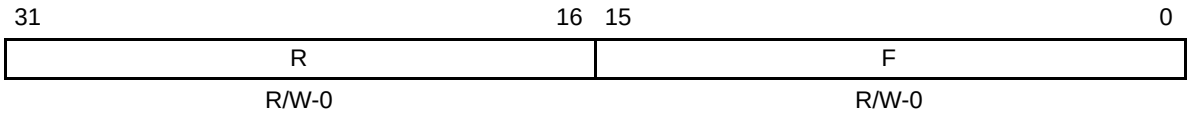
Bit	*field*†	*symval*†	Value	Description
31–29	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
28	YAMEN			Yamamoto algorithm enable bit.
		DISABLE	0	Yamamoto algorithm is disabled.
		ENABLE	1	Yamamoto algorithm is enabled.
27–16	YAMT	OF(*value*)	0–FFFh	Yamamoto threshold value bits.
15–0	Reserved	–	0	Reserved. These Reserved bit locations must be 0. A value written to this field has no effect.

† For CSL implementation, use the notation VCP_IC1_*field_symval*

B.16.3 VCP Input Configuration Register 2 (VCPIC2)

The VCP input configuration register 2 (VCPIC2) is shown in Figure B–191 and described in Table B–201.

Figure B–191. VCP Input Configuration Register 2 (VCPIC2)



Legend: R/W = Read/write; -*n* = value after reset

Table B–201. VCP Input Configuration Register 2 (VCPIC2) Bit Field Description

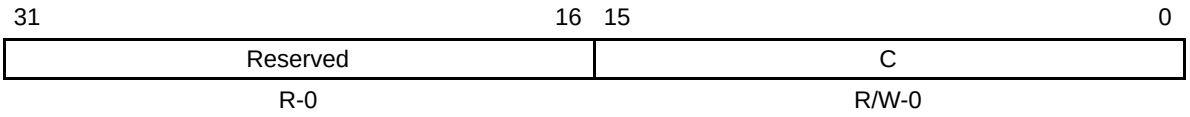
Bit	*field*†	*symval*†	Value	Description
31–16	R	OF(*value*)	0–FFFFh	Reliability length bits.
15–0	F	OF(*value*)	0–FFFFh	Frame length bits.

† For CSL implementation, use the notation VCP_IC2_*field_symval*

B.16.4 VCP Input Configuration Register 3 (VCPIC3)

The VCP input configuration register 3 (VCPIC3) is shown in Figure B–192 and described in Table B–202.

Figure B–192. VCP Input Configuration Register 3 (VCPIC3)



Legend: R/W = Read/write; -*n* = value after reset

Table B–202. VCP Input Configuration Register 3 (VCPIC3) Bit Field Description

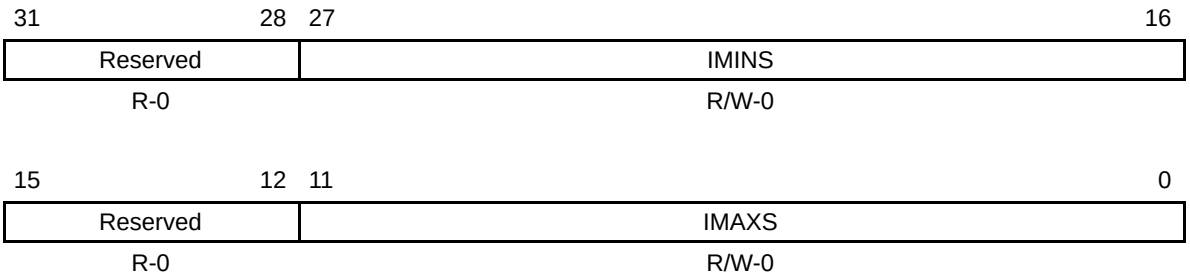
Bit	Field	*symval*†	Value	Description
31–16	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
15–0	C	OF(*value*)	0–FFFFh	Convergence distance bits.

† For CSL implementation, use the notation VCP_IC3_C_*symval*

B.16.5 VCP Input Configuration Register 4 (VCPIC4)

The VCP input configuration register 4 (VCPIC4) is shown in Figure B–193 and described in Table B–203.

Figure B–193. VCP Input Configuration Register 4 (VCPIC4)



Legend: R/W = Read/write; -*n* = value after reset

Table B–203. VCP Input Configuration Register 4 (VCPIC4) Bit Field Description

Bit	*field*†	*symval*†	Value	Description
31–28	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
27–16	IMINS	OF(*value*)	0–FFFh	Minimum initial state metric value bits.
15–12	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
11–0	IMAXS	OF(*value*)	0–FFFh	Maximum initial state metric value bits.

† For CSL implementation, use the notation VCP_IC4_*field_symval*

B.16.6 VCP Input Configuration Register 5 (VCPIC5)

The VCP input configuration register 5 (VCPIC5) is shown in Figure B–194 and described in Table B–204.

Figure B–194. VCP Input Configuration Register 5 (VCPIC5)

31	30	29–26	25–24	23–20	19–16
SDHD	OUTF	Reserved	TB	SYMR	SYMX
R/W-0	R/W-0	R-0	R/W-0	R/W-0	R/W-0

15–8	7–0
Reserved	IMAXI
R-0	R/W-0

Legend: R/W = Read/write; -*n* = value after reset

Table B–204. VCP Input Configuration Register 5 (VCPIC5) Bit Field Description

Bit	*field*†	*symval*†	Value	Description
31	SDHD			Output decision type select bit.
		HARD	0	Hard decisions.
		SOFT	1	Soft decisions.
30	OUTF			Output parameters read flag bit.

† For CSL implementation, use the notation VCP_IC5_*field_symval*

Table B–204. VCP Input Configuration Register 5 (VCPIC5) Bit Field Description (Continued)

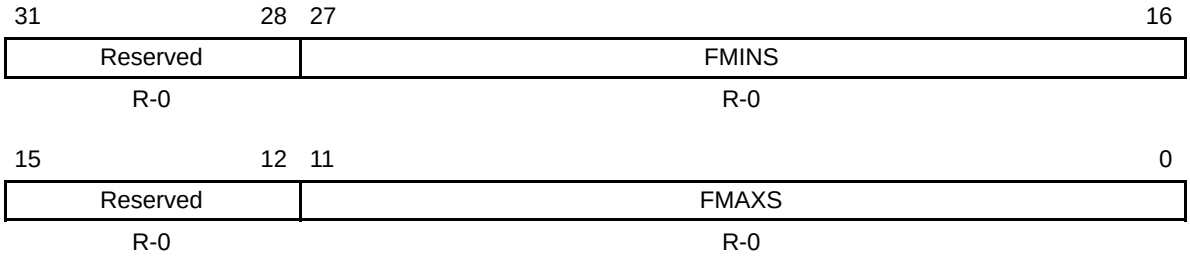
Bit	*field*†	*symval*†	Value	Description
		NO	0	VCPREVT is not generated by VCP for output parameters read.
		YES	1	VCPREVT generated by VCP for output parameters read.
29–26	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
25–24	TB			Traceback mode select bits.
		NO	0	Not allowed.
		TAIL	1h	Tailed.
		CONV	2h	Convergent.
		MIX	3h	Mixed.
23–20	SYMR	OF(*value*)	0–Fh	Determines decision buffer length in output FIFO. When programming register values for the SYMR bits, always subtract 1 from the value calculated. Valid values for the SYMR bits are from 0 to Fh.
19–16	SYMX	OF(*value*)	0–Fh	Determines branch metrics buffer length in input FIFO. When programming register values for the SYMX bits, always subtract 1 from the value calculated. Valid values for the SYMX bits are from 0 to Fh.
15–8	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
7–0	IMAXI	OF(*value*)	0–FFh	Maximum initial state metric value bits. IMAXI bits determine which state should be initialized with the maximum state metrics value (IMAXS) bits in VCPIC4; all the other states will be initialized with the value in the IMINS bits.

† For CSL implementation, use the notation VCP_IC5_*field_symval*

B.16.7 VCP Output Register 0 (VCPOUT0)

The VCP output register 0 (VCPOUT0) is shown in Figure B–195 and described in Table B–205.

Figure B–195. VCP Output Register 0 (VCPOUT0)



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset

Table B–205. VCP Output Register 0 (VCPOUT0) Bit Field Description

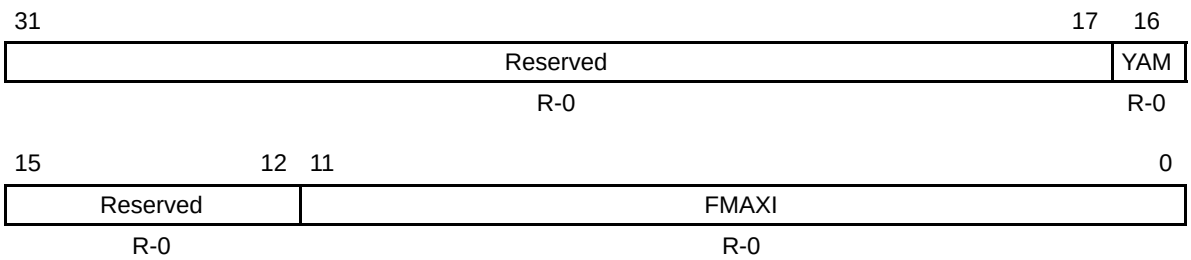
Bit	***field***†	***symval***†	Value	Description
31–28	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
27–16	FMINS	OF(*value*)	0–FFFh	Final minimum state metric value bits.
15–12	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
11–0	FMAXS	OF(*value*)	0–FFFh	Final maximum state metric value bits.

† For CSL implementation, use the notation VCP_OUT0_*field_symval*

B.16.8 VCP Output Register 1 (VCPOUT1)

The VCP output register 1 (VCPOUT1) is shown in Figure B–196 and described in Table B–206.

Figure B–196. VCP Output Register 1 (VCPOUT1)



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset

Table B–206. VCP Output Register 1 (VCPOUT1) Bit Field Description

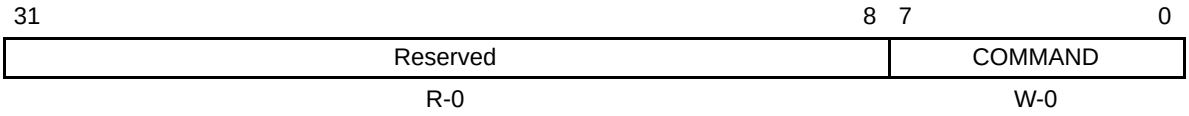
Bit	*field*†	*symval*†	Value	Description
31–17	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
16	YAM			Yamamoto bit result.
		NO	0	
		YES	1	
15–12	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
11–0	FMAXI	OF(*value*)	0–FFFh	State index for the state with the final maximum state metric.

† For CSL implementation, use the notation VCP_OUT1_*field_symval*

B.16.9 VCP Execution Register (VCPEXE)

The VCP execution register (VCPEXE) is shown in Figure B–197 and described in Table B–207.

Figure B–197. VCP Execution Register (VCPEXE)



Legend: R/W = Read/write; W = Write only; -*n* = value after reset

Table B–207. VCP Execution Register (VCPEXE) Bit Field Description

Bit	Field	*symval*†	Value	Description
31–8	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
7–0	COMMAND			VCP command select bits.
			0	Reserved.
		START	1h	Start.
		PAUSE	2h	Pause.

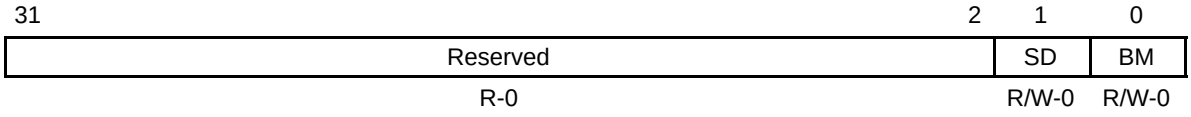
Bit	Field	*symval*†	Value	Description
			3h	Reserved
		UNPAUSE	4h	Unpause.
		STOP	5h	Stop
			6h–FFh	Reserved.

† For CSL implementation, use the notation VCP_EXE_COMMAND_*symval*

B.16.10 VCP Endian Mode Register (VCPEND)

The VCP endian mode register (VCPEND) is shown in Figure B–198 and described in Table B–208. VCPEND has an effect only in big-endian mode.

Figure B–198. VCP Endian Mode Register (VCPEND)



Legend: R/W = Read/write; -*n* = value after reset

Table B–208. VCP Endian Mode Register (VCPEND) Bit Field Description

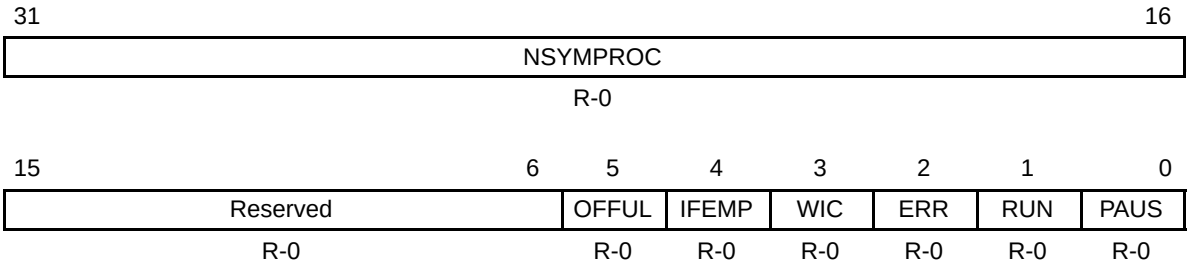
Bit	*field*†	*symval*†	Value	Description
31–2	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
1	SD			Soft-decisions memory format select bit.
		32BIT	0	32-bit-word packed.
		NATIVE	1	Native format (16 bits).
0	BM			Branch metrics memory format select bit.
		32BIT	0	32-bit-word packed.
		NATIVE	1	Native format (7 bits).

† For CSL implementation, use the notation VCP_END_*field_symval*

B.16.11 VCP Status Register 0 (VCPSTAT0)

The VCP status register 0 (VCPSTAT0) is shown in Figure B–199 and described in Table B–209.

Figure B–199. VCP Status Register 0 (VCPSTAT0)



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset

Table B–209. VCP Status Register 0 (VCPSTAT0) Bit Field Description

Bit	***field***†	***symval***†	Value	Description
31–16	NSYMPROC	OF(*value*)	0–FFFFh	Number of symbols processed bits. The NSYMPROC bits indicate how many symbols have been processed in the state metric unit.
15–6	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
5	OFFUL			Output FIFO buffer full status bit.
		NO	0	Output FIFO buffer is not full.
		YES	1	Output FIFO buffer is full.
4	IFEMP			Input FIFO buffer empty status bit.
		NO	0	Input FIFO buffer is not empty.
		YES	1	Input FIFO buffer is empty.
3	WIC			Waiting for input configuration bit. The WIC bit indicates that the VCP is waiting for new input control parameters to be written. This bit is always set after decoding of a user channel.
		NO	0	Not waiting for input configuration words.
		YES	1	Waiting for input configuration words.

† For CSL implementation, use the notation VCP_STAT0_*field_symval*

Table B–209. VCP Status Register 0 (VCPSTAT0) Bit Field Description (Continued)

Bit	*field*†	*symval*†	Value	Description
2	ERR			VCP error status bit. The ERR bit is cleared as soon as the DSP reads the VCP error register (VCPERR).
		NO	0	No error.
		YES	1	VCP paused due to error.
1	RUN			VCP running status bit.
		NO	0	VCP is not running.
		YES	1	VCP is running.
0	PAUS			VCP pause status bit.
		NO	0	VCP is not paused. The UNPAUSE command is acknowledged by clearing the PAUS bit.
		YES	1	VCP is paused. The PAUSE command is acknowledged by setting the PAUS bit. The PAUS bit can also be set, if the input FIFO buffer is becoming empty or if the output FIFO buffer is full.

† For CSL implementation, use the notation VCP_STAT0_*field_symval*

B.16.12 VCP Status Register 1 (VCPSTAT1)

The VCP status register 1 (VCPSTAT1) is shown in Figure B–200 and described in Table B–210.

Figure B–200. VCP Status Register 1 (VCPSTAT1)

31 16	15 0
NSYMOF	NSYMIF
R-0	R-0

Legend: R = Read only; -*n* = value after reset

Table B–210. VCP Status Register 1 (VCPSTAT1) Bit Field Description

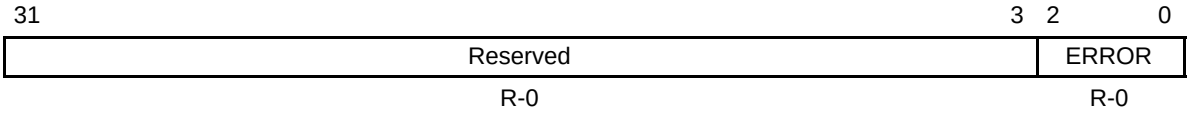
Bit	*field*†	*symval*†	Value	Description
31–16	NSYMOF	OF(*value*)	0–FFFFh	Number of symbols in the output FIFO buffer.
15–0	NSYMIF	OF(*value*)	0–FFFFh	Number of symbols in the input FIFO buffer.

† For CSL implementation, use the notation VCP_STAT1_*field_symval*

B.16.13 VCP Error Register (VCPERR)

The VCP error register (VCPERR) is shown in Figure B–201 and described in Table B–211.

Figure B–201. VCP Error Register (VCPERR)



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset

Table B–211. VCP Error Register (VCPERR) Bit Field Description

Bit	Field	*symval*†	Value	Description
31–3	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
2–0	ERROR			VCP error indicator bits.
		NO	0	No error is detected.
		TBNA	1h	Traceback mode is not allowed.
		FTL	2h	F too large for tailed traceback mode.
		FCTL	3h	*R* + *C* too large for mixed or convergent traceback modes.
			4h–7h	Reserved

† For CSL implementation, use the notation VCP_ERR_ERROR_*symval*

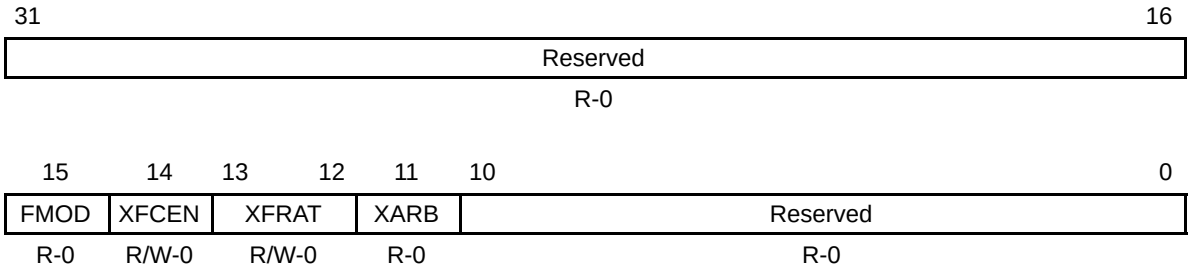
B.17 Expansion Bus (XBUS) Registers

Table B–212. Expansion Bus Registers

		Read/Write Access		
Acronym	**Register Name**	**DSP**	**Host**	**Section**
XBGC	Expansion bus global control register			B.17.1
XCECTL0-3	Expansion bus XCE space control registers			B.17.2
XBHC	Expansion bus host port interface control register	R/W	—	B.17.3
XBIMA	Expansion bus internal master address register	R/W	—	B.17.4
XBEA	Expansion bus external address register	R/W	—	B.17.5
XBD	Expansion bus data register	—	R/W	B.17.6
XBISA	Expansion bus internal slave address register	—	R/W	B.17.7

B.17.1 Expansion Bus Global Control Register (XBGC)

Figure B–202. Expansion Bus Global Control Register (XBGC)



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset

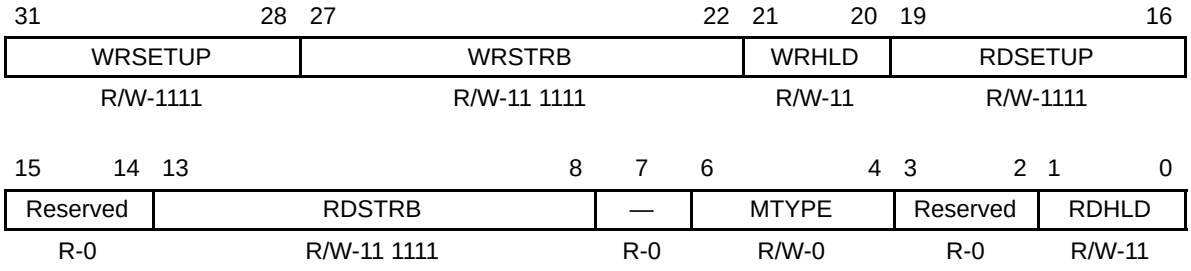
Table B–213. Expansion Bus Global Control Register (XBGC) Field Descriptions

Bit	*field*†	*symval*†	Value	Description
31-16	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
15	FMOD			FIFO boot-mode selection bit.
		GLUE	0	Glue logic is used for FIFO read interface in all XCE spaces operating in FIFO mode.
		GLUELESS	1	Glueless read FIFO interface. If $\overline{\text{XCE3}}$ is selected for FIFO mode, then XOE acts as FIFO output enable and $\overline{\text{XCE3}}$ acts as FIFO read enable. XOE is disabled in all other XCE spaces regardless of MTYPE setting in XCECTL.
14	XFCEN			FIFO clock enable bit. The FIFO clock enable cannot be changed while a DMA request to XCE space is active.
		DISABLE	0	XFCLK is held high.
		ENABLE	1	XFCLK is enabled to clock.
13-12	XFRAT		0–3h	FIFO clock rate bits. The FIFO clock setting cannot be changed while a DMA request to XCE space is active. The XFCLK should be disabled before changing the XFRAT bits. There is no delay required between enabling/disabling XFCLK and changing the XFRAT bits.
		ONEEIGHTH	0	XFCLK = 1/8 CPU clock rate
		ONESIXTH	1h	XFCLK = 1/6 CPU clock rate
		ONEFOURTH	2h	XFCLK = 1/4 CPU clock rate
		ONEHALF	3h	XFCLK = 1/2 CPU clock rate
11	XARB			Arbitration mode select bit.
		DISABLE	0	Internal arbiter is disabled. DSP wakes up from reset as the bus slave.
		ENABLE	1	Internal arbiter is enabled. DSP wakes up from reset as the bus master.
10-0	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.

† For CSL implementation, use the notation XBUS_XBGC_*field_symval*

B.17.2 Expansion Bus XCE Space Control Registers (XCECTL0–3)

Figure B–203. Expansion Bus XCE Space Control Register (XCECTL)



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset

Table B–214. Expansion Bus XCE Space Control Register (XCECTL) Field Descriptions

Bit	***field***†	***symval***†	Value	Description
31-28	WRSETUP	OF(*value*)	0-Fh	Write setup width. Number of CLKOUT1 cycles of setup time for byte-enable/address (XBE/XA) and chip enable (XCE) before write strobe falls. For asynchronous read accesses, this is also the setup time of XOE before XRE falls.
27-22	WRSTRB	OF(*value*)	0-3Fh	Write strobe width. The width of write strobe (XWE) in CLKOUT1 cycles.
21-20	WRHLD	OF(*value*)	0-3h	Write hold width. Number of CLKOUT1 cycles that byte-enable/address (XBE/XA) and chip enable (XCE) are held after write strobe rises. For asynchronous read accesses, this is also the hold time of XCE after XRE rising.
19-16	RDSETUP	OF(*value*)	0-Fh	Read setup width. Number of CLKOUT1 cycles of setup time for byte-enable/address (XBE/XA) and chip enable (XCE) before read strobe falls. For asynchronous read accesses, this is also the setup time of XOE before XRE falls.
15-14	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
13-8	RDSTRB	OF(*value*)	0-3Fh	Read strobe width. The width of read strobe (XRE) in CLKOUT1 cycles.
7	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.

† For CSL implementation, use the notation XBUS_XCECTL_*field_symval*

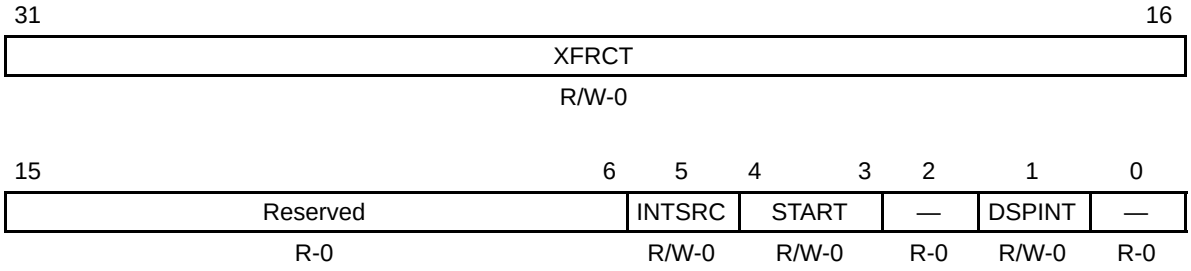
Table B–214. Expansion Bus XCE Space Control Register (XCECTL) Field Descriptions (Continued)

Bit	***field***†	***symval***†	Value	Description
6-4	MTYPE		0–7h	Memory type is configured during boot using pull-up or pull-down resistors on the expansion bus.
			0-1h	Reserved
		32BITASYN	2h	32-bit wide asynchronous interface
			3h-4h	Reserved
		32BITFIFO	5h	32-bit wide FIFO interface
			6h-7h	Reserved
3-2	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
1-0	RDHLD	OF(*value*)	0-3h	Read hold width. Number of CLKOUT1 cycles that byte-enable/address ($\overline{XBE}$/XA) and chip enable ($\overline{XCE}$) are held after read strobe rises. For asynchronous read accesses, this is also the hold time of $\overline{XCE}$ after $\overline{XRE}$ rising.

† For CSL implementation, use the notation XBUS_XCECTL_*field_symval*

B.17.3 Expansion Bus Host Port Interface Control Register (XBHC)

Figure B–204. Expansion Bus Host Port Interface Control Register (XBHC)



Legend: R = Read only; R/W = Read/Write; -*n* = value after reset

Table B–215. Expansion Bus Host Port Interface Control Register (XBHC) Field Descriptions

Bit	***field***†	***symval***†	Value	Description
31-16	XFRCT	OF(*value*)	0-FFFFh	Transfer counter bits control the number of 32-bit words transferred between the expansion bus and an external slave when the CPU is mastering the bus (range of up to 64k).
15-6	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
5	INTSRC			The interrupt source bit selects between the DSPINT bit of the expansion bus internal slave address register (XBISA) and the XFRCT counter. An XBUS host port interrupt can be caused either by the DSPINT bit or by the XFRCT counter.
		INTSRC	0	Interrupt source is the DSPINT bit of XBISA. When a zero is written to the INTSRC bit, the DSPINT bit of XBISA is copied to the DSPINT bit of XBHC.
		INTSRC	1	Interrupt is generated at the completion of the master transfer initiated by writing to the START bits.

† For CSL implementation, use the notation XBUS_XBHC_*field_symval*

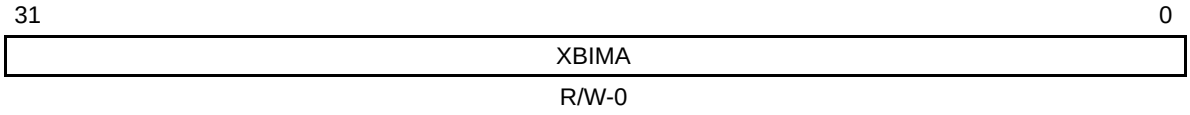
Table B–215. Expansion Bus Host Port Interface Control Register (XBHC) Field Descriptions (Continued)

Bit	*field*†	*symval*†	Value	Description
4-3	START		0–3h	Start bus master transaction bit.
		ABORT	0	Writing 00 to the the START field while an active transfer is stalled by XRDY high, aborts the transfer. When a transfer is aborted, the XBUS registers reflect the state of the aborted transfer. Using this state information, you can restart the transfer.
		WRITE	1h	Starts a burst write transaction from the address pointed to by the expansion bus internal master address register (XBIMA) to the address pointed to by the expansion bus external address register (XBEA).
		READ	2h	Starts a burst read transaction from the address pointed to by the expansion bus external address register (XBEA) to the address pointed to by the expansion bus internal master address register (XBIMA).
			3h	Reserved
2	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
1	DSPINT			The expansion bus to DSP interrupt (set either by the external host or the completion of a master transfer) is cleared when this bit is set. The DSPINT bit must be manually cleared before another one can be set.
		NONE	0	DSP interrupt bit is not cleared.
		CLR	1	DSP interrupt bit is cleared.
0	Reserved	–	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.

† For CSL implementation, use the notation XBUS_XBHC_*field_symval*

B.17.4 Expansion Bus Internal Master Address Register (XBIMA)

Figure B–205. Expansion Bus Internal Master Address Register (XBIMA)



Legend: R/W = Read/Write; -*n* = value after reset

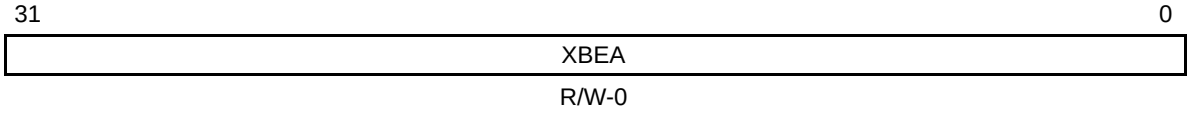
Table B–216. Expansion Bus Internal Master Address Register (XBIMA) Field Descriptions

Bit	Field	*symval*†	Value	Description
31-0	XBIMA	OF(*value*)	0-FFFF FFFFh	Specifies the source or destination address in the DSP memory map where the transaction starts.

† For CSL implementation, use the notation XBUS_XBIMA_XBIMA_*symval*

B.17.5 Expansion Bus External Address Register (XBEA)

Figure B–206. Expansion Bus External Address Register (XBEA)



Legend: R/W = Read/Write; -*n* = value after reset

Table B–217. Expansion Bus External Address Register (XBEA) Field Descriptions

Bit	Field	*symval*†	Value	Description
31-0	XBEA	OF(*value*)	0-FFFF FFFFh	Specifies the source or destination address in the external slave memory map where the data is accessed.

† For CSL implementation, use the notation XBUS_XBEA_XBEA_*symval*

B.17.6 Expansion Bus Data Register (XBD)

Figure B–207. Expansion Bus Data Register (XBD)

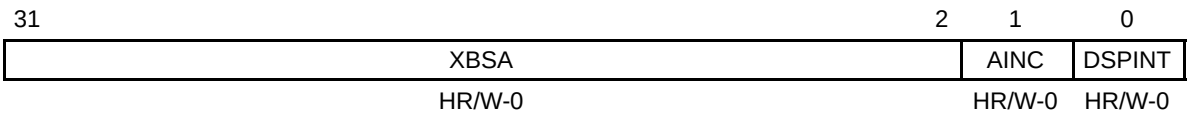
Legend: H = Host access; R/W = Read/Write; -*n* = value after reset

Table B–218. Expansion Bus Data Register (XBD) Field Descriptions

Bit	Field	Value	Description
31-0	XBD	0-FFFF FFFFh	Contains the data that was read from the memory accessed by the XBUS host port, if the current access is a read; contains the data that is written to the memory, if the current access is a write.

B.17.7 Expansion Bus Internal Slave Address Register (XBISA)

Figure B–208. Expansion Bus Internal Slave Address Register (XBISA)



Legend: H = Host access; R/W = Read/Write; -*n* = value after reset

Table B–219. Expansion Bus Internal Slave Address Register (XBISA) Field Descriptions

Bit	Field	Value	Description
31-2	XBSA	0-3FFF FFFFh	This 30-bit word address specifies the memory location in the DSP memory map being accessed by the host.
1	AINC		Autoincrement mode enable bit. (Asynchronous mode only)
		0	The expansion bus data register (XBD) is accessed with autoincrement of XBSA bits.
		1	The expansion bus data register (XBD) is accessed without autoincrement of XBSA bits.
0	DSPINT		The external master to DSP interrupt bit. Used to wake up the DSP from reset. The DSPINT bit is cleared by the corresponding DSPINT bit in the expansion bus host port interface control register (XBHC).

Appendix C

How to Use CSL Using GUI

This chapter describes how to use the TMS320C6000™ CSL graphic user interface (GUI) integrated into the DSP/BIOS™ Configuration Tool. It describes a detailed workflow for setting and accessing the CSL GUI data via the generation of C-code files.

This chapter includes a complete procedure for creating configuration objects and generating peripheral pre-initialization.

Section C.4 of this chapter provides an example of configuring a CSL peripheral using GUI.

Topic **Page**

C.1 Overview

The CSL provides a GUI that is part of the DSP/BIOS Configuration Tool integrated into Code Composer Studio IDE.

The CSL GUI allows you to view and set the chosen register settings, determine which flags/options have been set by a particular mode selection, and most importantly, create the code for the configuration settings automatically and store it in a C source file that can be integrated directly into your application.

The Configuration Tool provides static definitions and observations at compile time. The CSL objects may be changed at runtime via user code and the changes can be viewed dynamically via the Code Composer Studio debugger.

The CSL GUI is not supported for TMS320C6713 or TMS320DA610.

C.2 Introduction to the DSP/BIOS Configuration Tool: CSL Tree

The DSP/BIOS Configuration Tool allows you to access the CSL GUI and to configure some of the on-chip peripherals. Each peripheral is represented as a subdirectory of the CSL Tree as shown in Figure C–1.

The following steps are needed to configure a peripheral using the DSP/BIOS Configuration Tool:

Step 1: Create the DSP/BIOS configuration file (.cdb file); File → New → DSP/BIOS Configuration (this is also known as selection of the cdb seed).

Step 2: Configure the on-chip peripherals by the user through the CSL hierarchy tree.

Step 3: Automatically generate the C-code files when saving the configuration file. The source C-code files are included into the “Generated files” folder when the cdb file is added to the project : Project → Add file to project.

Step 4: Include the header file *cdbname*cfg.h in at least one of your source files in order to access the functions/variables defined in the source C-code.

Step 5: Add the linker command file associated to the cdb file to the project manually: Project → Add file to Project

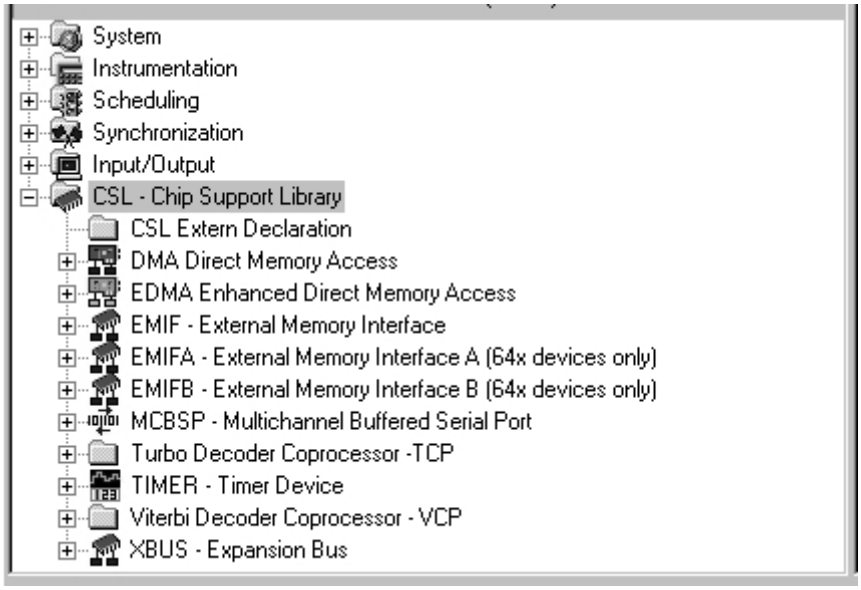
C.2.1 CSL Tree Definition

The CSL tree is used to create peripheral configuration objects and to pre-open and pre-initialize the peripherals. The configuration and handle objects and the functions PER_open() and PER_config() are implemented in the output C files. These files are generated by the DSP/BIOS Configuration Tool after saving the .cdb file.

The CSL tree is composed of two subsections:

- CSL extern declaration — allows the symbolic addressing used by the DMA and EDMA registers.
- CSL modules — dedicated to the peripherals which require an initialization before running.

Figure C–1. CSL Tree



C.2.2 CSL Extern Declaration

This folder allows symbolic addresses to be used for DMA and EDMA address registers. The symbolic addresses can come from one of these two sources:

- A user's header file
- An extern declaration object

C.2.2.1 User's Header File

The user's header file is entered under the properties of CSL Extern Declaration. The filename must be typed between quotes.

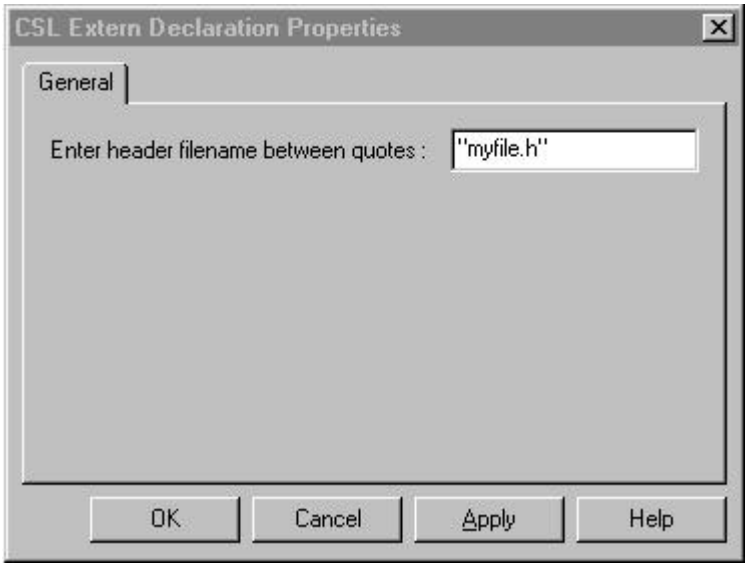
Ex. "myfile.h" will generate `#include "myfile.h"` in the CSL source file.

The user's file may contain constants, variables, additional include files which will be used by the CSL generated file _cfg_c.c or by the user's main.c file.

Example of users' header file:

```
#include <csl_mcbsp.h>

#define BUFFSIZE 256
#define COUNTER 32

extern far Uint32 buffa[]
extern far Uint32 buffb[]
```

Figure C–2. User's Header File Entry



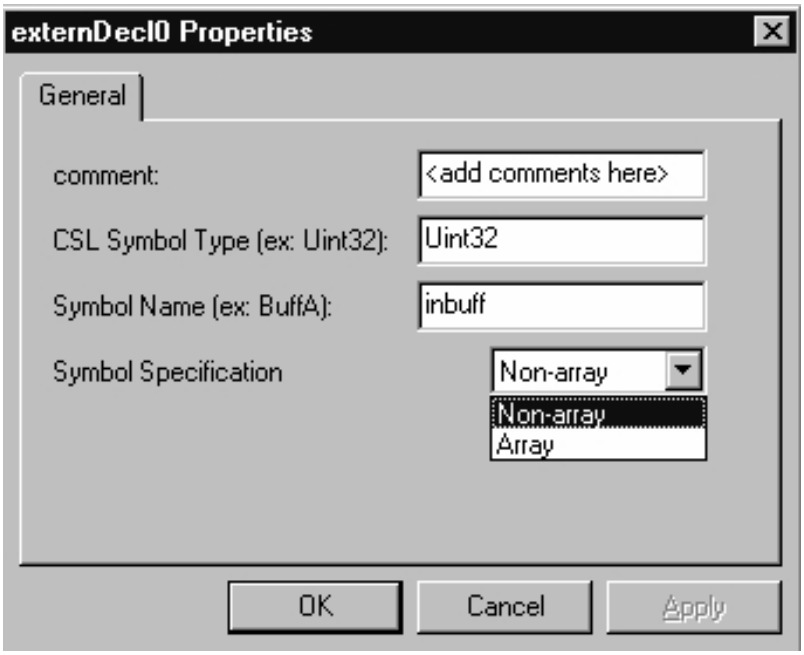
C.2.2.2 Extern Declaration Object

The Create Extern Declaration object generates variable declarations under the C file which will be generated by DSP/BIOS. The default name is "extern-DecIx" where x is a number value which increments each time a new object is inserted. The object names are fixed and cannot be modified.

Actually, the Extern Declaration object server is used to declare a variable of specified type and this variable can be used by EDMA/DMA modules for setting the register value of a configuration structure.

For example, the source address of the EDMA register can be set to an address location "inbuff", as seen in Figure C–3. The user must insert an object and define the type, name, and specification of the symbol.

Figure C–3. Extern Declaration Object Properties Page



CSL Symbol Type: belongs to CSL type (e.g., Uint32, Uint16)

Symbol Name:
inbuff, Ping, Pong, ...

Symbol Specification:

- Non-array: the symbol is declared as a pointer and will appear with “&” in the register address
- Array: the symbol is declared as an array of undefined size

In the following generated code example, three objects were created using the graphic interface:

- externDecl0 with the field value: Uint32 — inbuff — Non-array
- externDecl1 with the field values: Uint32 — Ping — Array
- externDecl2 with the field values: Uint32 — Pong — Array

The three objects generate related code declaration directly under the C source file.

The symbol names can be used to set the source address and destination address registers of configuration structures.

Because inbuff was declared non-array, the prefix "&" is added automatically.

(See Chapter 6, *DMA Module*, and Chapter 7, *EDMA Module*, for more details about setting the symbolic addressing.)

```
/*   Include Header File  */

#include "testcdbcfg.h"
#include   "myfile.h"

extern far Uint32 inbuff;

extern far Uint32 Ping[];

extern far Uint32 Pong[];

/*  Config Structures  */
EDMA_Config edmaCfg0 = {
    0x40000000,         /*  Option  */
    (Uint32) &inbuff,   /*  Source Address – Symbolic  */
    0x00000001,         /*  Transfer Counter  */
    (Uint32) Ping,      /*  Destination Address – Symbolic  */
    0x00000000,         /*  Transfer Index  */
    0x00000000          /*  Element Count Reload and Link Address  */
};

EDMA_Config edmaCfgl = {
    0x40000000,         /*  Option  */
    (Uint32) &inbuff,   /*  Source Address – Symbolic  */
    0x00000001,         /*  Transfer Counter  */
    (Uint32) Pong,      /*  Destination Address – Symbolic  */
    0x00000000,         /*  Transfer Index  */
    0x00000000          /*  Element Count Reload and Link Address  */
};
```

Note:

In the user code, ping and pong arrays have to be defined with a specified size.

C.2.3 CSL Modules

For the TMS320C6000 DSP platform, the peripherals available in the DSP/BIOS Configuration Tool are:

- DMA
- EDMA
- EMIF
- EMIFA(B) (C64x only)
- MCBSP
- TCP (C64x only)
- TIMER
- VCP (C64x only)
- XBUS

Each peripheral is organized into several sections (see Figure C–4):

- *PERIPHERAL* **Configuration Manager** allows the user to build and configure peripheral objects using the CSL PER_Config structure (see previous chapters). The user can create multiple configuration objects, even creating more configuration objects than there are actual peripherals. The menu options allow you to rename and delete the configuration object, and to display the Dependency dialog box (Figure C–5, Figure C–6, and Figure C–7).

- *PERIPHERAL* **Resource Manager** allows the user to allocate (i.e., reserve) the on-chip device which will be used like a DMA channel, a McBSP port , or a TIMER device, and to bind a peripheral configuration to a physical peripheral (resource). Furthermore, the Resource Manager can configure a specified device with any configuration object created with the related Configuration Manager. The Resource Manager performs this task using the PER_open() and PER_config() functions. Note that there can only be one configuration bound to a particular device at initialization time. The handle objects can only be renamed, not deleted.

- *PERIPHERAL*-**specific, section-specific** peripherals contain additional resources that may need to be configured. Specific details on these can be found in the individual peripheral chapters.

Figure C–4. Expanded CSL Tree

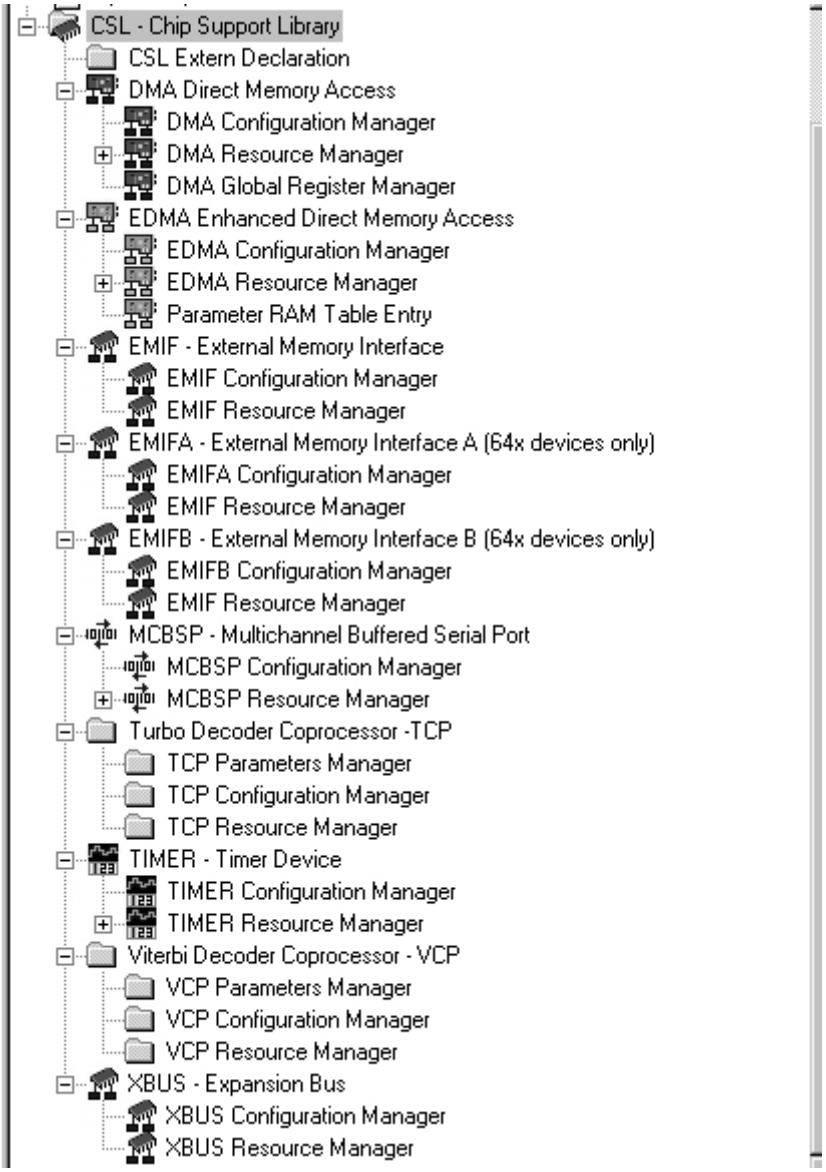
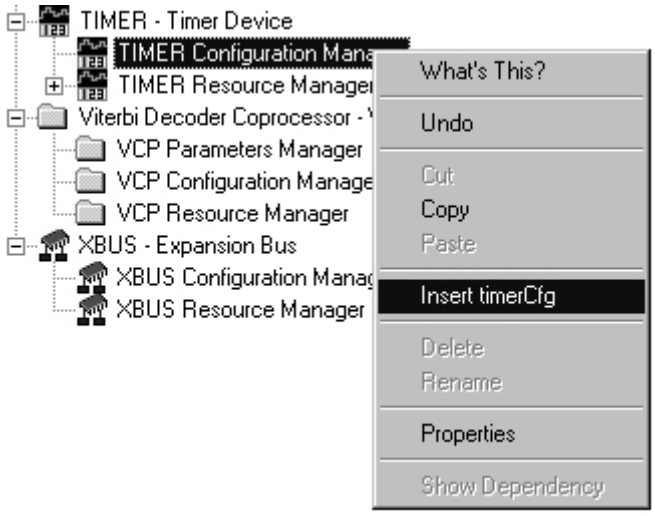
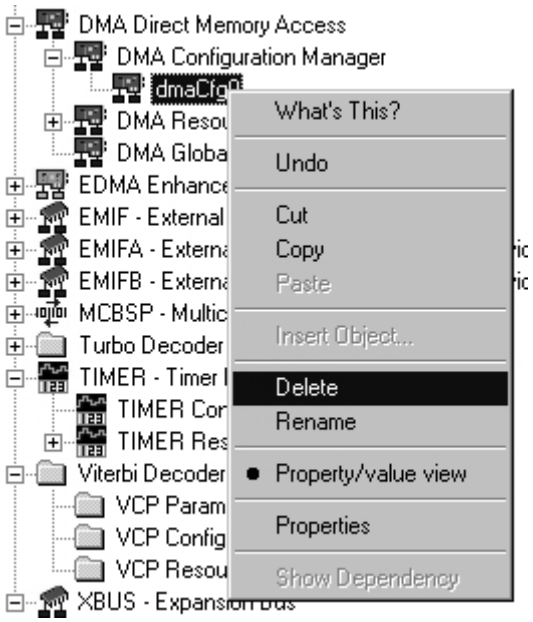


Figure C–5. Insert Configuration Object



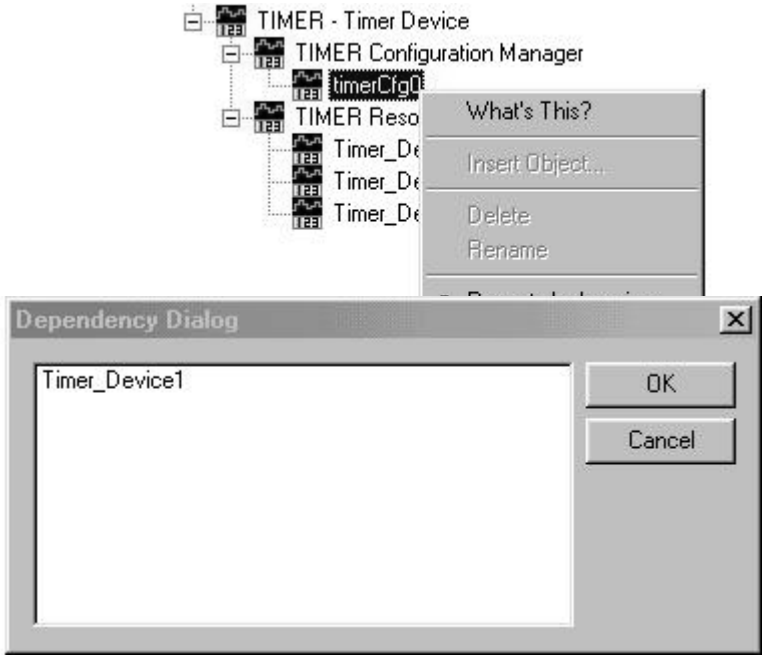
The Insert option allows the user to create the configuration objects. Once the object is created, the user can access the Properties Pages and set the register values.

Figure C–6. Delete/Rename Options



The Delete option allows you to delete a configuration object previously created using Insert. The Delete option is not accessible if the configuration is used by a physical peripheral of the associated Resource Manager.

Figure C–7. Show Dependency Option



The Show Dependency menu option shows which device/channel is using the configuration. Once a configuration has been bound to a peripheral, the Show Dependency option can be used to indicate which peripheral is using the particular configuration. In the case shown above, the “timerCfg0” configuration is bound to the physical timer device ‘Timer_Device1’ at design time (in the configuration tool).

Note:

A configuration object cannot be deleted if there is a dependency.

C.3 Generation of the CSL GUI Files (CSL APIs)

After saving the configuration file *cdbname***.cdb**, a number of files are generated by the Configuration Tool. Two of these files are dedicated to the CSL and use CSL APIs:

- Header file: *cdbname***cfg.h**
- Source file: *cdbname***cfg_c.c**

In these examples, *cdbname* is the user's cdb file name. The bold characters are attached automatically.

C.3.1 Header File *cdbname*cfg.h

The header file must be added to one of the user's source files in order to access the CSL functions and/or objects predefined in the DSP/BIOS configuration tool (see section C.4). The header file contains several elements:

- The definition of the Device Support Symbol chip set in the Global Settings subfolder of the DSP/BIOS Configuration Tool

  ```
  #define CHIP_6201 1
  ```

- The csl header files that are used by the CSL tree

  The csl.h file is included automatically.

  ```
  #include <csl.h>
  ```

  The peripheral header file is added if the user has used the related peripheral graphic interface. For example, if the McBSP folder of the CSL tree has been used, the csl_mcbsp.h header file will be automatically added.

  ```
  #include <csl_mcbsp.h>
  ```

- The declaration list of the CSL handle objects and configuration names defined in the *cdbname*.cdb. They are declared external and can be used in user code

  ```
  extern far TIMER_Config timerCfg1;
  extern far MCBSP_Config mcbspCfg0;
  extern far TIMER_Handle hTimer1;
  extern far MCBSP_Handle hMcbsp0;
  ```

  In order to access the predefined handle and configuration objects, you must include the prototypes for all functions used and declare an external reference to objects. A quick way to do this is to use the pre-generated file *cdbname*cfg.h by adding the following line to your code:

  ```
  #include "cdbnamecfg.h"
  ```

C.3.2 Source File *cdbname*cfg_c.c

The source file consists of the Include section, the Declaration section, and the Code section.

- **Include** section

  The source file has access to the data declared in the header file.

  `#include "cdbnamecfg.h"`

  `#include "myfile.h"` (user header file under CSL Extern Declaration)

Note:

If this line is added before the other csl header files (csl_emif, csl_timer, ...), you are not required to specify the device number under the compiler option (–dCHIP_6xxxx not required).

- **Declaration** section

  This section describes the configuration structures and the handle objects defined in the configuration tool.

  The values of the registers reflect the options selected through the Configuration Manager Properties dialogs of each device, as shown in Example C–1.

Example C–1. Configuration and Handle Objects of the C source file

```
/*  Config Structures */
TIMER_Config timerCfg1 = {
    0x00000000,        /*  Control Register (CTL)   */
    0x00000000,        /*  Period Register (PRD)   */
    0x00000000         /*  Counter Register (CNT)   */
};
EMIF_Config emifCfg0 = {
    0x00003078,        /*  Global Control Reg. (GBLCTL)   */
    0xFFFF3F23,        /*  CE0 Space Control Reg. (CE0CTL)   */
    0xFFFF3F23,        /*  CE1 Space Control Reg. (CE1CTL)   */
    0xFFFF3F23,        /*  CE2 Space Control Reg. (CE2CTL)   */
    0xFFFF3F23,        /*  CE3 Space Control Reg. (CE3CTL)   */
    0x0388F000,        /*  SDRAM Control Reg.(SDCTL)   */
    0x00000080,        /*  SDRAM Timing Reg.(SDTIM)   */
    0x00000000         /*(6211/6711 only) SDRAM Extended Reg.(SDEXT) */
};
/*  Handles  */
TIMER_Handle hTimer1;
```

- **Code** section

  The body is composed of a unique function, `CSL_cfgInit()`, which is automatically called by the DSP/BIOS boot.

  The function CSL_cfgInit() is where the tool implements the open and config options the user specifies in the CSL GUI. Other functions, such as enabling an EDMA channel or allocating a PARAM table, can also be implemented. See the individual peripheral chapters (chapters 2–23) for more details.

  These two main functions, PER_open() and PER_config(), are generated when the *Open* and *Enable Pre-initialization* options are checked in the Properties page of the related Resource Manager (see Figure C–8).

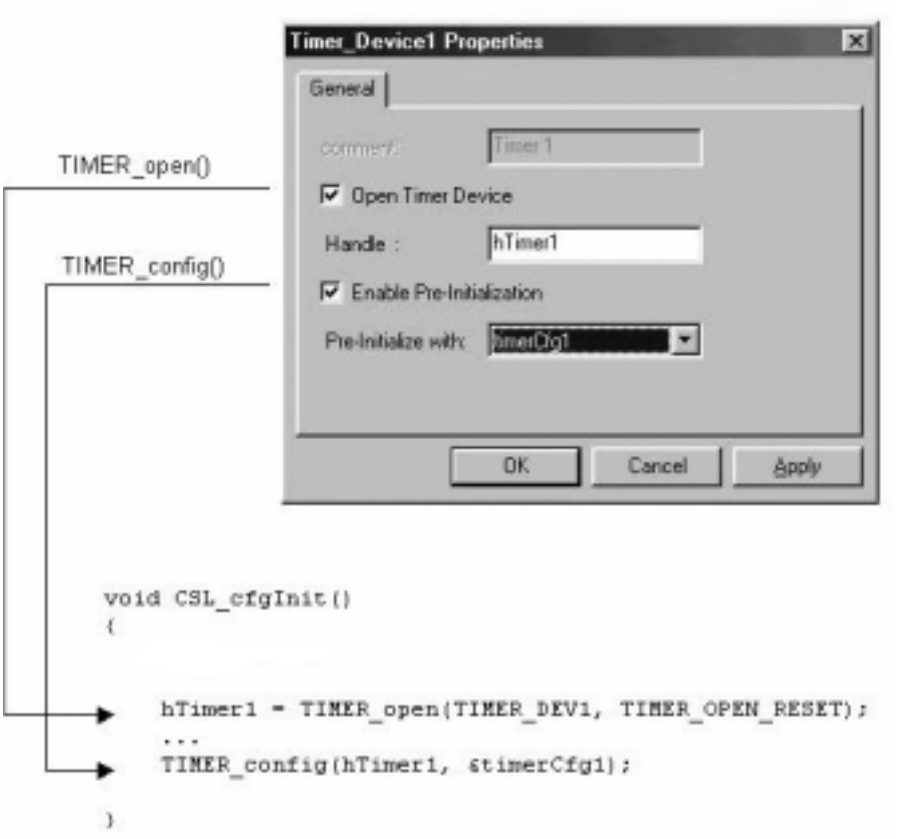
Note:

A device can be allocated without being configured.

In the example shown in Figure C–8, the configuration structure must exist for a Resource Manager device to be pre-initialized. When this is true, even if you choose not to pre-initialize a device, any configuration object created will still be defined and accessible by the user's source file (by including the header file "cdbnamecfg.h")

- If Enable Pre-initialization is checked, the `TIMER_config()` function will be generated.
- If Enable Pre-initialization is unchecked, `TIMER_config()` will not be generated, but the configuration structure `timerCfg1` will be created and available to use.

Figure C–8. Resource Manager Properties Page



C.4 Example CSL Peripheral Configuration Using GUI

This section provides an example using the 6201 device that demonstrates how to create, open and define a configuration for a TIMER device using the GUI. It also provides a full example of C files generated from a cdb file when using the chip support library (CSL) APIs.

As mentioned previously, the CSL GUI is part of the DSP/BIOS Configuration Tool and requires the following work-setting steps to access the data set graphically.

C.4.1 Adding the CSL GUI Data to the Project

Step 1: Create a cdb file:

File → New → DSP/BIOS Configuration (select a template cdb)

Default name: Config1.cdb

Step 2: Rename the .cdb file:

File → Save as: Enter the cdb name; ex., mytimer.cdb (see Figure C–9 and Figure C–10)

Step 3: Use the CSL branch for defining the peripheral objects required by your system (see section C.4.3)

Step 4: Add cdb file to your project:

Project → Add Files to Project. Select the mytimer.*cdb* file

Step 5: Add the linker command file associated with the cdb file:

Project → Add files to Project. Select mytimercfg.cmd.

Figure C–9. Code Composer Studio IDE Project Window

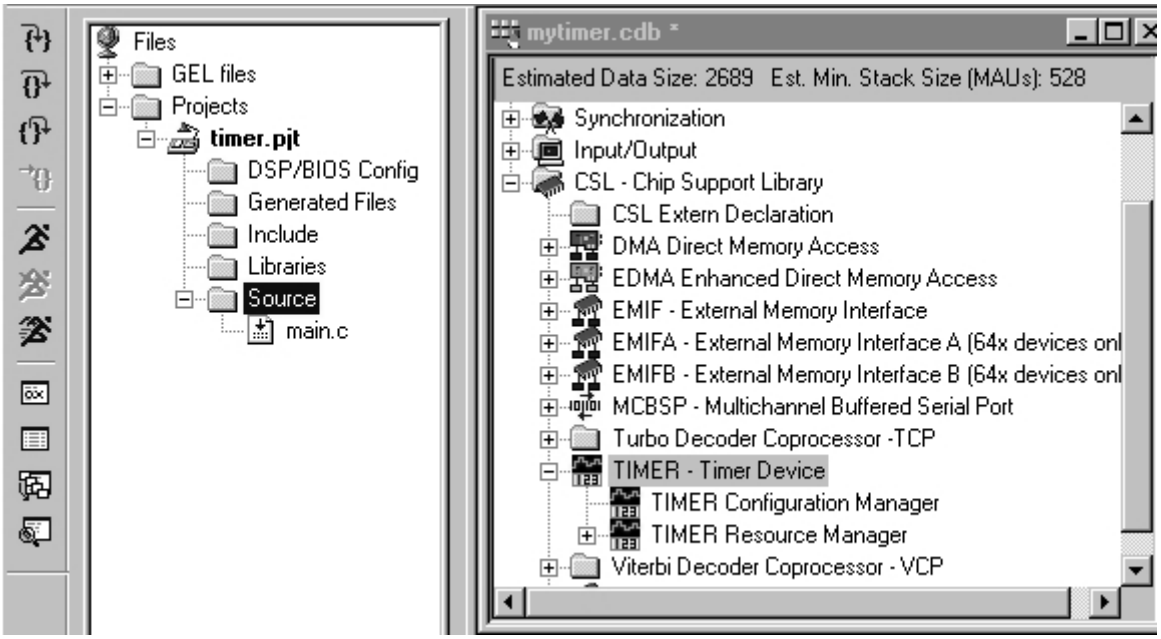
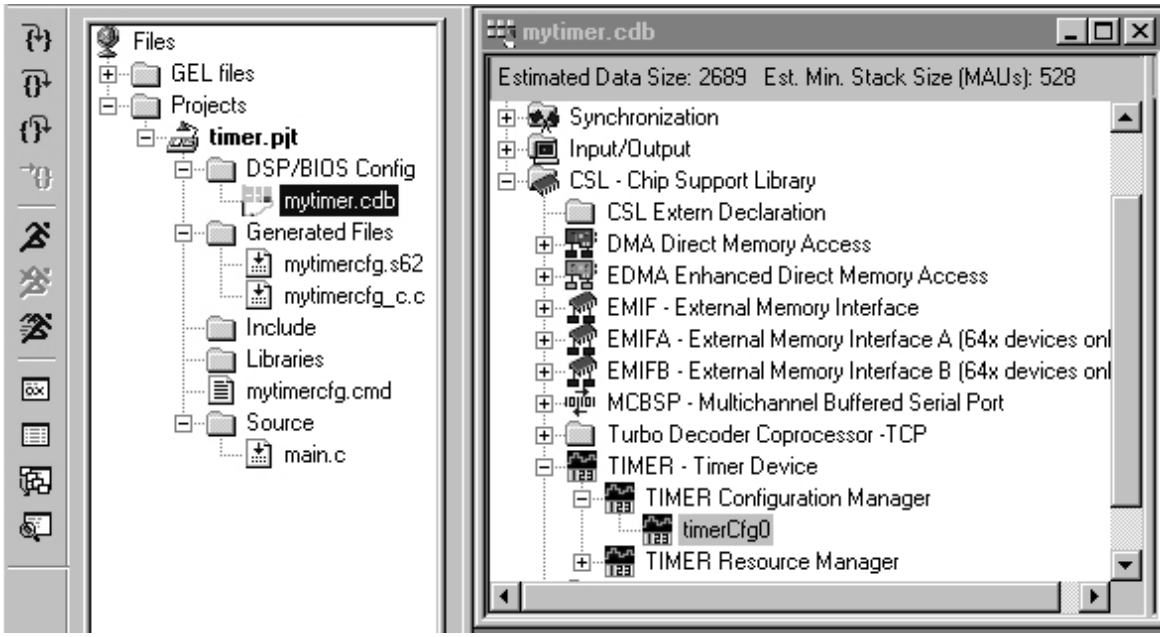


Figure C–10. Code Composer Studio IDE Project Window with cdb File Addition



C.4.2 Modifying the Code (main.c)

The following code line is required to access the CSL objects predefined by the DSP/BIOS Configuration Tool:

```
#include “mytimercfg.h”
```

This includes CSL handle and configuration declarations to the user’s code.

The header file must be included before any other CSL header files; otherwise, the –dCHIP_XXXX device support symbol has to be set in the Compiler Option of the project. (See Table 1–10, *CSL Device Support Library Name and Symbol Conventions*, on page 1-19.)

Example C–2 summarizes the basic main.c file configuration.

Example C–2. Basic main.c File Configuration

```
/* Include file */
#include “mytimercfg.h”
#include   <csl_irq.h>

/* main program */
void main()
{

...
}
```

C.4.3 Configuring the Timer1 Device

Because the default values (device power-on reset) are different from one chip to another, it is recommended that you delete any existing config objects before changing the chip type under the Global Setting (system). This insures that you get the right reset value when you want to use the default setting of the registers.

The configuration file mytimer.cdb is assumed to be created previously and opened (see section C.4 for more details).

In the mytimer.cdb Configuration Tool window, go to the TIMER module under the CSL branch. Using Figure C–11, follow these steps:

Step 1: Right-click on TIMER Configuration Manager, Insert a New Configuration Object.

In this example, the timerCfg object was renamed. Renaming was done by clicking on the timerCfg object, selecting Rename from the drop-down menu, and typing timerCfg1.

Step 2: Right-click on timerCfg1, and select Properties. Set the configuration by selecting the options on the tabbed pages.

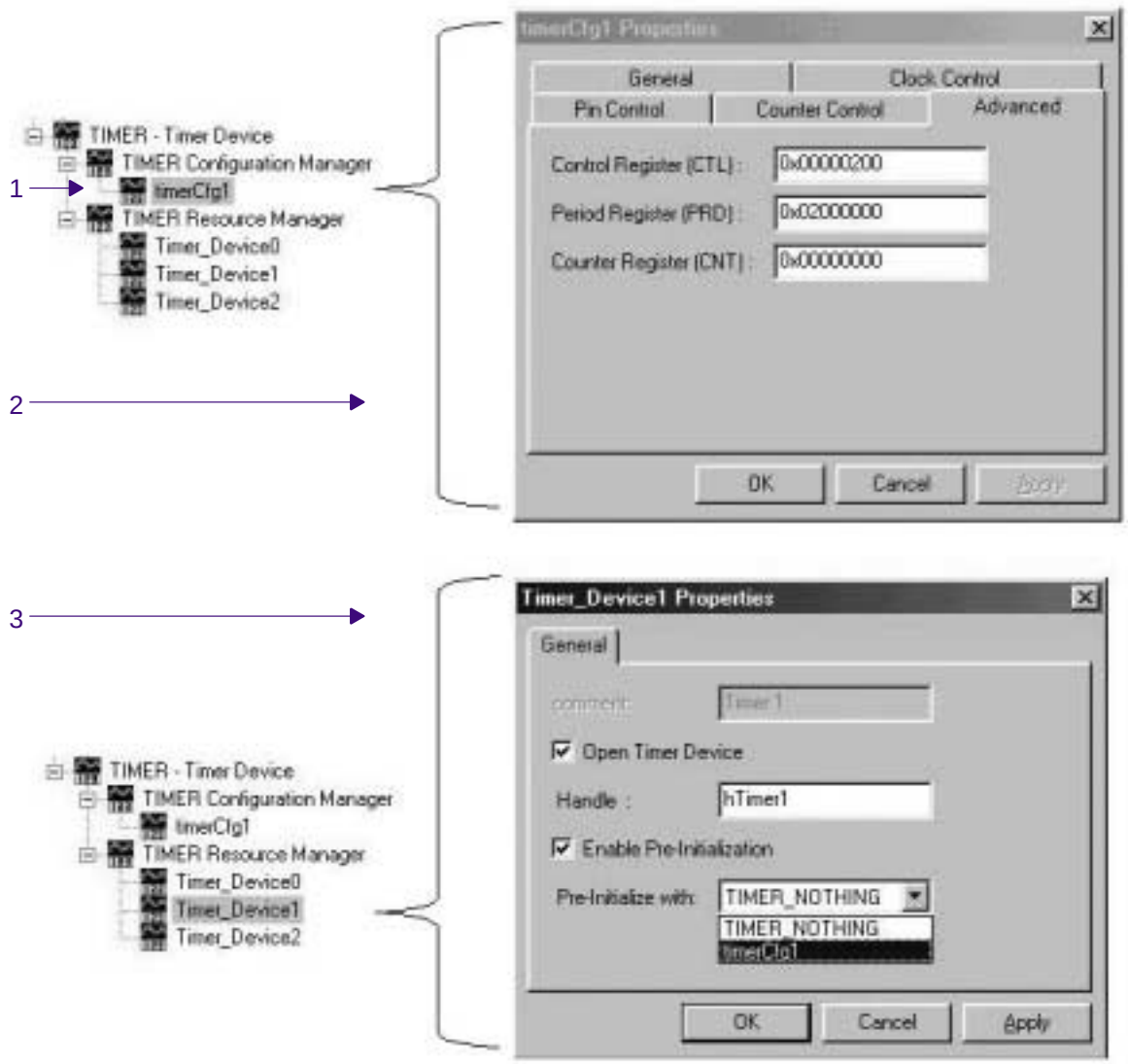
Step 3: Select TIMER Resource Manager, right-click on TIMER_Device1, and select Properties.

- ❏ Check: Open TIMER Device and Enable Pre-Initialization.
- ❏ Select the configuration timerCfg1 from the drop-down list.

Note:

It is recommended not to start the device when setting field registers. If for some reason it is necessary to start the device immediately, go to the Advanced Page and set the related bits.

Figure C–11. Configuring the Timer1 Device



C.4.4 Accessing Data Defined in Files Generated by the CSL GUI

After saving the configuration file mytimer.cdb, the following header file (mytimercfg.h) and source file (mytimercfg_c.c), both dedicated to the CSL branch, are generated (see Figure C–12 and Figure C–13).

The header file has to be included in at least one of your source files in order to access the configuration objects and peripheral pre–initializations you defined via the CSL GUI.

Figure C–12. Header File mytimercfg.h

```
/*   Do *not* directly modify this file.  It was    */
/*   generated by the Configuration Tool; any       */
/*   changes risk being overwritten.                */

/* INPUT mytimer.cdb */
#define CHIP_6201 1

/*  Include Header Files  */
#include  <std.h>
#include  <hst.h>
#include  <swi.h>
#include  <tsk.h>
#include  <log.h>
#include  <sts.h>
#include  <csl.h>
#include <csl_timer.h>

#ifdef __cplusplus
extern "C" {
#endif

extern far HST_Obj RTA_fromHost;
extern far HST_Obj RTA_toHost;
extern far SWI_Obj KNL_swi;
extern far TSK_Obj TSK_idle;
extern far LOG_Obj LOG_system;
extern far STS_Obj IDL_busyObj;
extern far TIMER_Config timerCfg1;
extern far TIMER_Handle hTimer1;
extern far void CSL_cfgInit();

#ifdef __cplusplus
}
#endif /* extern "C" */
```

CHIP device Symbol defined in the Global Setting subsection (System) of the Configuration tool

csl header files of the peripherals which have a set of data set under the CSL tree. (CSL GUI contains data set for the timer devices)

The Handle and Configuration objects are defined and can be used by other C files (user's files).

Figure C–13. Source File mytimercfg_c.c

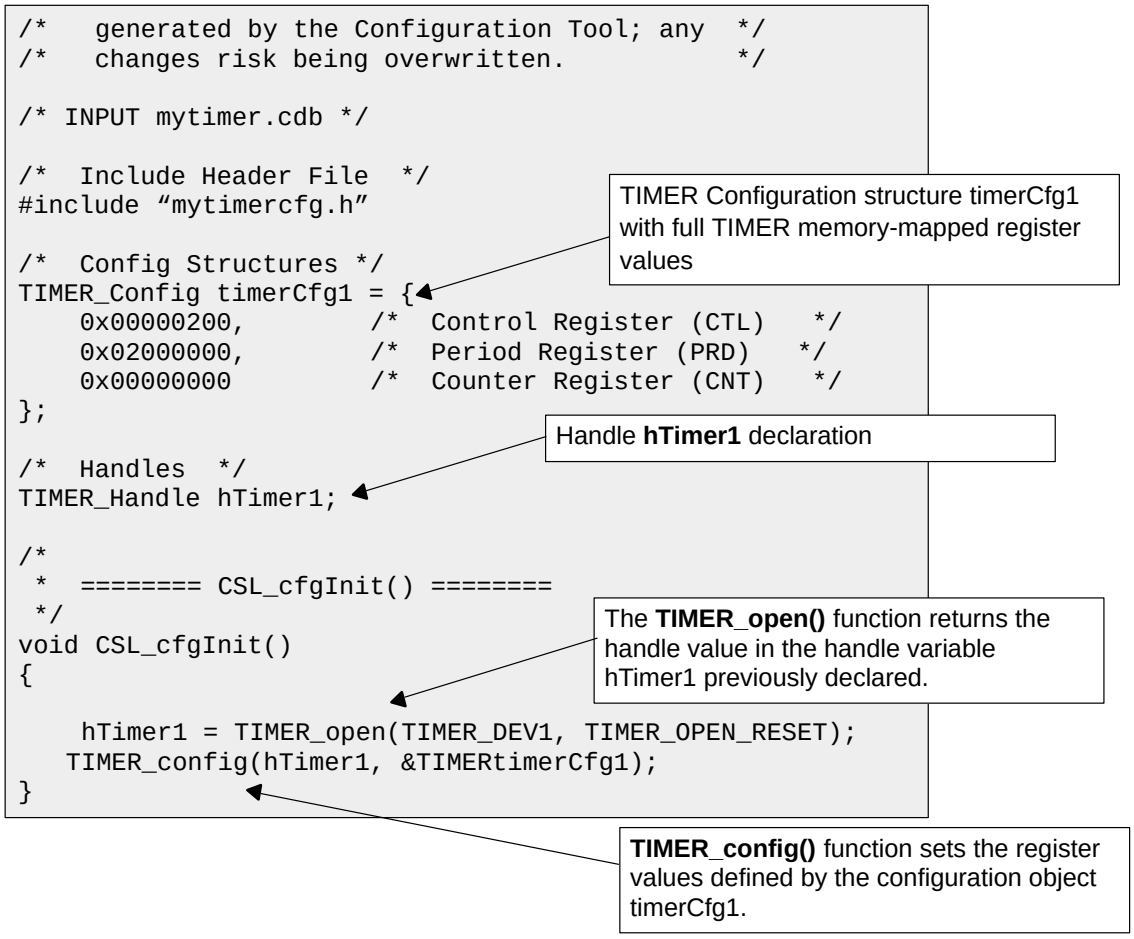
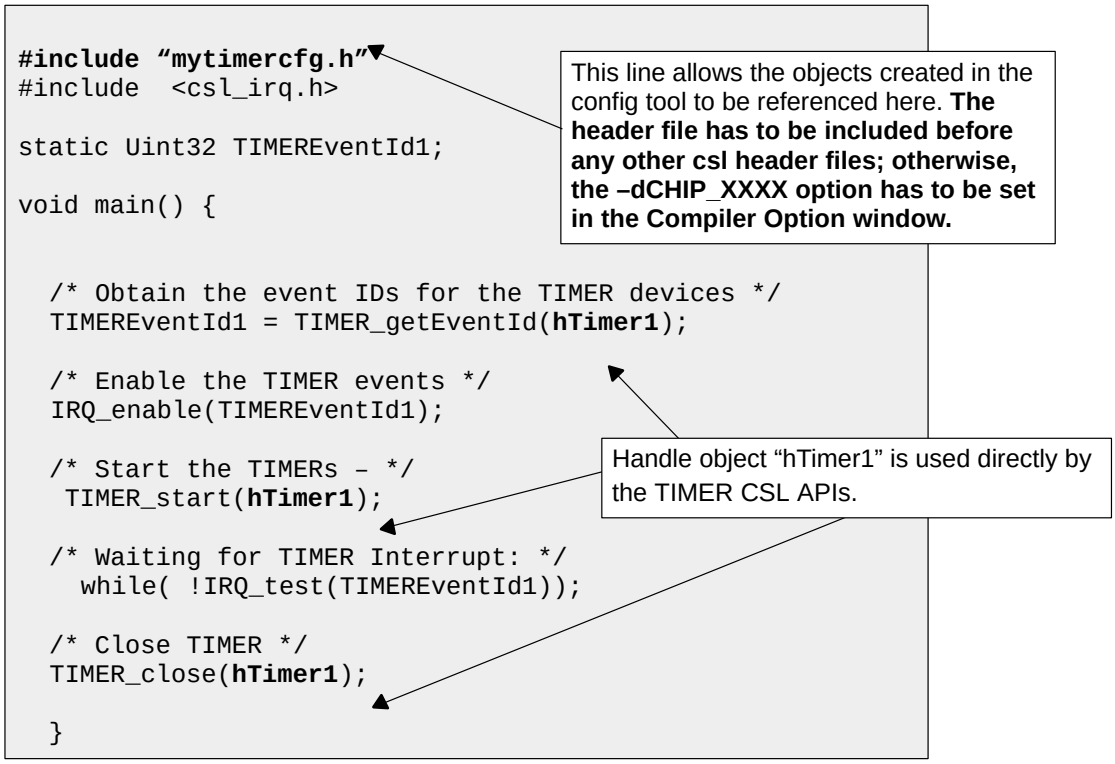


Figure C–14. Example of main.c File Using Data Generated by the Configuration Tool



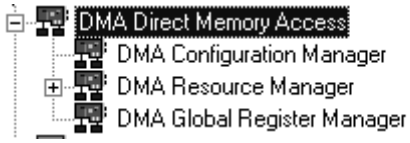
C.5 Configuring the DMA Module Using CSL GUI

The DMA module facilitates configuration of the Direct Memory Access (DMA) controller. The DMA module consists of a configuration manager, a resource manager, and a global register manager.

The configuration manager allows creation of an object that contains the complete set of register values needed to configure a DMA channel. The resource manager associates a configuration object with a specific DMA channel. The global register manager pre-initializes the global registers.

Figure C–15 illustrates the DMA sections menu on the CSL graphical user interface (GUI).

Figure C–15. DMA Sections Menu



The DMA module includes the following three sections:

- **DMA Configuration Manager** allows you to create configuration objects by setting the memory-mapped registers related to the DMA.
- **DMA Resource Manager** allows you to select a DMA channel and to associate a configuration object to this channel . The four channel handle objects are predefined.
- **DMA Global Register Manager** allows you to pre-initialize the global registers, A, B, C, and D.

C.5.1 DMA Configuration Manager

The DMA Configuration Manager allows you to create DMA Channel configurations through the Properties page and to generate the configuration objects.

C.5.1.1 Inserting a Configuration Object

There is no predefined configuration object available.

To configure a DMA channel through the memory-mapped registers, use the drop-down menu to insert a new configuration object by right-clicking on the DMA Configuration Manager and selecting Insert dmaCfg. The configuration objects can be renamed. Their use depends on the on-chip device resources.

Because four channels are available, a maximum of four configurations can be used simultaneously.

Note:

The number of configuration objects is unlimited. You can create several configurations select the right one for a specific port, and change the configuration later just by selecting a new one under the DMA Resource Manager. This feature provides you with more flexibility and reduces the time required to modify register values. The same configuration can be used by different channels.

C.5.1.2 Deleting/Renaming a Configuration Object

To delete or rename an object, use the drop-down menu by clicking on the configuration object to be deleted or renamed.

If a configuration object is used by one of the predefined handle objects of the DMA Resource Manager, the Delete and Rename options are grayed-out and non-usable. The Show Dependency option is accessible and shows which device is using the configuration object. See section C.2, *Introduction to DSP/BIOS Configuration Tool: CSL Tree,* on page C-3.

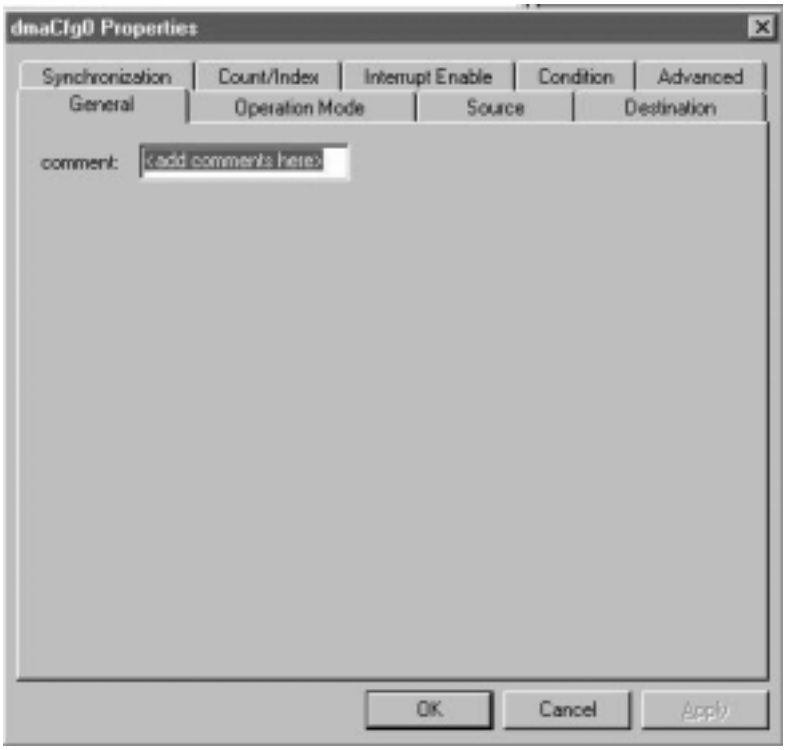
C.5.1.3 Configuring the Object Properties

You can configure object properties through the Properties dialog box. (See Figure C–16). To access the Properties dialog box, right-click on a configuration object and select Properties. By default, the General page of the Properties dialog box is displayed.

The Properties pages allow you to set the memory-mapped registers related to the DMA. The configuration options are divided into the following tab pages:

- Operation Mode
- Source: Source Address, Increment, Reload
- Destination: Destination Address, Increment, Reload
- Synchronization: Synchronization Control
- Count/Index: Selection of Global registers and Count values
- Interrupt Enable: Interrupt Control
- Condition: Clear condition bits
- Advanced Page: This page contains the full hexadecimal register values and reflects the setting of the previous tab pages. Also, the full register values can be entered directly and the new options will be mirrored on the related tab pages automatically.

Figure C–16. DMA Properties Page



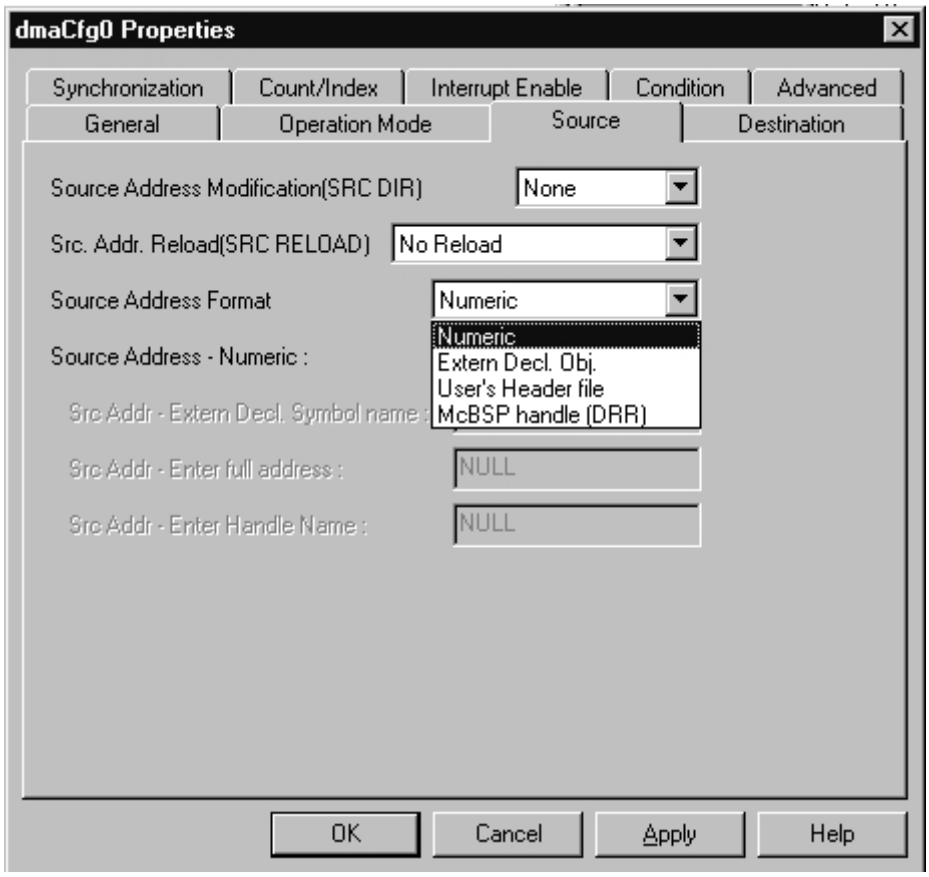
Each tab page is composed of several options that are set to a default value (at device power-on reset).

The options represent the fields of the DMA registers; the associated field name is shown in parenthesis. For further details on the fields and registers, refer to the *Direct Memory Access (DMA) Controller* chapter in the *TMS320C6000 Peripherals Reference Guide* (SPRU190).

C.5.1.4 Specifying Address Formats

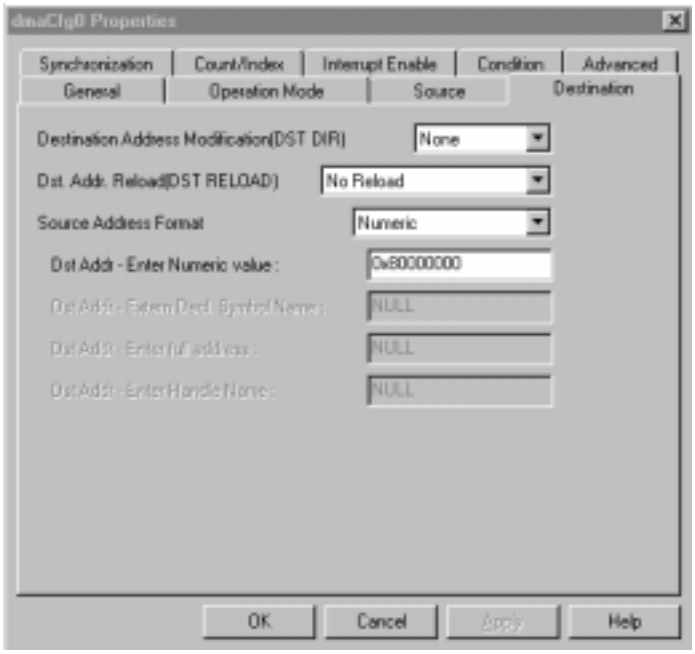
The source and destination address values can be set four different ways: numeric, extern declaration object (extern decl. obj.), user's header file, and McBSP handle (see Figure C–17).

Figure C–17. DMA Configuration Dialog Showing Four Address Types



- **Numeric:** Hexadecimal address value. This value is also reported on the DMA configuration properties “Advanced” tab (see Figure C–18).

Figure C–18. Specifying an Address as Numeric Type



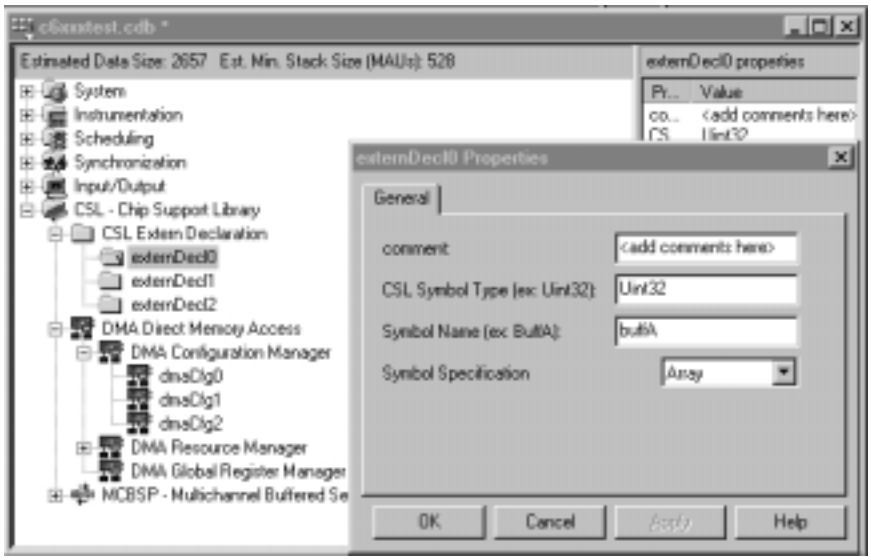
- **Extern Declaration Object (Extern Decl. Obj.):** A symbol previously defined as a CSL Extern Declaration may be used to represent an address.

Notes:

- If the extern symbol referenced in the Extern Declaration Object is not defined elsewhere in your project, an error will occur at compilation time.
- The name entered in the Address must exactly match the symbol name field defined in the Extern Declaration Object.

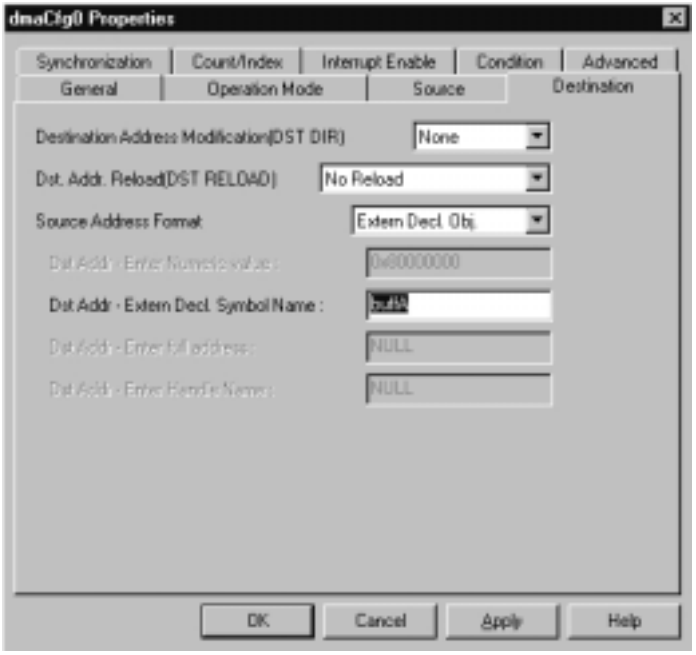
As shown in Figure C–19 , `buffA` is the symbol name of an array defined under CSL Extern Declaration.

Figure C–19. Creating an Extern Declaration Object (Referencing an External Symbol)



With `buffA` defined as an extern, it may be used as a symbolic destination address (see Figure C–20).

Figure C–20. Specifying an Address Using an Extern Declaration Object



With buffA defined as external and referenced as the destination address, the DSP/BIOS Configuration Tool will automatically create the following DMA Configuration object (dmaCfg0) in code:

```
extern far Uint32 buffA[];
DMA_Config dmaCfg0 = {
    0x00000000,     /*  Primary Control Register  */
    0x00000080,     /*  Secondary Control Register  */
    0000000000,     /*  Source Address – From User's Header File  */
    (Uint32) buffA, /*  Destination Address – From User's Header file  */
    0x00000000     /*  Transfer Counter – Numeric  */
};
```

- **User's Header File:** Symbols defined in a header file may be used as a source or destination address. When the *User's Header file* option is selected, the field "Enter full address" is made accessible.

Notes:

- The header file must first have been referenced under the CSL Extern Declaration.
- If the symbol is not defined, an error will occur at compilation time.
- If you referenced the header file in any other C file, you must be careful to prevent defining any global variables multiple times. #IFDEF can be used to prevent this from occurring.

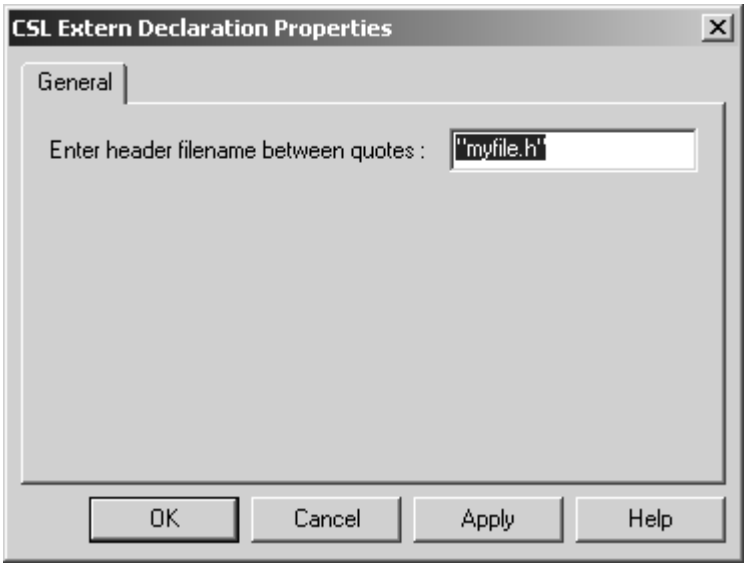
Example C–3 is an example user header file called *myfile.h*.

Example C–3. User's Header File for Source and Destination Address Setting

```
// myfile.h
#define     BUFFSIZE     256
#define     myESIZE      0x0004   //element size of 4 bytes
Uint32      myBuffer[BUFFSIZE];
```

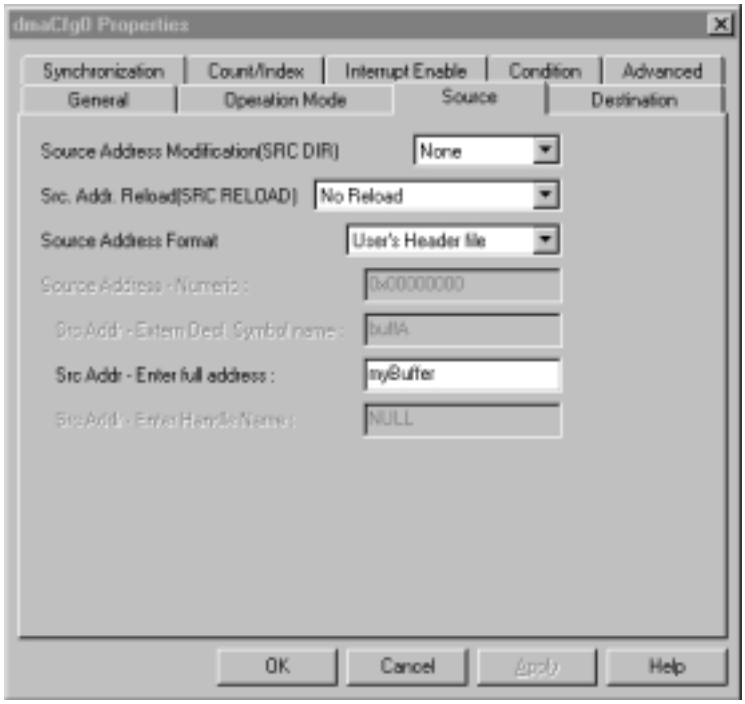
To allow the CSL graphical interface to reference the header file, right-click on the *CSL Extern Declaration* folder and choose *Properties*. Type the header file (along with quotes) into the dialog box as shown in Figure C–21.

Figure C–21. Referencing a User's Header File within the CSL GUI



With myfile.h declared, the symbols defined in the header file may now be used within CSL configurations. Figure C–22 uses myBuffer as the DMA source address.

Figure C–22. Specifying a Symbol from a User's Header File



Similar to the *Extern Declaration Object,* the *User's Header file* symbol will also be set directly within DMA configuration object:

```
#include "myfile.h"
extern far Uint32 buffA[];

/* Config Structures */
DMA_Config dmaCfg0 = {
    0x00000000,        /*  Primary Control Register  */
    0x00000080,        /*  Secondary Control Register  */
    (Uint32) myBuffer, /*  Source Address – From User's Header File  */
    (Uint32) buffA,    /*  Destination Address – Extern Decl. Obj.   */
    0x00000000         /*  Transfer Counter – Numeric  */
};
```

- **McBSP Handle (DRR) or (DXR):** Allows the McBSP receive or transmit register to be set as either the source or destination address for the DMA configuration.

  McBSP serial ports are commonly used as either a source or destination for DMA transfers. For example, if an analog-to-digital converter (ADC) is

connected to one of the McBSPs, the DMA may be used to move the digital samples into on-chip memory for processing. In this case, the DMA source address needs to be set to the McBSP receive register (DRR) address. The McBSP handle option has been provided to prevent the need to look up the numerical address from the peripherals reference guide and to provide greater portability in the event another device is chosen in the CDB file's "General Settings" (i.e., improved portability).

To use the McBSP handle option, follow these steps:

1) Allocate the McBSP by right-clicking on one of the McBSP ports (found under the McBSP Resource Manager CSL tree) and choosing Properties. Click on the Open McBSP Port and define a handle name in the appropriate text box (or use the default handle name). Click OK to save the changes and Close.

2) Open the DMA configuration object to be associated with the McBSP.

3) Choose either the Source or Destination tab of the DMA configuration, whichever is appropriate.

4) If configuring the source address, select McBSP handle (DRR) from the Source Address format field. Choose the similar McBSP handle (DXR) option if configuring the destination address.

5) When the McBSP handle format is chosen, the Enter Handle Name field becomes accessible. Enter the handle name chosen in step 1.

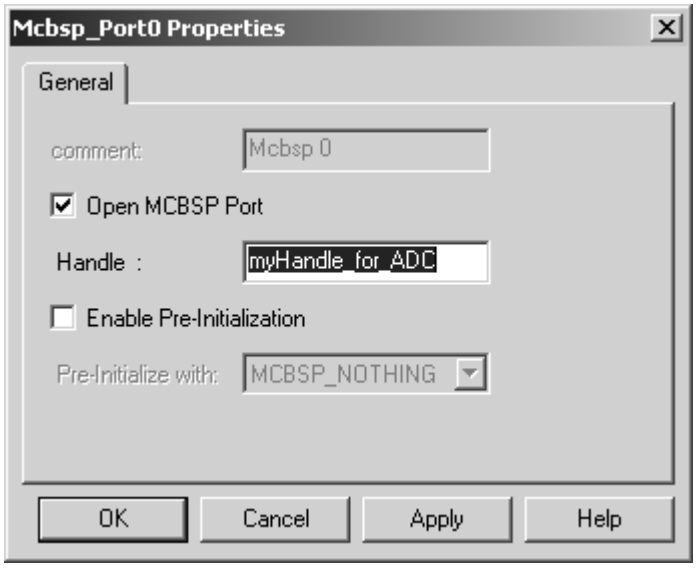
Note:

The handle name used in the DMA Enter Handle Name field must match the McBSP name exactly or a compilation error will occur.

The following is an example implementation of the ADC –> McBSP –> DMA example from the above *McBSP handle (DRR)* description.

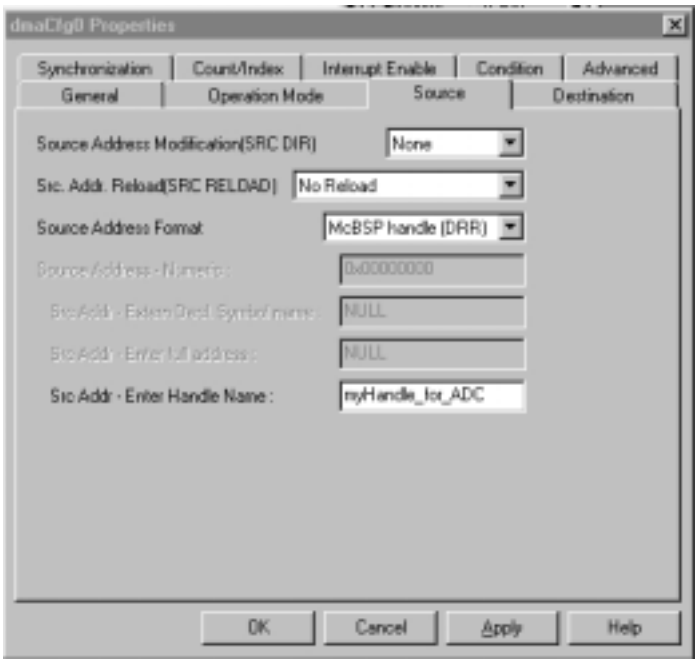
❑ Open the McBSP port and define a handle (see Figure C–23).

Figure C–23. Requesting to Open a McBSP Port and Provide a Handle



- ❑ Open the DMA configuration to the Source tab (see Figure C–24).
- ❑ Choose the McBSP handle type.
- ❑ Enter the handle name.

Figure C–24. Specifying a DMA Source Address with a McBSP Handle



As with the previous examples, the Configuration Tool automatically configures the DMA configuration object. In this case the source (or destination) value is not directly specified in the DMA configuration object, instead, it must be initialized in code. The Configuration Tool creates the `CSL_cfgInit()` function to provide this type of code. The `CSL_cfgInit()` function is automatically run before `main()` is called (see Example C–4).

Example C–4. Specifying a DMA Source Address with a McBSP Handle

In this case, the “source” member value is set to a default numeric value (zero).

The “src” member value of dmaCfg0 is updated with McBSP DRR register address using CSL’s _RMK() macro.

```
/*  Config Structures */
DMA_Config dmaCfg0 = {
    0x00000000,        /*  Primary Control Register  */
    0x00000080,        /*  Secondary Control Register  */
    0x00000000,        /*  Source Address – Numeric  */
    0x80000000,        /*  Destination Address – Numeric  */
    0x00000000         /*  Transfer Counter – Numeric  */
};
/*  Handles  */
DMA_Handle hDma0;
MCBSP_Handle myHandle_for_ADC;
/*
 * ========CSL_cfgInit()=========
 */
void CSL_cfgInit()
{
    hDma0 = DMA_open(DMA_CHA0, DMA_OPEN_RESET);
    myHandle_for_ADC = MCBSP_open(MCBSP_DEV0, MCBSP_OPEN_RESET);
    dmaCfg0.src = DMA_SRC_RMK(myHandle_for_ADC->drrAddr);
}
```

C.5.1.5 Transfer Count Register Setting

The counter register value can be set two different ways:

- **Numeric:** Hexadecimal address value. This value is also reported on the EDMA configuration properties Advanced tab.
- **User’s Header file:** Symbols defined in a header file may be used for setting the transfer-count and/or Index fields. When the *User’s Header file* option is selected, the Enter Num or Symbol value field is made accessible.

Notes:

- ❑ The header file must first be referenced under the CSL Extern Declaration.
- ❑ If the symbol is not defined, an error will occur at compilation time.
- ❑ If you referenced the header file in any other C file, you must be careful to prevent defining any global variables multiple times. #IFDEF can be used to prevent this from occurring.

Using symbols from header files can be very convenient. For example, a header file can be created to #define a symbol, such as BUFFSIZE in the header file shown in Example C–5.

Example C–5. User's Header File Example for Transfer Count Register Setting

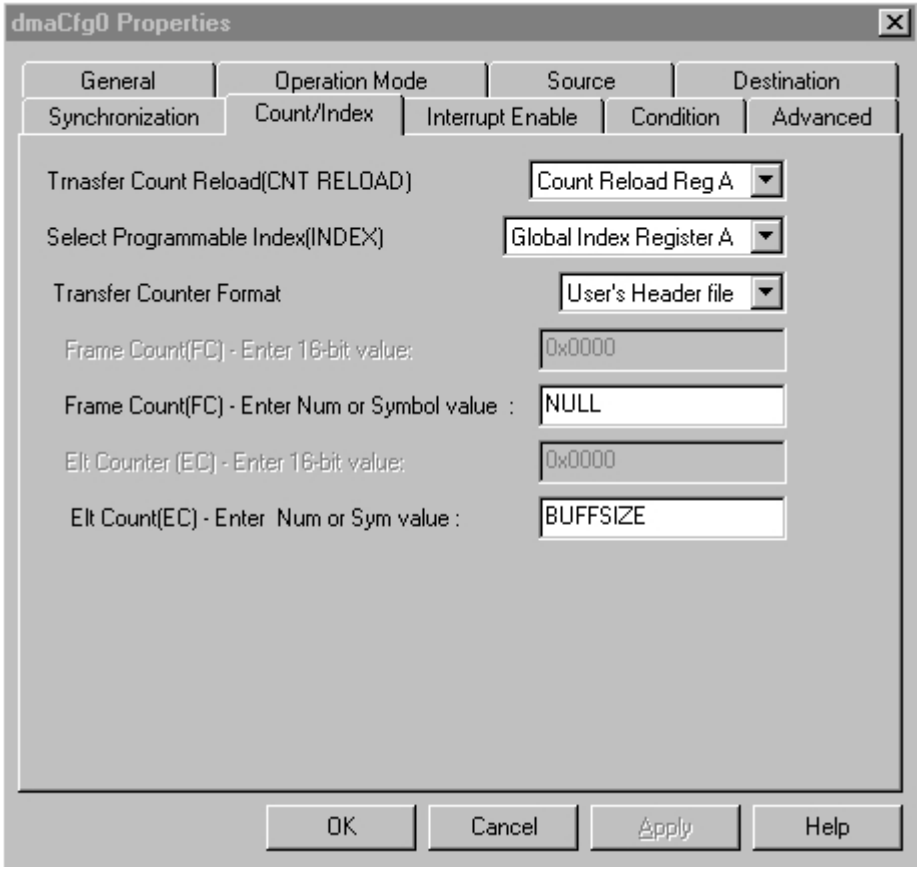
```
// myfile.h
#define     BUFFSIZE     256
#define     myESIZE      0x0004   //element size of 4 bytes
Uint32      myBuffer[BUFFSIZE];
```

(Note that this is the same header file as shown in the *User's Header File for Source and Destination Address Setting*, Example C–3).

If you added the appropriate #IFDEF's to myfile.h, it could be used by an entire project team to define project-wide symbols.

In this example, the BUFFSIZE value is used to set the DMA configuration object's element count. In other words, this DMA configuration will transfer BUFFSIZE number of elements.

Figure C–25. Using a Symbol from a User's Header File to Specify the DMA Element Transfer Count



Transfer Count page/tab: FRAME counter (FC) is not used "NULL"; ELEMENT counter (EC) is set to BUFFSIZE.

Because the element transfer count and index value do not fill up an entire register, the values cannot be set directly in the DMA configuration object. Instead, they are set in the `CSL_cfgInit()` function.

Example C–6. Using a Symbol from a User's Header File to Specify the DMA Element Transfer Count

```
#include "myfile.h"
extern far Uint32 buffA[];
/*  Config Structures */
DMA_Config dmaCfg0 = {
    0x00000000,        /*  Primary Control Register  */
    0x00000080,        /*  Secondary Control Register  */
    0x00000000,        /*  Source Address – Numeric  */
    (Uint32) buffA,    /*  Destination Address – Extern Decl. Obj  */
    0x00000000         /*  Transfer Counter – Numeric  */
};
/*  Handles  */
MCBSP_Handle myHandle_for_ADC;

/*
 *  ========CSL_cfgInit()=========
 */
void CSL_cfgInit()
{
   myHandle_for_ADC = MCBSP_open(MCBSP_DEV0, MCBSP_OPEN_RESET);
   dmaCfg0.xfrcnt = DMA_XFRCNT_RMK(0,BUFFSIZE);
   dmaCfg0.src = DMA_SRC_RMK(myHandle_for_ADC–>drrAddr);
}
```

The "counter" object member is set to a default numeric value (zero).

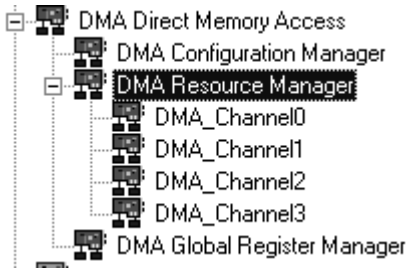
The dmaCfg0 configuration object's "xfrcnt" value is updated via the _RMK() macros using the symbols defined in "myfile.h"

C.5.2 DMA Resource Manager

The DMA Resource Manager allows you to generate the `DMA_open()` and `DMA_config()` CSL functions.

Figure C–26 illustrates the DMA Resource Manager menu on the CSL GUI.

Figure C–26. DMA Resource Manager Menu



C.5.2.1 Predefined Objects

The four channel handle objects are predefined and each is associated with a supported on-chip DMA channel.

- **DMA_Channel0** – Default handle name: hDma0
- **DMA_Channel1** – Default handle name: hDma1
- **DMA_Channel2** – Default handle name: hDma2
- **DMA_Channel3** – Default handle name: hDma3

C.5.2.2 Properties Page

The Properties page is accessible by right-clicking on the drop-down menu option Properties.

The first time the Properties page appears, only the Open DMA Channel check-box can be selected.

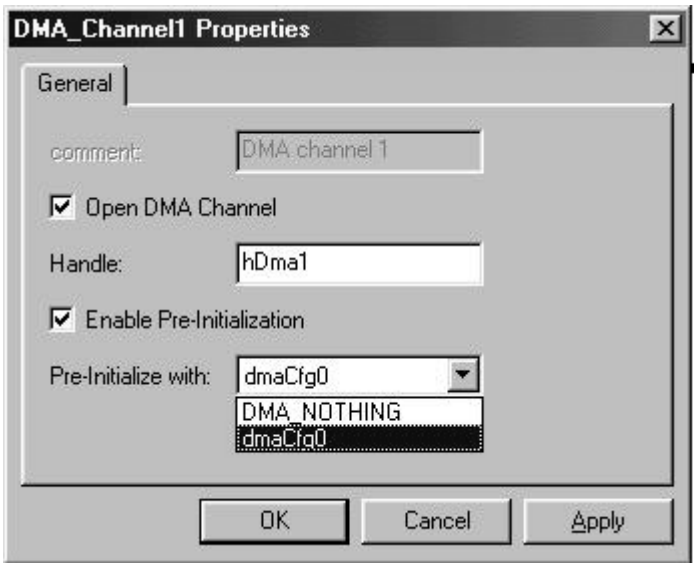
DMA_NOTHING is used to indicate that there is no configuration object selected for this serial port.

To enable Enable Pre-Initialization, create a new configuration object first (see section C.5.1). When a configuration object is available, Enable Pre-Initialization can be checked and you can select a configuration object in the drop-down list.

If DMA_NOTHING remains selected, no configuration object will be generated for the related DMA handle (see section C.5.4).

Figure C–27 shows that the Open DMA Channel option is checked and the handle object hDma1 is now accessible (renaming allowed). The `DMA_open( )` function will be generated with hDma1 containing the returned handle address.

Figure C–27. DMA Properties Page With Handle Object Accessible



C.5.3 DMA Global Register Manager

C.5.3.1 Properties Page

Since the global registers are shared, they must be controlled using resource management. This is done by `DMA_globalAlloc()`.

The Global Register Manager Properties Page, shown in Figure C–28, allows you to generate the `DMA_globalAlloc()` and `DMA_globalConfig()` CSL functions.

The Properties page is accessible by right-clicking on the drop-down menu option Properties.

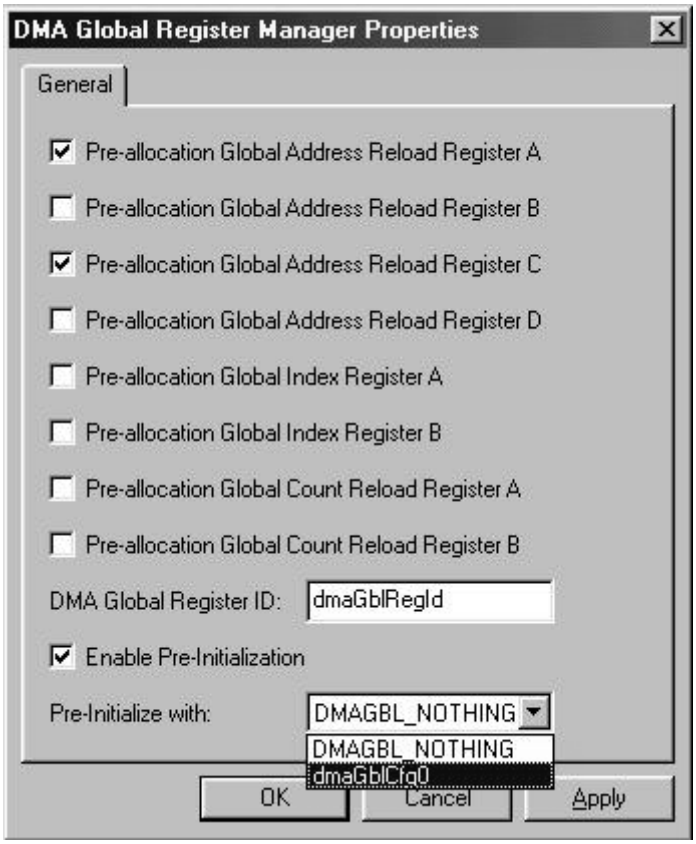
The first time the Properties page appears, no preallocation Global register is defined. Also, DMAGBL_NOTHING is used to indicate that there is no configuration object associated with the selected Global registers.

The DMA Global register ID parameter contains the returned value of the `DMA_globalAlloc()` API.

The register ID variable is the result of Global register combination.

Prior to selecting Enable Pre-Initialization, create a new configuration object first (see section C.5.3.2). When a configuration object is available, Enable pre-initialization can be checked and you can select a configuration object in the drop-down list.

Figure C–28. DMA Global Register Manager Properties Page



C.5.3.2 Inserting a Configuration Object

There is no predefined Global register handle object available.

To configure the Global registers, use the drop-down menu to insert a new Global register configuration object by right-clicking on the DMA Global Register Manager (see Figure C–26) and selecting Insert dmaGblCfg. The configuration objects can be renamed.

Note:

The number of configuration objects is unlimited. Several configurations can be created and the user can select the right one for the four global registers and can change the configuration later just by selecting a new one under the DMA Global Registers Manager Properties page. Only one configuration on the list can be used for the four registers. Creating several pre–configuration objects provides more flexibility and reduces the time of modifying register values.

C.5.3.3 Deleting/Renaming a Configuration Object

To delete or rename an object, use the drop-down menu by right-clicking on the configuration object to be deleted or renamed.

If a configuration object is used by the preallocated objects of the DMA Global Register Manager, the Delete and Rename options are grayed-out and non-usable. The Show Dependency option is accessible and shows which configuration is used by the Global Register Manager. See section C.2, *Introduction to DSP/BIOS Configuration Tool: CSL Tree,* on page C-3.

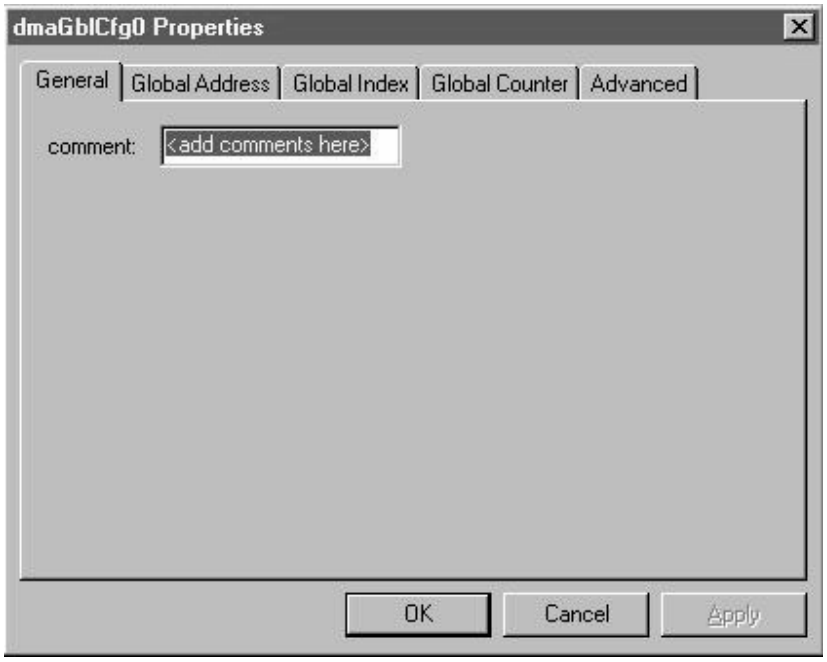
C.5.3.4 Properties Page of the Object

The Properties page allows you to set the memory-mapped registers related to the four Global registers. The configuration options are divided into the following tab pages:

- Global Address: Sets the Reload Addresses (registers A, B, C, and D)
- Global Index: Sets the Frame and Element Indexes (registers A and B)
- Global Counter: Sets the Frame and Element Counters (registers A and B)
- Advanced Page: Contains the full hexadecimal register values and reflects the setting of the previous tab pages. Also, the full register values can be entered directly and the new options will be mirrored on the related tab pages automatically.

Figure C–29 shows the DMA global properties page dialog box.

Figure C–29. DMA Global Properties Page



Each tab page is composed of several options that are set to a default value (at device power-on reset).

The options represent the fields of the DMA global registers. For further details on the fields and registers, refer to the *Direct Memory Access (DMA) Controller* chapter in the *TMS320C6000 Peripherals Reference Guide* (SPRU190).

C.5.4 C Code Generation for DMA Module

The two C files generated from the Configuration Tool are a header file and a source file.

C.5.4.1 Header File

The generated header file *cdbnamecfg.h* (see Example C–7) includes the csl header files required for the DMA module: csl.h, and csl_dma.h. The header file contains the declaration of DMA handles, global register IDs, and configuration objects.

Example C–7. DMA Header File

```
extern far DMA_Config dmaCfg0;
extern far DMA_GlobalConfig dmaGblCfg0;
extern far DMA_Handle hDma0;
extern far DMA_Handle hDma1;
extern far Uint32 dmaGblRegId;
```

C.5.4.2 Source File

The generated source file *cdbnamecfg_c.c (*see Example C–8) includes the declaration of the channel handle objects, the Global register ID variable, and the configuration structures.

Example C–8. DMA Source File (Declaration Section)

```
/*  Config Structures */
DMA_Config dmaCfg0 = {
    0x00000000,        /*  Primary Control Register  */
    0x00000080,        /*  Secondary Control Register  */
    0x00000000,        /*  Source Address Register  */
    0x00000000,        /*  Destination Address Register  */
    0x00000000         /*  Transfer Counter Register  */
};

DMA_GlobalConfig dmaGblCfg0 = {
    0x80000000,        /*  Global Address Reload Register A  */
    0x80001000,        /*  Global Address Reload Register B  */
    0x80002000,        /*  Global Address Reload Register C  */
    0x00000000,        /*  Global Address Reload Register D  */
    0x00000000,        /*  Global Index Register A  */
    0x00000000,        /*  Global Index Register B  */
    0x00000000,        /*  Global Count Reload Register A  */
    0x00000000         /*  Global Count Reload Register B  */
};

/*  Handles  */
DMA_Handle hDma1;
Uint32 dmaGblRegId;
```

The source file contains the channel handle object, Global register ID variable, and Configuration Pre-Initialization using, respectively, CSL DMA APIs `DMA_open()`, `DMA_globalAlloc()`, `DMA_config()`, and `DMA_globalConfig()`. These four functions are encapsulated in a unique function, `CSL_cfgInit()`. These functions will be generated if the associated check boxes are checked under the DMA Resource Manager and DMA Global Register Manager Properties pages.

Example C–9. DMA Source File (Body Section)

```
void CSL_cfgInit()
{
    dmaGblRegId = DMA_globalAlloc(0x15);
   DMA_globalConfig(dmaGblRegId, &dmaGblCfg0);
    hDma1 = DMA_open(DMA_CHA1, DMA_OPEN_RESET);
    DMA_config(hDma1, &dmaCfg0);
}
```

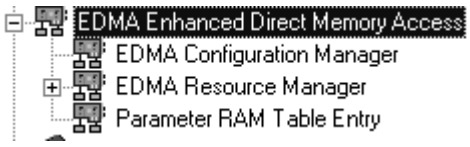
C.6 Configuring the EDMA Module Using CSL GUI

The EDMA module facilitates configuration of the Enhanced Direct Memory Access (EDMA) controller. The EDMA module consists of a configuration manager, a resource manager, and a parameter RAM table entry.

The configuration manager allows creation of an object that contains the complete set of register values needed to configure an EDMA channel. The resource manager associates a configuration object with a specific EDMA channel. The parameter RAM table entry allows creation handle objects associated with EDMA memory tables.

Figure C–30 illustrates the EDMA sections menu on the CSL graphical user interface (GUI).

Figure C–30. EDMA Sections Menu



The EDMA includes the following three sections:

- **EMIF Configuration Manager** allows you to create configuration objects by setting the memory-mapped registers related to the EDMA.
- **EDMA Resource Manager** allows you to select an EDMA channel and to associate a configuration object to this channel. The 16 channel handle objects are predefined.
- **Parameter RAM Table Entry** allows you to create handle objects associated with the EDMA memory tables.

C.6.1 EDMA Configuration Manager

The EDMA Configuration Manager allows you to create EDMA Channel configurations through the Properties page and to generate the configuration objects.

C.6.1.1 Inserting a Configuration Object

There is no predefined configuration object available.

To configure an EDMA channel through the memory-mapped registers, use the drop-down menu to insert a new configuration object by right-clicking on the EDMA Configuration Manager and selecting Insert edmaCfg. The configuration objects can be renamed. Their use depends on the on-chip device resources. Because 32 channels are available, a maximum of 32 configurations can be used simultaneously.

Note: The number of configuration objects is unlimited. Several configurations can be created and the user can select the right one for a specific port and can change the configuration later just by selecting a new one under the EDMA Resource Manager or under Parameter RAM Table Entry. This feature allows you more flexibility and reduces the time required to modify register values. The same configuration can be used by different channels.

C.6.1.2 Deleting/Renaming a Configuration Object

To delete or rename an object, use the drop-down menu by clicking on the configuration object to be deleted or renamed.

If a configuration object is used by one of the predefined handle objects of the EDMA Resource Manager or by one of the Table handle objects defined in the Parameter RAM Table Entry (see section C.6.3), the Delete and Rename options are grayed-out and are non-usable. The Show Dependency option is accessible and shows which device is using the configuration object. See section C.2, *Introduction to DSP/BIOS Configuration Tool: CSL Tree*, on page C-3.

C.6.1.3 Configuring the Object Properties

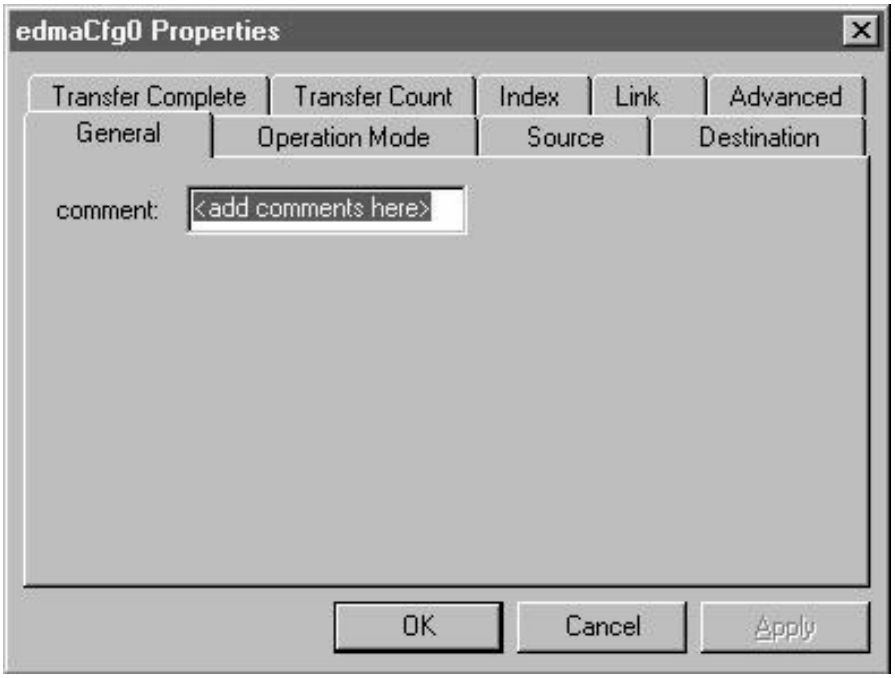
The Properties pages allow you to set the memory-mapped registers related to the EDMA. The configuration options are divided into the following tabpages:

- ❏ Operation Mode: Frame sync, Element size
- ❏ Source: Source transfer mode, Source Address
- ❏ Destination: Destination transfer mode, Destination Address
- ❏ Transfer Complete: Interrupt Mode
- ❏ Transfer Count: Frame and Element Counters setting
- ❏ Index: Frame and Element Indexes Setting

- Link: Link event to a specified memory table
- Advanced: Summary of the previous pages. This page contains the full hexadecimal register values and reflects the setting of the options done under the previous pages.
- The full register values can be entered directly and the new options will be mirrored on the corresponding pages automatically.

Figure C–31 shows the *EDMA Properties Page* dialog box.

Figure C–31. EDMA Properties Page



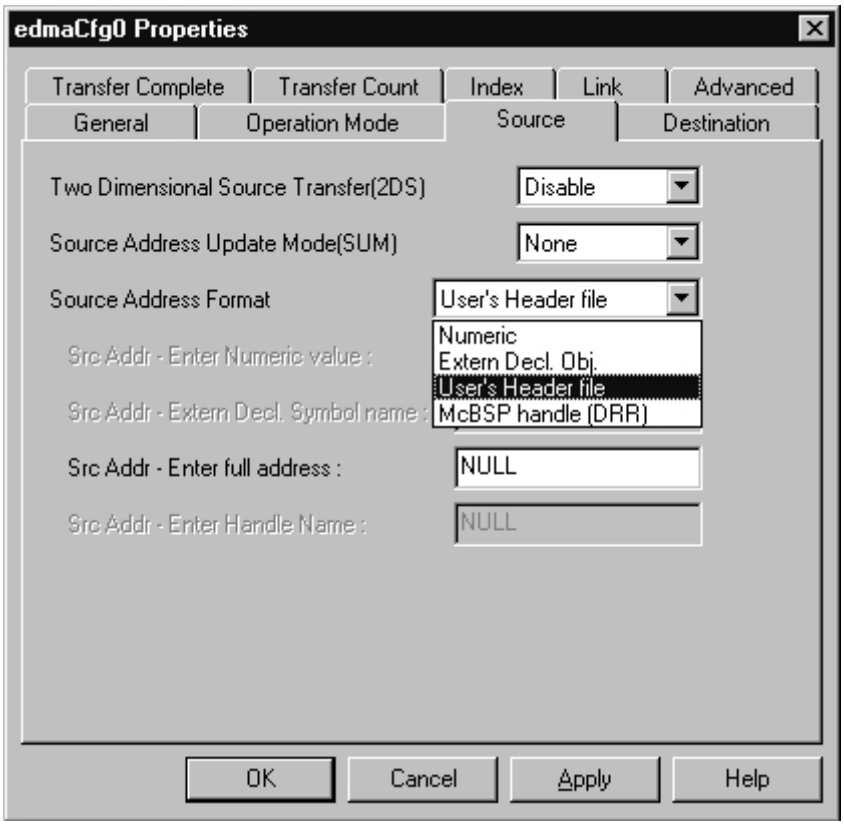
Each tabpage is composed of several options that are set to a default value (at device power-on reset).

The options represent the fields of the EDMA registers; the associated field name is shown in parenthesis.

C.6.1.4 Specifying Address Formats

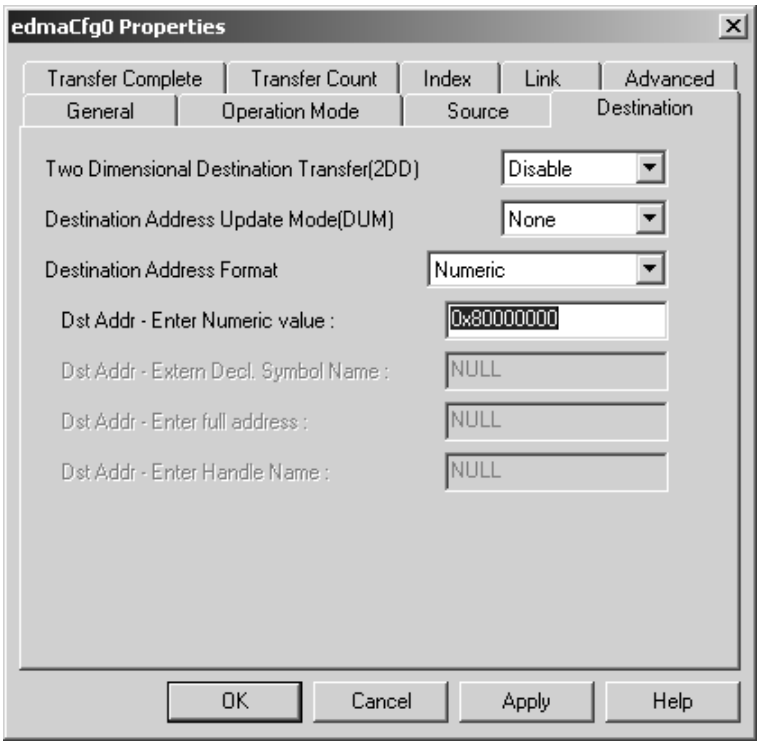
The source and destination address values can be set four different ways: numeric, extern declaration object (extern decl. obj.), user's header file, and McBSP handle (see Figure C–32).

Figure C–32. EDMA Configuration Dialog Showing Four Address-Format Types



- **Numeric:** Hexadecimal address value. This value is also reported on the EDMA configuration properties Advanced tab (see Figure C–33).

Figure C–33. Specifying an Address with a Numeric Type



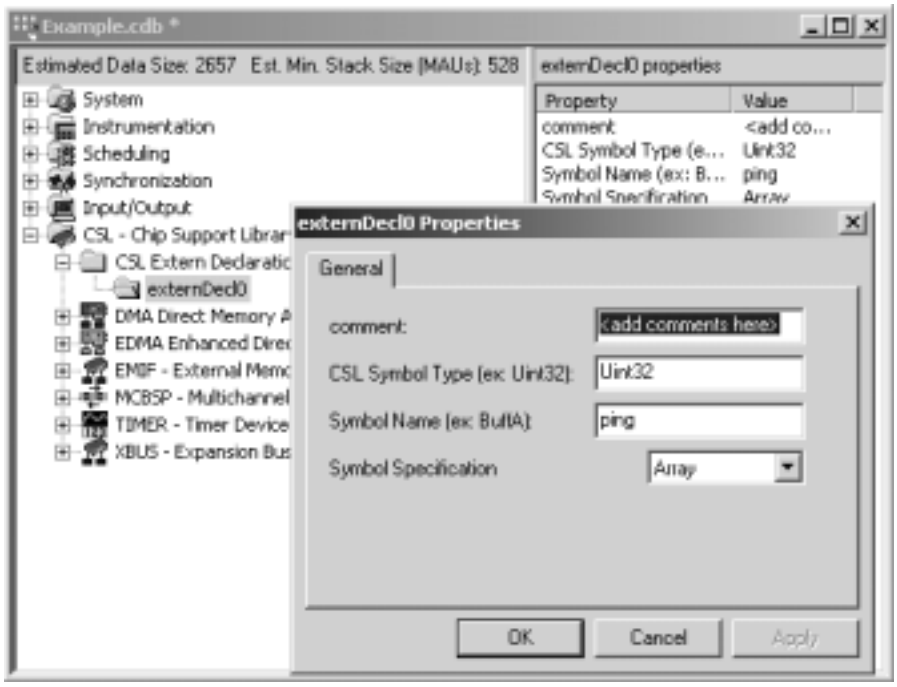
- **Extern Declaration Object (Extern Decl. Obj.):** A symbol previously defined as a CSL Extern Declaration may be used to represent an address.

Notes:

- If the extern symbol referenced in the Extern Declaration Object is not defined elsewhere in your project, an error will occur at compilation time.
- The name entered in the Address must exactly match the symbol name field defined in the Extern Declaration Object.

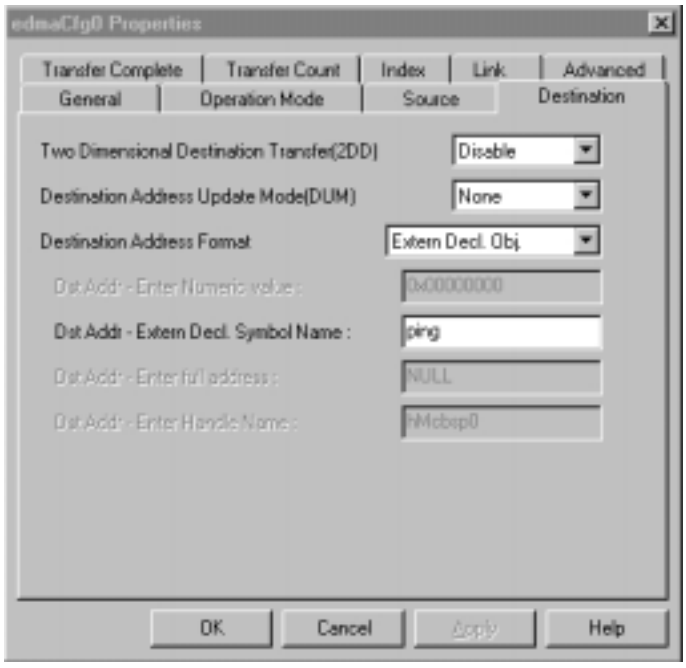
As shown in Figure C–34, **`ping`** is the symbol name of an array defined under CSL Extern Declaration.

Figure C–34. Creating an Extern Declaration Object



With `ping` defined as an extern, it may be used as a symbolic destination address (see Figure C–35).

Figure C–35. Specifying an Address Using an External Declaration Object



With `ping` defined as external and referenced as the destination address the DSP/BIOS Configuration Tool will automatically create the following EDMA Configuration object (edmaCfg0) in code:

```
extern far Uint32 ping[];
EDMA_Config edmaCfg0 = {
    0x40000002,      /*  Option  */
    0x00000000,      /*  Source Address – Numeric   */
    0x00000000,      /*  Transfer Counter – Numeric  */
    (Uint32) ping,   /*  Destination Address – Extern Decl. Obj  */
    0x00000004,      /*  Index register – Numeric  */
    0x00000000       /*  Element Count Reload and Link Address  */

};
```

- **User's Header file:** Symbols defined in a header file may be used as a source or destination address. When the *User's Header file* option is selected, the field "Enter full address" is made accessible.

Notes:

- ❑ The header file must first have been referenced under the CSL Extern Declaration.
- ❑ If the symbol is not defined, an error will occur at compilation time.
- ❑ If you referenced the header file in any other C file, you must be careful so as to prevent defining any global variables multiple times. #IFDEF can be used to prevent this from occurring.

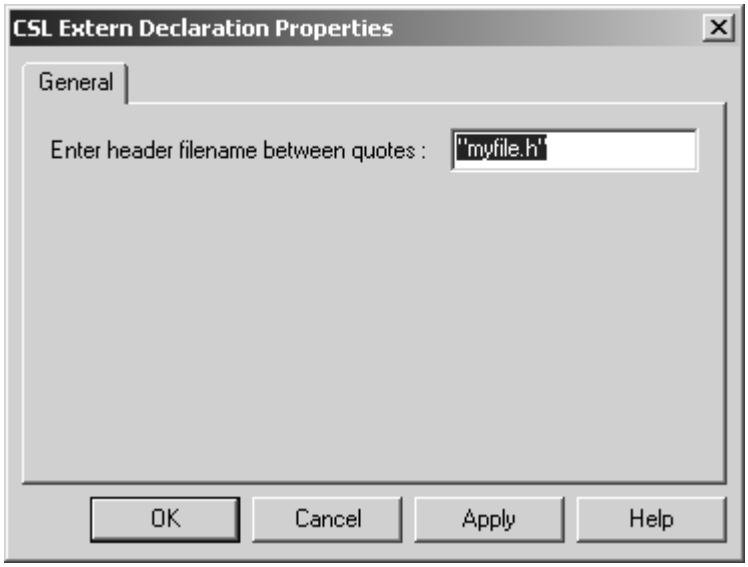
Example C–10 is an example user header file called *myfile.h*.

Example C–10. User's Header File for Source and Destination Address Setting

```
// myfile.h
#define     BUFFSIZE     256
#define     myESIZE      0x0004   //element size of 4 bytes
Uint32      myBuffer[BUFFSIZE];
```

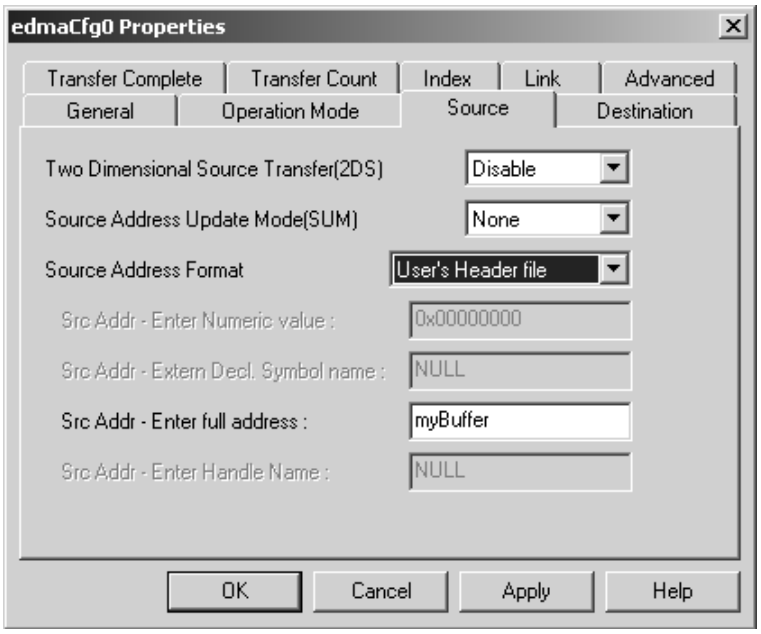
To allow the CSL graphical interface to reference it, right-click on the *CSL Extern Declaration* folder and choose *Properties*. Type the header file (along with quotes) into the dialog box as shown in Figure C–36.

Figure C–36. Referencing a User's Header File within the CSL GUI



With myfile.h declared, the symbols defined in the header file may now be used within CSL configurations. Figure C–37 uses myBuffer from as the EDMA source address.

Figure C–37. Specifying a Symbol from a User's Header File



Similar to the *Extern Declaration Object* the *User's Header file* symbol will also be set directly within EDMA configuration object:

```
#include "myfile.h"
extern far Uint32 ping[];

/* Config Structures */
EDMA_Config edmaCfg0 = {
 0x40000002,        /*  Option  */
(Uint32) myBuffer,  /*  Source Address – From User's Header File  */
 0x00000000,        /*  Transfer Counter – Numeric  */
(Uint32) ping,      /*  Destination Address – Extern Decl. Obj  */
 0x00000004,        /*  Index register – Numeric  */
 0x00000000         /*  Element Count Reload and Link Address  */
};
```

- **McBSP handle (DRR) or (DXR):** Allows the McBSP receive or transmit register to be set as either the source or destination address for the EDMA configuration.

  McBSP serial ports are commonly used as either a source or destination for EDMA transfers. For example, if an analog-to-digital converter (ADC) is connected to one of the McBSPs, the EDMA may be used to move the digital samples into on-chip memory for processing. In this case, the EDMA source address needs to be set to the McBSP receive register (DRR) address. The McBSP handle option has been provided to prevent the need to look up the numerical address from the peripherals reference

guide, and to provide greater portability in the event that another device is chosen in the CDB file's General Settings (i.e., improved portability).

To use the McBSP handle option, follow these steps:

1) Allocate the McBSP by right-clicking on one of the McBSP ports (found under the McBSP Resource Manager CSL tree) and choosing Properties. Click on the "Open McBSP Port" and define a handle name in the appropriate text box (or use the default handle name). Click OK to save the changes and close.
2) Open the EDMA configuration object to be associated with the McBSP.
3) Choose either the Source or Destination tab of the EDMA configuration, whichever is appropriate.
4) If configuring the source address, select "McBSP handle (DRR)" from the Source Address format field. Choose the similar "McBSP handle (DXR)" option if configuring the destination address.
5) When the McBSP handle format is chosen, the "Enter Handle Name" field becomes accessible. Enter the handle name chosen in the preceding step one.

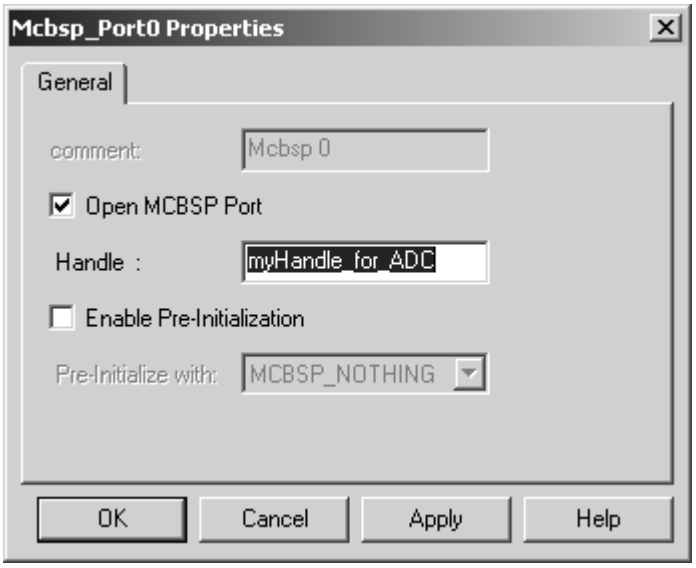
Note:

The handle name used in the EDMA "Enter Handle Name" field must match the McBSP name exactly or a compilation error will occur.

The following is an implementation of the ADC –> McBSP –> EDMA example from the above *McBSP handle (DRR)* description.

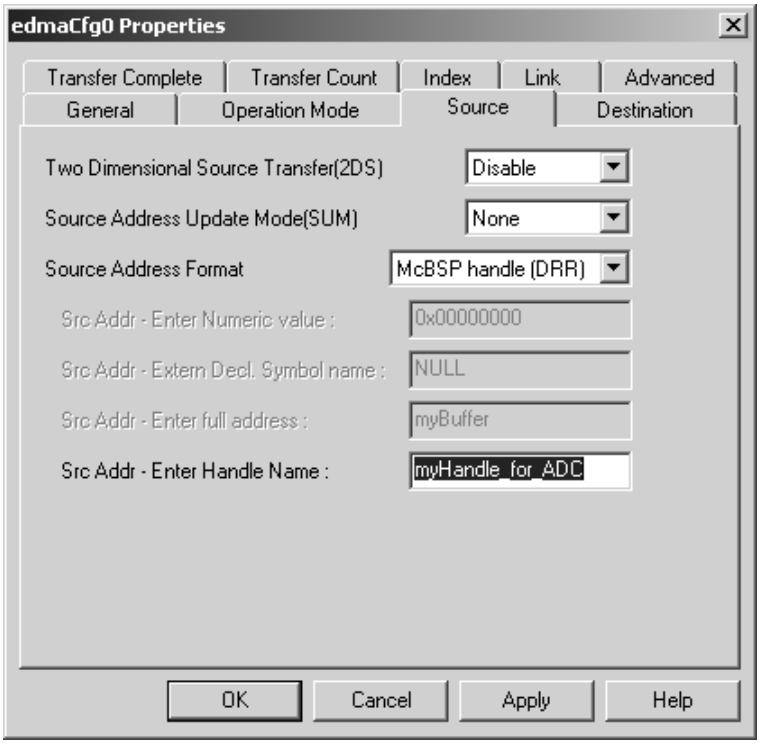
❑ Open McBSP port and define a handle (see Figure C–38).

Figure C–38. Requesting to Open a McBSP Port and Provide a Handle



- ❑ Open the EDMA configuration to the Source tab (see Figure C–39).
- ❑ Choose the McBSP handle type.
- ❑ Enter the handle name.

Figure C–39. Specifying an EDMA Source Address with a McBSP Handle



As with the previous examples, the Configuration Tool automatically configures the EDMA configuration object. In this case, though, the source (or destination) value is not directly specified in the EDMA configuration object, instead, it must be initialized in code. The Configuration Tool creates the `CSL_cfgInit()` function (as described in an earlier chapter) to provide this type of code. The `CSL_cfgInit()` function is automatically run before `main()` is called (see Example C–11).

Example C–11. Specifying an EDMA Source Address with a McBSP Handle

```
/*  Config Structures */
EDMA_Config edmaCfg0 = {
    0x40000002,        /*  Option  */
    0x00000000,        /*  Source Address – Numeric   */
    0x00000000,        /*  Transfer Counter – Numeric  */
    (Uint32) ping,     /*  Destination Address – Extern Decl. Obj  */
    0x00000004,        /*  Index register – Numeric  */
    0x00000000         /*  Element Count Reload and Link Address  */
};
/*  Handles  */
MCBSP_Handle myHandle_for_ADC;
void CSL_cfgInit()
{
myHandle_for_ADC = MCBSP_open(MCBSP_DEV0, MCBSP_OPEN_RESET);
edmaCfg0.src = EDMA_SRC_RMK(myHandle_for_ADC->drrAddr);
}
```

In this case, the “source” member value is set to a default numeric value (zero).

The “src” member value of edmaCfg0 is updated with McBSP DRR register address using CSL’s _RMK() macro.

C.6.1.5 Transfer Count and Index Setting

The counter and index register values can be set two different ways:

- **Numeric:** Hexadecimal address value. This value is also reported on the EDMA configuration properties’ Advanced tab.
- **User’s Header file:** Symbols defined in a header file may be used for setting the transfer count and/or Index fields. When the *User’s Header file* option is selected, the “Enter Num or Symbol value” field is made accessible.

Notes:

- The header file must first have been referenced under the CSL Extern Declaration.
- If the symbol is not defined, an error will occur at compilation time.
- If you referenced the header file in any other C file, you must be careful so as to prevent defining any global variables multiple times. #IFDEF can be used to prevent this from occurring.

Using symbols from header files can be very convenient. For example, a header file can be created to #define a symbol, such as BUFFSIZE in the following header file:

Example C–12. Transfer Count and Index Setting with User's Header File

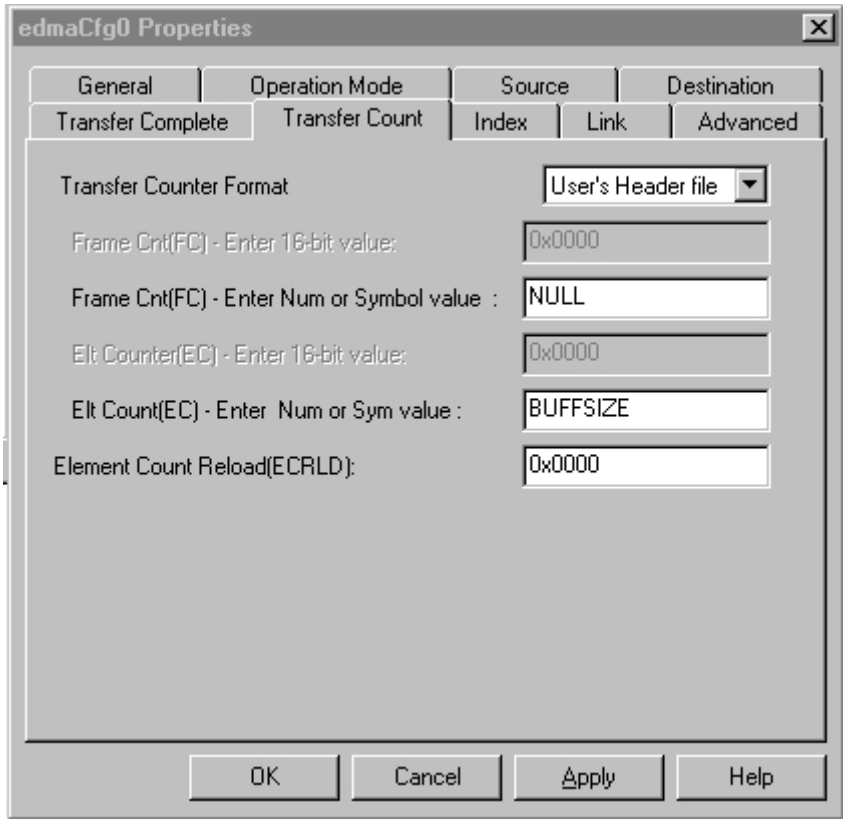
```
// myfile.h
#define     BUFFSIZE     256
#define     myESIZE      0x0004   //element size of 4 bytes
Uint32      myBuffer[BUFFSIZE];
```

(Note that this is the same header file as shown in the *User's Header File for Source and Destination Address Setting*, Example C–10).

If you added the appropriate #IFDEF's to myfile.h, it could be used by an entire project team to define project-wide symbols.

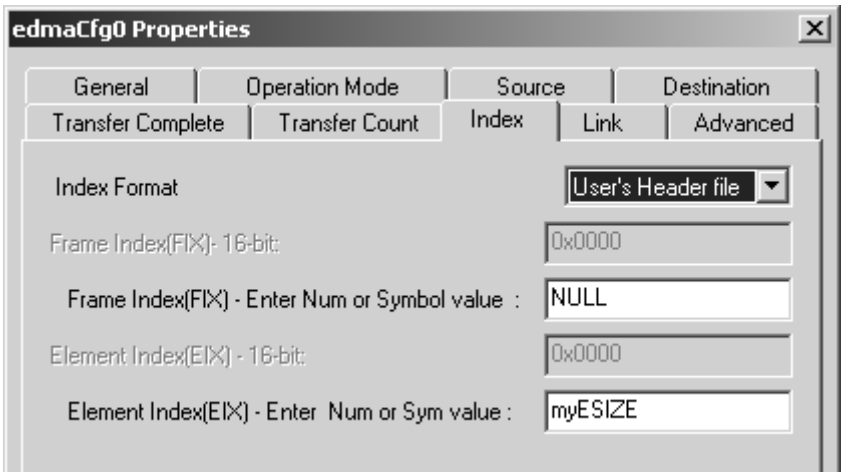
In this example, the BUFFSIZE value is used to set the EDMA configuration object's element count. In other words, this EDMA configuration will transfer BUFFSIZE number of elements.

Figure C–40. Using a Symbol from a User's Header File to Specify Element Transfer Count



Additionally, this example uses the #define myESIZE to set the element index, which describes how far should the source/destination address pointers move after each transfer.

Figure C–41. Using a Symbol from a User's Header File to Set Element Index



Transfer Count page/tab: FRAME counter (FC) is not used "NULL"; ELEMENT counter (EC) is set to BUFFSIZE.

Index page/tab: Element Index is set to myESIZE.

Because the element transfer count and index value do not fill–up an entire register, their value cannot be set directly in the EDMA configuration object. Instead, they are set in the `CSL_cfgInit()` function.

Example C–13. Using a Symbol from a User's Header File to Set Element Index

```
#include "myfile.h"
extern far Uint32 ping[];
/*  Config Structures */
EDMA_Config edmaCfg0 = {
    0x40000002,        /*  Option  */
    0x00000000,        /*  Source Address – Numeric   */
    0x00000000,        /*  Transfer Counter – Numeric  */
    (Uint32) ping,     /*  Destination Address – Extern Decl. Obj  */
    0x00000000,        /*  Index register – Numeric  */
    0x00000000         /*  Element Count Reload and Link Address  */
};
/*  Handles  */
MCBSP_Handle myHandle_for_ADC
void CSL_cfgInit()
{
    myHandle_for_ADC = MCBSP_open(MCBSP_DEV0, MCBSP_OPEN_RESET);
    edmaCfg0.cnt = EDMA_CNT_RMK(0,BUFFSIZE);
    edmaCfg0.idx = EDMA_IDX_RMK(0,myESIZE);
    edmaCfg0.src = EDMA_SRC_RMK(myHandle_for_ADC–>drrAddr);
}
```

The "counter" and "index" object members are set to a default numeric values.

The edmaCfg0 configuration object's "cnt" and "idx" values are updated using the _RMK() macros using the symbols defined in "myfile.h"

C.6.1.6 Link Address Setting

The reload/link register value can be set two different ways: table number and table handle.

- **Table Number:** Use a specific Reload/Link table number.
  - For 6x1x™ devices: Table number range goes from 0 to 66.

    Table number "0" is located at the address 0x01A01B0. Actually, there are 68 reload/link tables but the first two (located at 0x01A0180 and 0x01A0198, respectively) are reserved.
  - For 64x™ devices: Table number range goes from 0 to 18.

    Table number "0" is located at the address 0x01A00630. Again, there are actually 20 tables, but the first two (located at 0x01A0600 and 0x01A00618) are reserved (see `EDMA_allocTable()` function for further details).

Note:

CSL GUI does not support the C64x devices in Code Composer Studio IDE 2.0 yet.

Example C–14. Setting the Reload/Link Register Value Using Table Number "0"

```
/*  Config Structures */
EDMA_Config edmaCfg0 = {
    0x40000002,        /*  Option  */
    0x00000000,        /*  Source Address – Numeric   */
    0x00000000,        /*  Transfer Counter – Numeric  */
    (Uint32) ping,     /*  Destination Address – Extern Decl. Obj  */
    0x00000000,        /*  Index register – Numeric  */
    0x000001B0         /*  Element Count Reload and Link Address  */
};
```

- **Table Handle:** Rather than "hardcode" a specific table number, a *Table Handle name* may be specified from a drop-down list, as shown in Figure C–42.

Figure C–42. Setting the Reload/Relink Register Value by Selecting a Table Handle

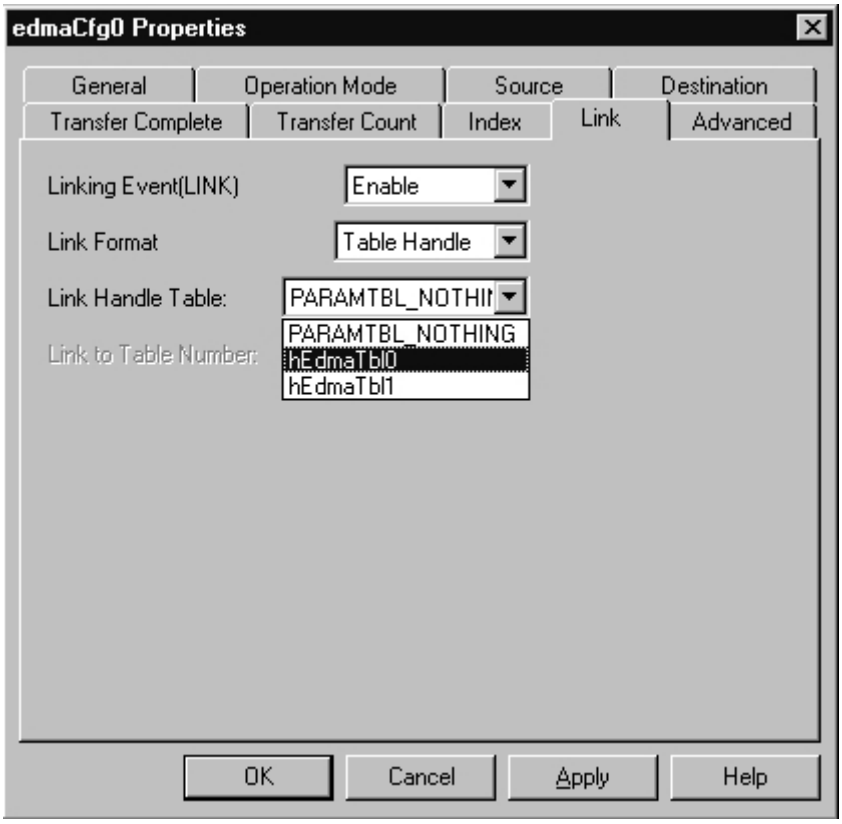
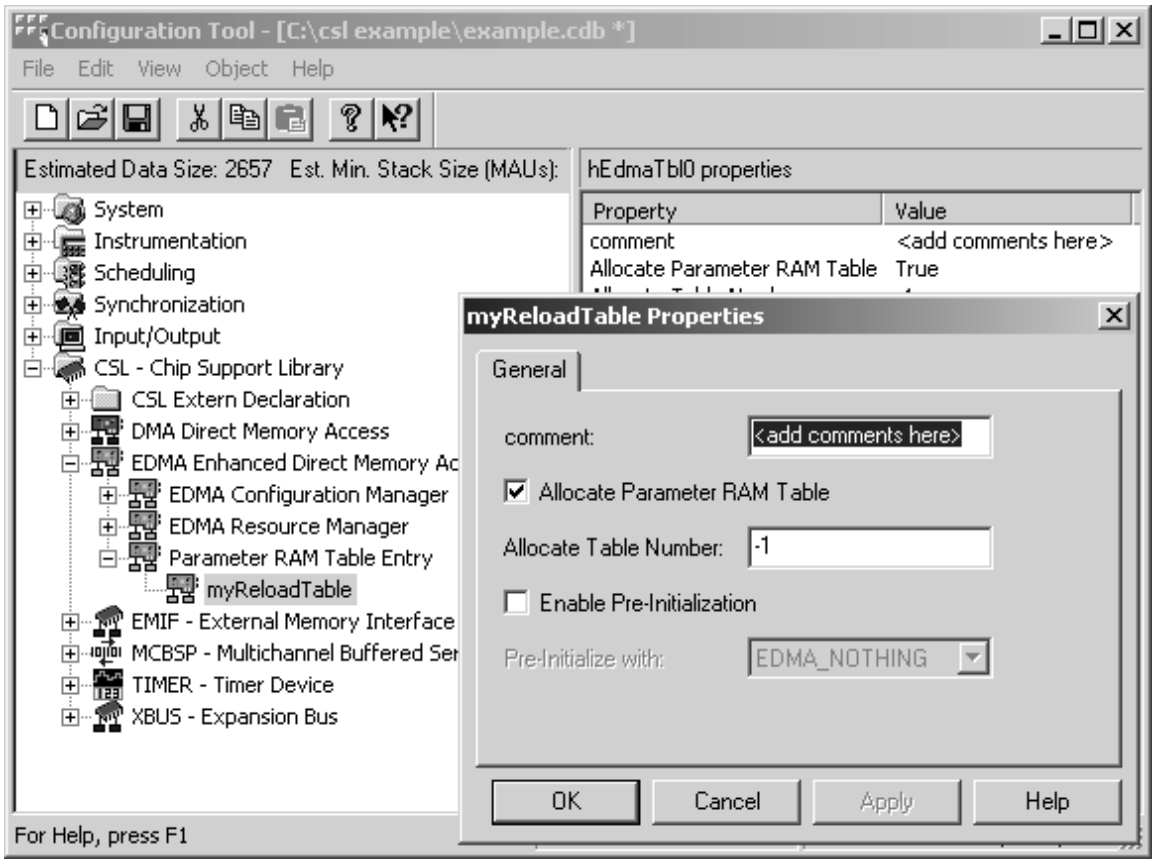


Table handles must be created before they can be used. They are created using the Parameter RAM Table Entry branch of the EDMA CSL tree. Figure C–43 an example of creating a reload/link table.

Figure C–43. Creating a Reload/Link Table



The “–1” option tells the CSL graphical interface that any available table location may be used. This allows maximum flexibility and when assigning a Parameter RAM table to an EDMA configuration object.

Example C–15. Setting the Reload/Link Register Value Using Table Handle

```
/*  Config Structures */
EDMA_Config edmaCfg0 = {
    0x40000002,      /* Option  */
    0x00000000,      /* Source Address – Numeric   */
    0x00000000,      /* Transfer Counter – Numeric  */
    (Uint32) ping,   /* Destination Address – Extern Decl. Obj  */
    0x00000000,      /* Index register – Numeric  */
    0x00000000       /* Element Count Reload and Link Address  */
};
/*  Handles  */
EDMA_Handle myReloadTable;

/*
 *  ======== CSL_cfgInit() ========
 */
void CSL_cfgInit()
{
myReloadTable = EDMA_allocTable(–1);

edmaCfg0.rld =(edmaCfg0.rld & 0xFFFF0000)|(EDMA_RLD_RMK(0, myReloadTable));

}
```

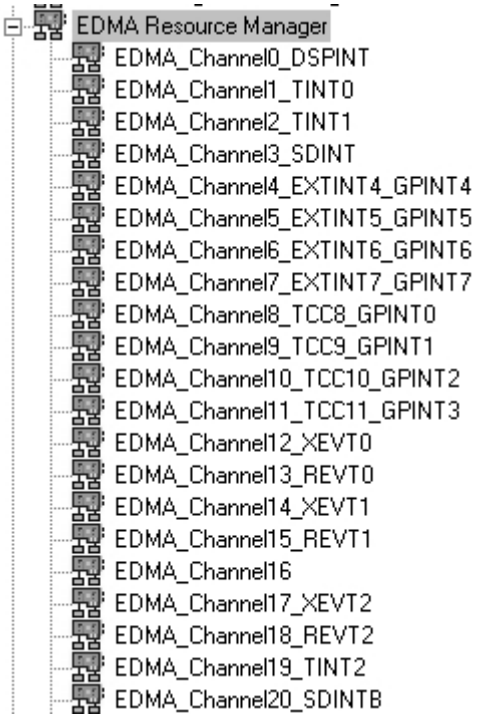
This code line is generated when Allocate Parameter Table is checked under Parameter RAM Table Entry.

This code line updates the lower 16-bits of the “rld” member of the edmaCfg0 EDMA configuration object. (The upper 16-bits specify the Element Count reload, which happen to be zero in this example. The lower 16-bits specify the link address.)

C.6.2 EDMA Resource Manager

The EDMA Resource Manager allows you to generate the `EDMA_open()`, `EDMA_config()`, and `EDMA_enableChannel()` CSL functions.

Figure C–44. EDMA Resource Manager Menu



C.6.2.1 Predefined Handles

The EDMA channels handle objects are predefined and each of them with a supported on-chip EDMA channels: 16 channels for 621x/671x and 64 channels for 641x devices.

The default handle name is pre-entered with **hEdmaCha**#number# format. The user may change the handle name if desired; for example, hEdmaCha1 can be renamed hEdmaTint0.

Note:

The above objects cannot be deleted. They can be renamed only. A configuration can be enabled if at least one configuration object was defined previously in section C.6.1.

C.6.2.2 Properties Page

The Properties page is accessible by right-clicking on the drop-down menu option Properties.

The first time the Properties page appears, only the Open EDMA Channel check-box can be selected.

EDMA_NOTHING is used to indicate that there is no configuration object selected for this channel handle.

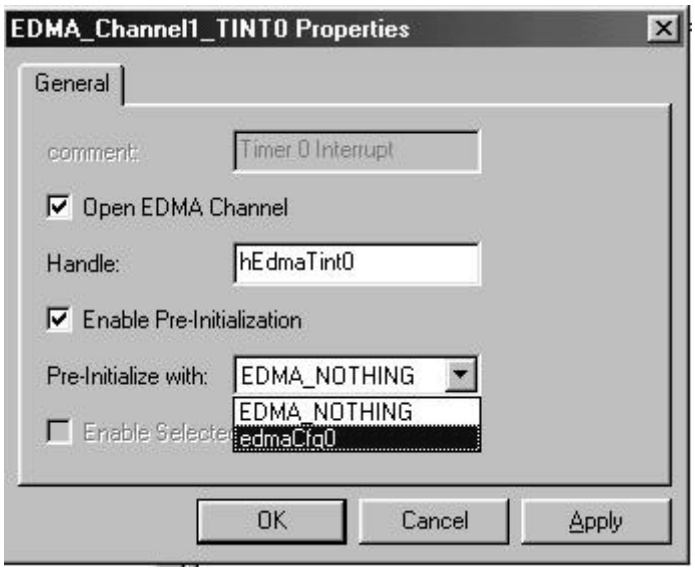
To enable Enable Pre-Initialization, create a new configuration object first (see section C.6.1). When a configuration object is available, Enable Pre-Initialization can be checked and you can select a configuration object on the drop-down list. Also, the user can enable the channel by checking the EDMA Selected Channel check box.

Channels 8–11 cannot be enabled; they are reserved for chaining.

If EDMA_NOTHING remains selected, no configuration object will be generated for the related EDMA handle, and the Enable Selected Channel is not accessible (see section C.6.4).

In Figure C–45 the Open EDMA Channel option is checked and the handle object hEdmaTint0 is now accessible (renaming allowed). The `EDMA_open()` function will be generated with hEdmaTint0 containing the returned handle address.

Figure C–45. EDMA Properties Page With Handle Object Accessible



C.6.3 Parameter RAM Table Entry

The Parameter RAM Table Entry allows you to create a table handle object and generate the `EDMA_allocTable()` and `EDMA_config()` CSL functions.

C.6.3.1 Inserting a Configuration Object

There is no predefined Table handle object. To insert an object, use the drop-down menu to insert a new Table handle object by clicking on the Parameter RAM Table Entry and selecting Insert hEdmaTbl. The Table handle objects can be renamed. Their use depends on the on-chip device resources.

Note:

The number of configuration objects is unlimited. Several Table objects can be created and the user can use the ones that are desired.

C.6.3.2 Deleting/Renaming a Configuration Object

To delete or rename an object, use the drop-down menu by clicking on the handle object and selecting the Delete or Rename option.

C.6.3.3 Properties Page

The Properties page is accessible by right-clicking on the drop-down menu option Properties.

The first time the Properties page appears, only the Allocate Parameter RAM Table check-box can be selected. The default Allocate Table Number is “0” and EDMA_NOTHING is used to indicate that there is no configuration object associated with the Table handle object.

To enable Enable Pre-Initialization, create a new configuration object first (see section C.6.1). When a configuration object is available, Enable Pre-Initialization can be checked and you can select a configuration object from the drop-down list.

For 6x1x devices, 67 Reload/Link tables are available: Table Number [0–66]

For 64x devices, 19 Reload/Link tables are available: Table Number [0–18]

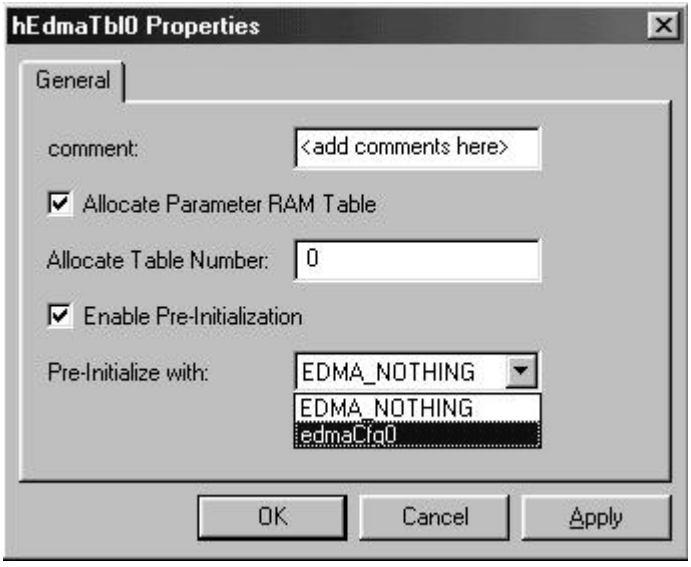
Note:

The table is allocated to the first available PRAM table by default with“–1” table number set and the Allocate Parameter RAM table checked.

If EDMA_NOTHING remains selected, no configuration object will be generated for the related Table handle.

Figure C–46 is the EDMA table Properties page showing the allocation of table number 0 and parameters setting with edmCfg0 configuration structure.

Figure C–46. EDMA Table Properties Page



C.6.4 C Code Generation for EDMA Module

The two C files generated from the configuration tool are a header file and a source file.

C.6.4.1 Header File

The generated header file *cdbnamecfg.h* includes the csl header files required for the EDMA module: csl.h, and csl_edma.h. It contains the declaration of EDMA handles, EDMA table handles, and configuration objects.

Example C–16. EDMA Header File

```
extern far EDMA_Config edmaCfg0;
extern far EDMA_Handle hEdmaTint0;
extern far EDMA_Handle hEdmaTbl0;
```

C.6.4.2 Source File

The source file includes the declaration of the channel and Table handle objects and the configuration structures.

Example C–17. EDMA Source File (Declaration Section)

```
/*  Config Structures */
EDMA_Config edmaCfg0 = {
    0x40000000,        /*  Option  */
    0x00000000,        /*  Source Address  */
    0x00000001,        /*  Transfer Counter  */
    0x00000000,        /*  Destination Address  */
    0x00000000,        /*  Transfer Index  */
    0x00000180       /*  Element Count Reload and Link Address  */
};

/*  Handles  */
EDMA_Handle hEdmaTint0;
EDMA_Handle hEdmaSdint;
EDMA_Handle hEdmaTbl0;
```

The source file contains the Channel handle object , Table handle object, and Configuration Pre-Initialization using CSL EDMA APIs `EDMA_open()`, `EDMA_allocTable()`, `EDMA_config()`, and `EDMA_enableChannel()`. These four functions are encapsulated in a unique function, `CSL_cfgInit()`. These functions will be generated if the associated check boxes are checked under the EDMA Resource Manager and Parameter RAM Table Entry Properties pages.

Example C–18. EDMA Source File (Body Section)

```
void CSL_cfgInit()
{
     hEdmaTint0 = EDMA_open(EDMA_CHA_TINT0, EDMA_OPEN_RESET);
     hEdmaTbl0 = EDMA_allocTable( 0);
     EDMA_config(hEdmaTint0, &edmaCfg0);
     EDMA_enableChannel(hEdmaTint0);
     EDMA_config(hEdmaTbl0, &edmaCfg0);
}
```

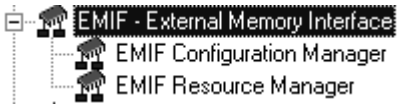
C.7 Configuring the EMIF Module Using CSL GUI

The EMIF module consists of a configuration manager and a resource manager.

The configuration manager allows creation of configuration objects by setting the memory-mapped registers related to the EMIF. The resource manager associates a pre-configuration object to the EMIF.

Figure C–47 illustrates the EMIF sections menu on the CSL graphical user interface (GUI).

Figure C–47. EMIF Sections Menu



The EMIF includes the following two sections:

- **EMIFA Configuration Manager** allows you to create configuration objects by setting the memory-mapped registers related to the EMIF.
- **EMIFA Resource Manager** allows you to associate a pre-configuration object to the EMIF.

C.7.1 EMIF Configuration Manager

The EMIF Configuration Manager allows you to create EMIF configurations through the Properties page and generate the configuration objects.

C.7.1.1 Inserting a Configuration Object

There is no predefined configuration object. To insert a new configuration object, access the drop-down menu, click on the EMIF Configuration Manager, and select Insert emifCfg. The configuration objects can be renamed. Their use depends on the on-chip device resources. Because there is only one EMIF, only one configuration will be used.

Note:

The number of configuration objects is unlimited. Several configurations can be created and the user can select one for EMIF and can change the configuration later just by selecting another configuration under the EMIF Resource Manager. The goal is to provide more flexibility and to reduce the time required to modify register values.

C.7.1.2 Deleting/Renaming a Configuration Object

To delete or rename an object, use the drop-down menu by clicking on the configuration object to be deleted or renamed.

If a configuration object is selected in the EMIF Resource Manager (see section C.7.2), the Delete and Rename options are grayed-out and non-usable for this object. The Show Dependency option is accessible and shows that the EMIF resource manager is using the configuration object. See section C.2, *Introduction to DSP/BIOS Configuration Tool: CSL Tree*, on page C-3.

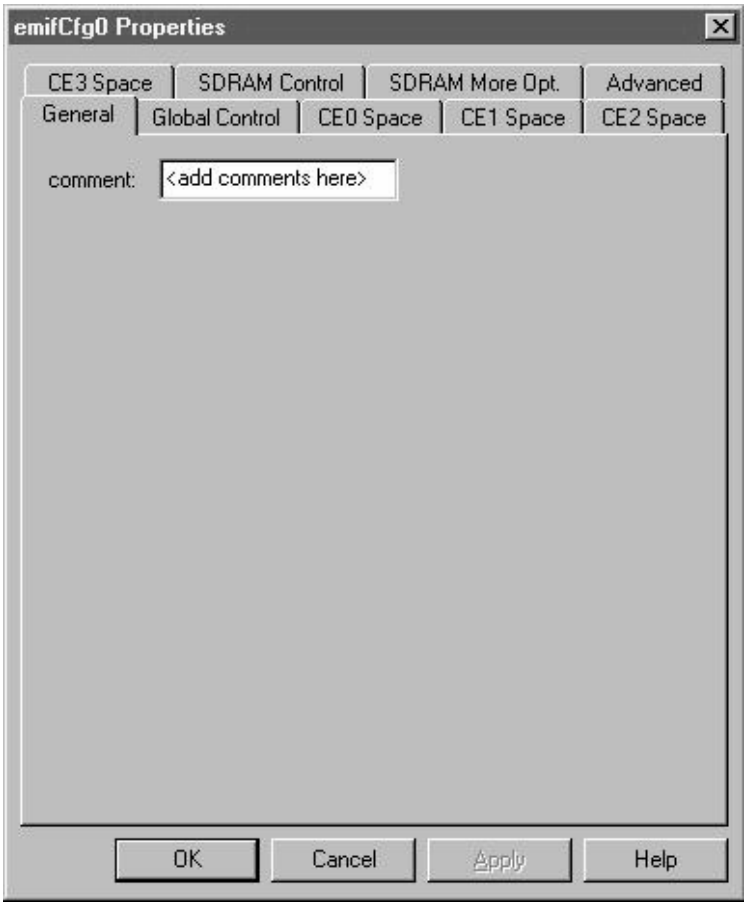
C.7.1.3 Properties Page of the Object

The Properties pages allow you to set the memory-mapped registers related to the EMIF. The configuration options are divided into the following tab pages:

- Global Control: Hold, clock enable
- CE0 Space: configuration of CE0 memory space (CECTL0 register)
- CE1 Space: configuration of CE1 memory space (CECTL1 register)
- CE2 Space: configuration of CE2 memory space (CECTL2 register)
- CE3 Space: configuration of CE3 memory space (CECTL3 register)
- SDRAM Control: Allows you to set SDRAM control functions.
- SDRAM More: Allows you to extend the SDRAM control setting (SDEXT register)
- Advanced Page: This page contains the full hexadecimal register values and reflects the setting of the previous tab pages. Also, the full register values can be entered directly and the new options will be mirrored on the related tab pages automatically.

Figure C–48 shows the EMIF Properties page dialog box.

Figure C–48. EMIF Properties Page



Each tab page is composed of several options that are set to a default value (at device reset).

The options represent the fields of the EMIF registers; the associated field name is shown in parenthesis. For further details on the fields and registers, refer to the *External Memory Interface* chapter in the *TMS320C6000 Peripherals Reference Guide* (SPRU190).

C.7.2 EMIF Resource Manager

The EMIF Resource Manager allows you to generate the EMIF_config CSL function with the predefined configuration as parameter. Because only one EMIF is supported, only one resource is available and used as the default.

C.7.2.1 Properties Page

The Properties page is accessible by right-clicking on the drop-down menu option Properties.

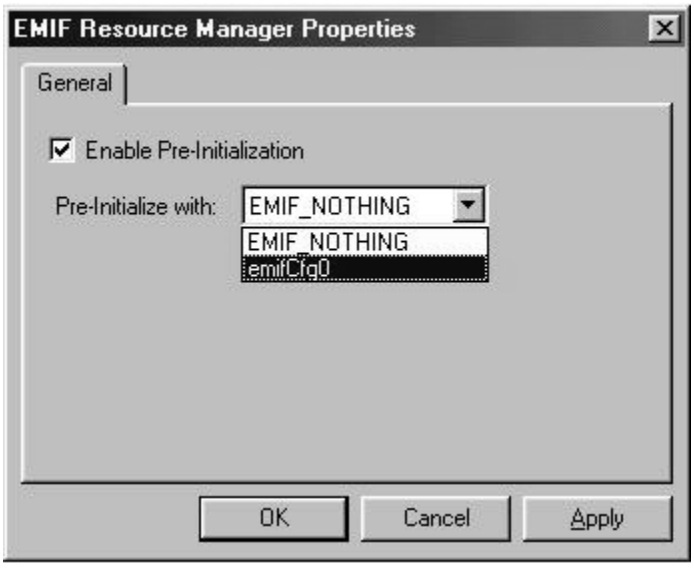
The first time the Properties page appears, EMIF_NOTHING is used to indicate that there is no configuration object selected for this peripheral.

To enable Enable Pre-Initialization, create a new configuration object first (see section C.7.1). When a configuration object is available, Enable Pre-Initialization can be checked and you can select a configuration object from the drop-down list.

If EMIF_NOTHING remains selected, no configuration object will be generated.

In the example shown in Figure C–49 the Enable Pre-Initialization option is checked, `EMIF_config()` will be generated with emifCfg0 as the parameter (see section C.7.3).

Figure C–49. EMIF Resource Manager Dialog Box



C.7.3 C Code Generation for EMIF Module

The two C files generated from the configuration tool are a header file and a source file.

C.7.3.1 Header File

The generated header file *cdbnamecfg.h* includes the csl header files required for the EMIF module: csl.h, and csl_emif.h. It contains the declaration of the EMIF configuration objects.

Example C–19. EMIF Header File

```
extern far EMIF_Config emifCfg0;
```

C.7.3.2 Source File

The source file includes the declaration of the configuration structures (values of the memory-mapped registers).

Example C–20. EMIF Source File (Declaration Section)

```
/*  Config Structures */
EMIF_Config emifCfg0 = {
    0x00003078,  /*  Global Control Reg. (GBLCTL)   */
    0xFFFF3F23,  /*  CE0 Space Control Reg. (CECTL0)   */
    0xFFFF3F23,  /*  CE1 Space Control Reg. (CECTL1)   */
    0xFFFF3F23,  /*  CE2 Space Control Reg. (CECTL2)   */
    0xFFFF3F23,  /*  CE3 Space Control Reg. (CECTL3)   */
    0x0388F000,  /*  SDRAM Control Reg.(SDCTL)   */
    0x00000080,  /*  SDRAM Timing Reg.(SDTIM)   */
    0x00000000   /* (6211/6711 only) SDRAM Extended
Reg.(SDEXT)*/
};
```

The source file contains the Configuration Pre-Initialization EMIF API, `EMIF_config()`. This function is encapsulated in a unique function, `CSL_cfgInit()`.

`EMIF_config()` will be generated only if Enable-Pre-Initialization is checked under the EMIF Resource Manager Properties page (with a selected configuration other than EMIF_NOTHING).

Example C–21. EMIF Source File (Body Section)

```
void CSL_cfgInit()
(

    EMIF_config(&emifCfg0);

}
```

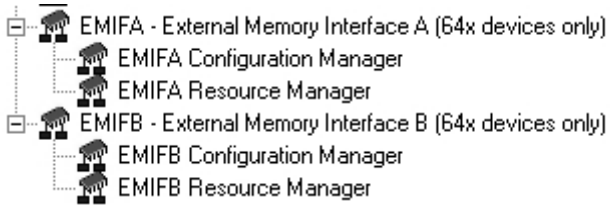
C.8 Configuring the EMIFA/EMIFB Modules Using CSL GUI

The EMIFA/EMIFB modules consist of a configuration manager and a resource manager.

The configuration manager allows creation of configuration objects by setting the memory-mapped registers related to the EMIFA/B. The resource manager associates a pre-configuration object to the EMIFA/B.

Figure C–47 illustrates the EMIF sections menu on the CSL graphical user interface (GUI).

Figure C–50. EMIFA(B) Sections Menu



The EMIFA(B) includes the following two sections:

- **EMIFA(B) Configuration Manager** allows you to create configuration objects by setting the memory-mapped registers related to the EMIFA/B.
- **EMIFA(B) Resource Manager** allows you to associate a preconfiguration object to the EMIFA and/or EMIFB.

C.8.1 EMIFA(B) Configuration Manager

The EMIFA(B) Configuration Manager allows you to create EMIFA/EMIFB configurations through the Properties page and generate the configuration objects.

C.8.1.1 Inserting a Configuration Object

There is no predefined configuration object. To insert a new configuration object, access the drop-down menu, click on the EMIFA(B) Configuration Manager, and select Insert emifCfg. The configuration objects can be renamed. Their use depends on the on-chip device resources. Because there is only one EMIFA and one EMIFB, only one configuration will be used for each one.

Note:

The number of configuration objects is unlimited. Several configurations can be created and the user can select one for EMIFA(B) and can change the configuration later just by selecting another configuration under the EMIFA(B) Resource Manager. The goal is to provide more flexibility and to reduce the time required to modify register values.

C.8.1.2 Deleting/Renaming a Configuration Object

To delete or rename an object, use the drop-down menu by clicking on the configuration object to be deleted or renamed.

If a configuration object is selected in the EMIFA(B) Resource Manager (see section C.8.2), the Delete and Rename options are grayed-out and non-usable for this object. The Show Dependency option is accessible and shows that the EMIFA(B) resource manager is using the configuration object. See section C.2, *Introduction to DSP/BIOS Configuration Tool: CSL Tree*, on page C-3.

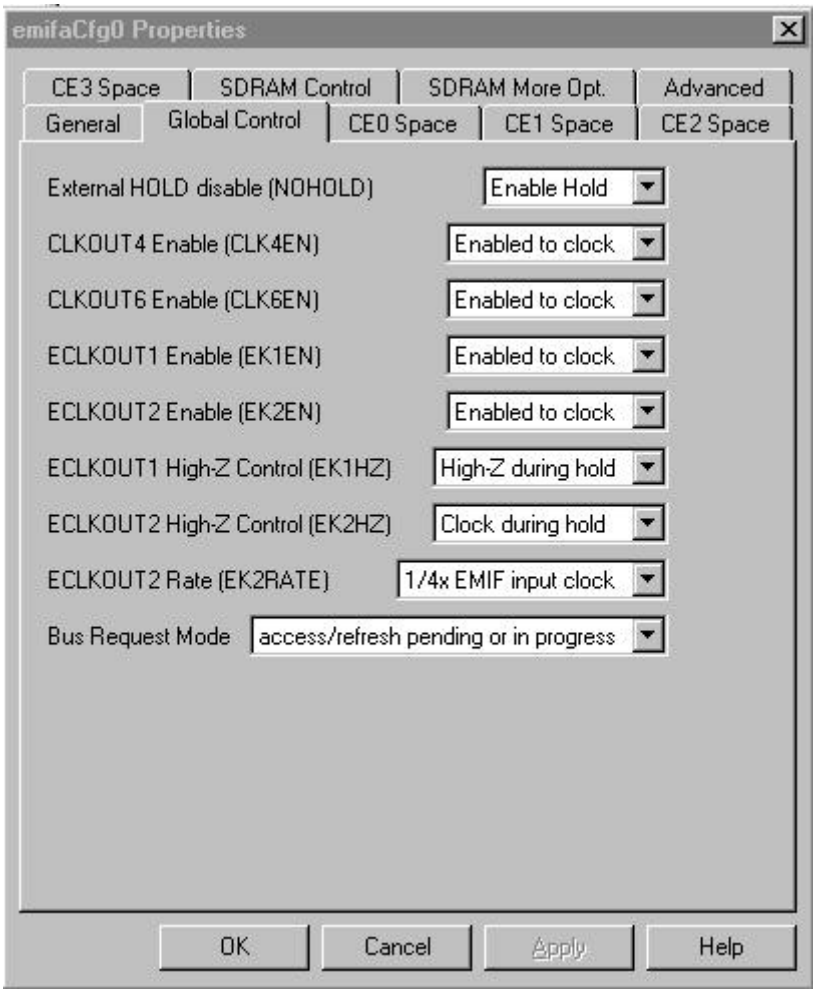
C.8.1.3 Properties Page of the Object

The Properties pages allow you to set the memory-mapped registers related to the EMIFA/B. The configuration options are divided into the following Tab pages:

- Global Control: Hold, clock enable
- CE0 Space: configuration of CE0 memory space (CE0CTL/CESECO registers)
- CE1 Space: configuration of CE1 memory space (CE1CTL/CESEC1 registers)
- CE2 Space: configuration of CE2 memory space (CE2CTL/CESEC2 registers)
- CE3 Space: configuration of CE3 memory space (CE3CTL/CESEC3 registers)
- SDRAM Control: Allows you to set SDRAM control functions.
- SDRAM More: Allows you to extend the SDRAM control setting (SDEXT register)
- Advanced Page: This page contains the full hexadecimal register values and reflects the setting of the previous Tab pages. Also, the full register values can be entered directly and the new options will be mirrored on the related Tab pages automatically.

Figure C–51 shows the EMIFA(B) Properties page dialog box.

Figure C–51. EMIFA(B) Properties Page



Each Tab page is composed of several options that are set to a default value (at device reset).

The options represent the fields of the EMIFA(B) registers; the associated field name is shown in parenthesis. For further details on the fields and registers, refer to the *External Memory Interface* chapter in the *TMS320C6000 Peripherals Reference Guide* (SPRU190).

C.8.2 EMIFA(B) Resource Manager

The EMIFA(B) Resource Manager allows you to generate the EMIFA_config() CSL function with the predefined configuration as parameter. Because only one EMIFA and one EMIFB is supported, only one resource is available and used as the default for each.

C.8.2.1 Properties Page

The Properties page is accessible by right-clicking on the drop-down menu option properties.

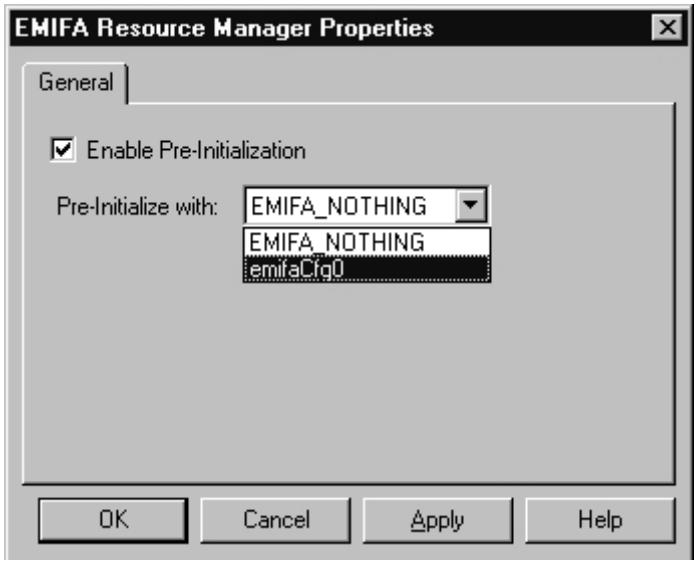
The first time the Properties page appears, EMIFA(B)_NOTHING is used to indicate that there is no configuration object selected for this peripheral.

To enable Enable Pre-Initialization, create a new configuration object first (see section C.8.1). When a configuration object is available, Enable Pre-Initialization can be checked and you can select a configuration object from the drop-down list.

If EMIFA(B)_NOTHING remains selected, no configuration object will be generated.

In the example shown in Figure C–52, the Enable Pre-Initialization option is checked, EMIFA_config() will be generated with emifCfg0 as the parameter (see section C.8.3).

Figure C–52. EMIFA(B) Resource Manager Dialog Box



C.8.3 C-Code Generation for EMIFA(B) Module

The two C files generated from the configuration tool are a header file and a source file.

C.8.3.1 Header File

Example C–22. EMIFA Header File

```
extern far EMIFA_Config emifaCfg0;
```

C.8.3.2 Source File

The source file includes the declaration of the configuration structures (values of the memory-mapped registers).

Example C–23. EMIFA Source File (Declaration Section)

```
/*  Config Structures */
EMIFA_Config emifaCfg0 = {
    0x0009207c,  /*  Global Control Reg. (GBLCTL)   */
    0xFFFFFF23,  /*  CE0 Space Control Reg. (CECTL0)   */
    0xFFFFFF23,  /*  CE1 Space Control Reg. (CECTL1)   */
    0xFFFFFF23,  /*  CE2 Space Control Reg. (CECTL2)   */
    0xFFFFFF23,  /*  CE3 Space Control Reg. (CECTL3)   */
    0x0348F000,  /*  SDRAM Control Reg.(SDCTL)   */
    0x005DC5DC,  /*  SDRAM Timing Reg.(SDTIM)   */
    0x00175F3F,  /*  SDRAM Extended Reg.(SDEXT)   */
    0x00000002,  /*  CE0 Space Secondary Control Reg. (CSEC0)   */
    0x00000002,  /*  CE1 Space Secondary Control Reg. (CSEC1)   */
    0x00000002,  /*  CE2 Space Secondary Control Reg. (CSEC2)   */
    0x00000002   /*  CE3 Space Secondary Control Reg. (CSEC3)   */
};
```

The source file contains the Configuration Pre-Initialization EMIFA API, EMIFA_config(). This function is encapsulated in a unique function, CSL_cfgInit().

EMIFA_config() will be generated only if Enable-Pre-Initialization is checked under the EMIFA(B) Resource Manager Properties page (with a selected configuration other than EMIFA_NOTHING or EMIFB_NOTHING, respectively).

Example C–24. EMIFA Source File (Body Section)

```
void CSL_cfgInit()
(

    EMIFA_config(&emifaCfg0);

}
```

C.9 Configuring the McBSP Module Using CSL GUI

The McBSP module consists of a configuration manager and a resource manager.

The configuration manager allows creation of configuration objects. The resource manager associates a configuration object to the device.

Figure C–53 illustrates the McBSP sections menu on the CSL graphical user interface (GUI).

Figure C–53. McBSP Sections Menu

The McBSP includes the following two sections:

- **McBSP Configuration Manager** allows you to create configuration objects. No predefined configuration objects.
- **McBSP Resource Manager** allows you to select a device and to associate a configuration object to that device. Three handle objects are predefined.

C.9.1 McBSP Configuration Manager

The McBSP Configuration Manager allows you to create device configurations through the Properties page and to generate the configuration objects.

C.9.1.1 Inserting a Configuration Object

There is no predefined configuration object available.

To configure a McBSP port through the memory-mapped registers, use the drop-down menu to insert a new configuration object by right-clicking on the McBSP Configuration Manager and selecting Insert mcbspCfg. The configuration objects can be renamed. Their use depends on the on-chip device resources. This means that if only two McBSPs are supported, a maximum of two configurations can be used simultaneously.

Note:

The number of configuration objects is unlimited. Several configurations can be created and you can select the right one for a specific port and can change the configuration later just by selecting a new one under the McBSP Resource Manager. The goal is to provide more flexibility and to reduce the time required to modify register values.

C.9.1.2 Deleting/Renaming a Configuration Object

To delete or rename an object, use the drop-down menu by clicking on the configuration object to be deleted or renamed.

If a configuration object is used by one of the predefined handle objects of the McBSP Resource Manager (see section C.9.2), the Delete and Rename options are grayed-out and are non-usable. The Show Dependency option is accessible and shows which device is using the configuration object. See section C.2, *Introduction to DSP/BIOS Configuration Tool: CSL Tree*, on page C-3.

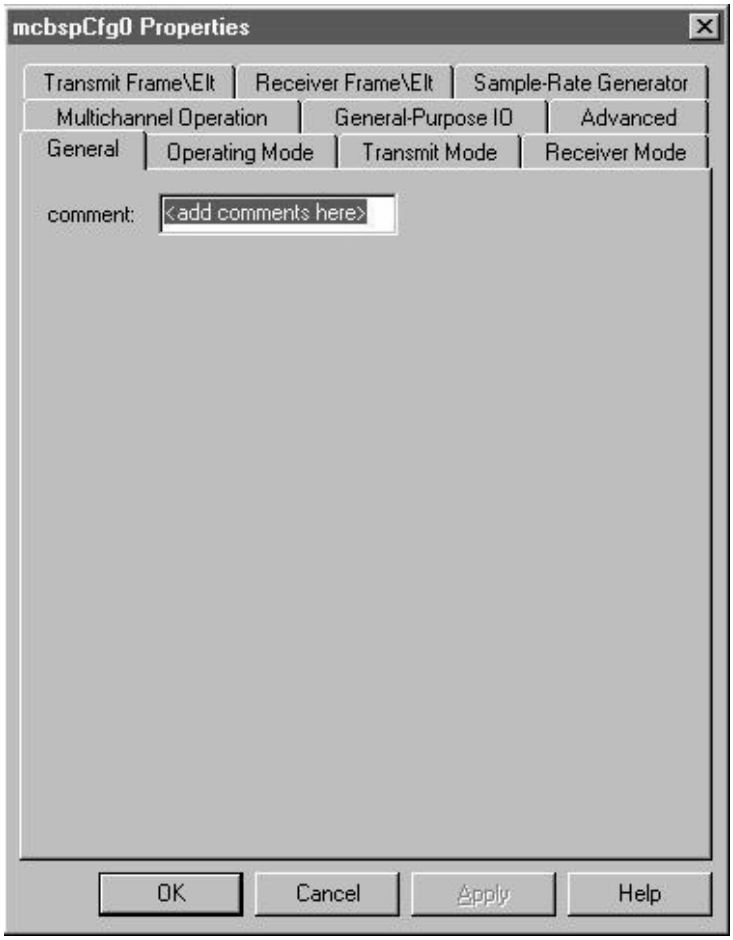
C.9.1.3 Properties Page of the Object

The Properties pages allow you to set the memory-mapped registers related to the McBSP Port. The configuration options are divided into the following tab pages:

- Operation Mode: Digital Loopback
- Transmit Mode: Interrupt mode, Frame Sync, Clock control
- Transmit Frame\Elt.: Phase, elements-per-word
- Receiver Mode: Interrupt mode, Frame Sync, Clock control
- Receiver Frame\Elt.: Phase, elements-per-word
- Sample-Rate Generator: Sample-Rate Generator (Frame Setup)
- Multichannel Operation: Element and Block partitioning
- General-Purpose I/O: enables the GPIOs of McBSP/set output value
- Some fields are activated according to the setup of the Transmitter, Receiver, and Sample-rate generator options
- Advanced Page: Summary of the previous pages. This page contains the full hexadecimal register values and reflects the setting of the options done under the previous pages
- The full register values can be entered directly and the new options will be mirrored on the corresponding pages automatically

Figure C–54 shows the McBSP Properties page.

Figure C–54. McBSP Properties Page



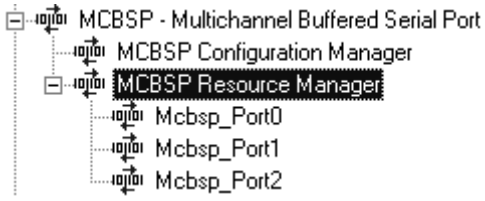
Each tab page is composed of several options that are set to a default value (at device reset).

The options represent the fields of the McBSP registers; the associated field name is shown in parenthesis. For further details on the fields and registers, refer to the *Multichannel Buffered Serial Ports* chapter of the *TMS320C6000 Peripherals Reference Guide* (SPRU190).

C.9.2 McBSP Resource Manager

The McBSP Resource Manager allows you to generate the `MCBSP_open()` and the `MCBSP_config()` CSL functions.

Figure C–55. McBSP Resource Manager Menu



C.9.2.1 Predefined Objects

Three handle objects are predefined and each of them is associated with a supported on-chip McBSP port.

The third McBSP, MCBSP_Port2, is supported by the C6202 and C6203 chips.

- **MCBSP_Port0** – Default handle name: hMcbsp0
- **MCBSP_Port1** – Default handle name: hMcbsp1
- **MCBSP_Port2** – Default handle name: hMcbsp2

Note: The above objects cannot be deleted. They can be renamed only.

A configuration can be enabled if at least one configuration object was defined previously (see section C.9.1).

C.9.2.2 Properties Page

The Properties page is accessible by right-clicking on the drop-down menu option, Properties.

The first time the Properties page appears, only the Open McBSP Port checkbox can be selected.

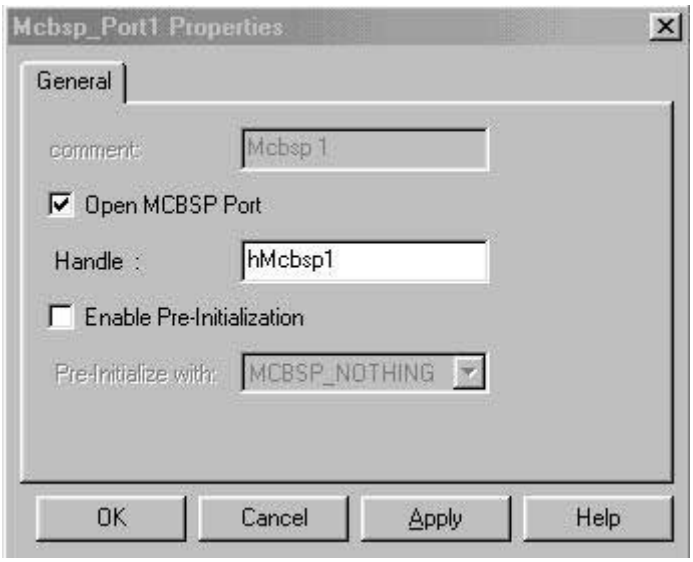
MCBSP_NOTHING is used to indicate that there is no configuration object selected for this serial port.

To enable Enable Pre-Initialization, create a new configuration object first (see section C.9.1). When a configuration object is available, Enable Pre-Initialization can be checked and you can select a configuration object from the drop-down list.

If MCBSP_NOTHING remains selected, no configuration object will be generated for the related McBSP handle. (see section C.9.3).

In Figure C–56, the Open McBSP Port option is checked and the handle object hMcbsp0 is now accessible (renaming allowed). The `MCBSP_open( )` function will be generated with hMcbsp0 containing the return handle address.

Figure C–56. McBSP Properties Page With Handle Object Accessible



C.9.3 C Code Generation for McBSP Module

The two C files generated from the configuration tool are a header file and a source file.

C.9.3.1 Header File

The generated header file *cdbnamecfg.h* includes the csl header files required for the McBSP module: csl.h, and csl_mcbsp.h. It contains the declaration of the McBSP handles and configuration objects.

Example C–25. McBSP Header File

```
extern far MCBSP_Config mcbsCfg0;
extern far MCBSP_Handle hMcbsp1;
```

C.9.3.2 Source File

The source file includes the declaration of the handle object and the configuration structures.

Example C–26. McBSP Source File (Declaration Section)

```
/*  Config Structures */
MCBSP_Config mcbspCfg0 = {
    0x00008000,        /*  Serial Port Control Reg. (SPCR)   */
    0x000000A0,        /*  Receiver Control Reg. (RCR)   */
    0x000000A0,        /*  Transmitter Control Reg. (XCR)   */
    0x203F1F0F,        /*  Sample–Rate Generator Reg. (SRGR)   */
    0x00000000,        /*  Multichannel Control Reg. (MCR)   */
    0x00000000,        /*  Receiver Channel Enable(RCER)   */
    0x00000000,        /*  Transmitter Channel Enable(XCER)   */
    0x00000A00         /*  Pin Control Reg. (PCR)   */
};

/*  Handles  */
MCBSP_Handle hMcbsp1;
```

The source file contains the Handle and Configuration Pre-Initialization using CSL McBSP APIs `MCBSP_open()` and `MCBSP_config()`. These two functions are encapsulated in a unique function, `CSL_cfgInit()`. `MCBSP_open()` and `MCBSP_config()` will be generated only if Open McBSP Port and Enable Pre-Initialization (with a selected configuration other than MCBSP_NOTHING) are, respectively, checked under the McBSP Resource Manager Properties page.

Example C–27. McBSP Source File (Body Section)

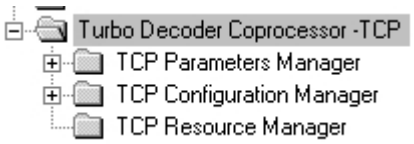
```
void CSL_cfgInit()
{
    hMcbsp1 = MCBSP_open(MCBSP_DEV1, MCBSP_OPEN_RESET);
    MCBSP_config(hMcbsp1, &mcbspCfg0);
}
```

C.10 Configuring the TCP Module Using CSL GUI

The TMS320C6000™ Chip Support Library (CSL) graphical user interface is part of the DSP/BIOS™ Configuration Tool integrated in Code Composer Studio™ IDE. This GUI reduces C-code typing and offers an easy way of using on-chip peripherals by programming the associated memory-mapped registers through the Properties pages.

The sections that follow provide explanations and screen captures to show you how to use the DSP/BIOS Configuration Tool.

Figure C–57. TCP Sections Menu



The TCP includes the following three sections:

- **TCP Parameters Manager** allows you to create a configuration object of TCP Basic Parameters.
- **TCP Configuration Manager** allows you to create configuration objects by setting the TCP programmable parameters via Input Control registers (IC0–5).
- **TCP Resource Manager** allows you to associate a pre-configuration object to the TCP module.

C.10.1 TCP Parameters Manager

The Parameters Manager allows the user to create a TCP configuration object of TCP_BaseParams type through the Properties page.

C.10.1.1 Inserting a Configuration Object

There is no predefined Base Param object. To insert a new configuration object, access the drop-down menu, click on the TCP Parameters Manager, and select Insert tcpBaseParam. The Base Param objects can be renamed. Their use depends on the on-chip device resources. Because there is only one TCP, only one set of parameters can be used at a time.

Note:

The number of Base Param objects is unlimited. Several sets of basic parameters can be created and the user can use one for configuring the TCP device by selecting the Base Param object under the TCP Resource Manager. The goal is to provide more flexibility. The user can generate several Base Param objects and use them at different times in his or her own program main.c.

C.10.1.2 Deleting/Renaming a Configuration Object

To delete or rename an object, use the drop-down menu by clicking on the Base Param object to be deleted or renamed.

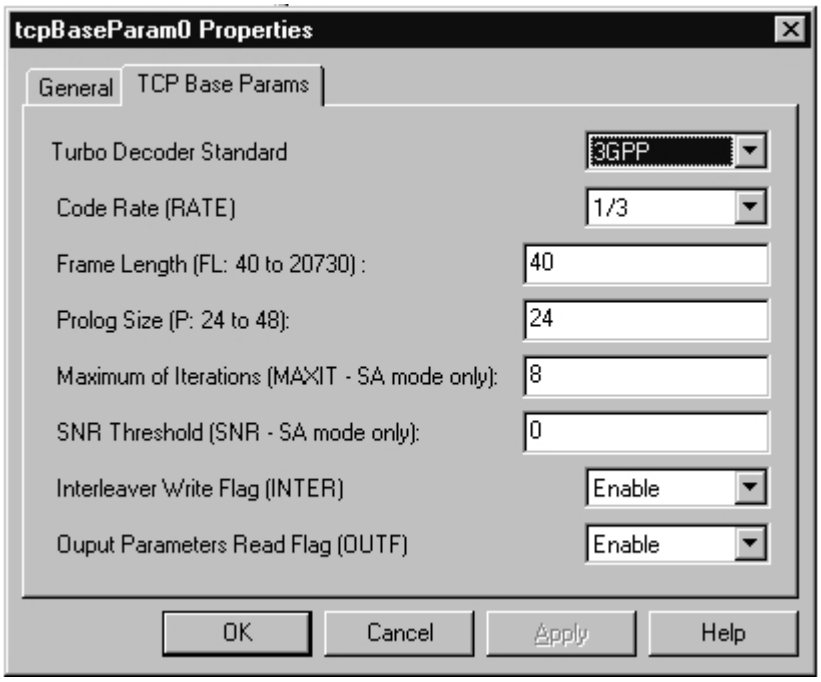
If a Base Param object is selected in the TCP Resource Manager (see section C.10.3) the Delete and Rename options are grayed-out and are non-usable for this object. Instead, the Show Dependency option is accessible and shows that the TCP Resource Manager is using this object. See section C.2, *Introduction to DSP/BIOS Configuration Tool: CSL Tree*, on page C-3 for further details.

C.10.1.3 Properties Page of the Object

The Properties page allows you to set some required TCP parameters under a basic parameters structure (TCP_BaseParams type). See Figure C–58.

```
typedef struct {
   TCP_Standard standard;
   TCP_Rate     rate;
   Uint16       frameLen;
   Uint8        prologSize;
   Uint8        maxIter;
   Uint8        snr;
   Uint8        intFlag;
   Uint8        outParmFlag;
} TCP_BaseParams;
```

Figure C–58. TCP Basic Parameter Properties Dialog



C.10.2 TCP Configuration Manager

The TCP Configuration Manager allows you to create TCP configurations through the Properties page and to generate the configuration objects.

C.10.2.1 Inserting a Configuration Object

There is no predefined configuration object. To insert a new configuration object, access the drop-down menu, click on the TCP Configuration Manager, and select Insert tcpCfg. The configuration objects can be renamed. Their use depends on the on-chip device resources. Because there is only one TCP, only one configuration will be used.

Note:

The number of configuration objects is unlimited. Several configurations can be created and the user can select one for TCP and can change the configuration later just by selecting another configuration under the TCP Resource Manager. The goal is to provide more flexibility and to reduce the time required to modify register values.

C.10.2.2 Deleting/Renaming a Configuration Object

To delete or rename an object, use the drop-down menu by clicking on the configuration object to be deleted or renamed.

If a configuration object is selected in the TCP Resource Manager (see section C.10.3) the Delete and Rename options are grayed-out and are non-usable for this object. The Show Dependency option is accessible and shows that the TCP Resource Manager is using the configuration object. See section 2.1, *Introduction to DSP/BIOS Configuration Tool: CSL Tree*, on page 2-2.

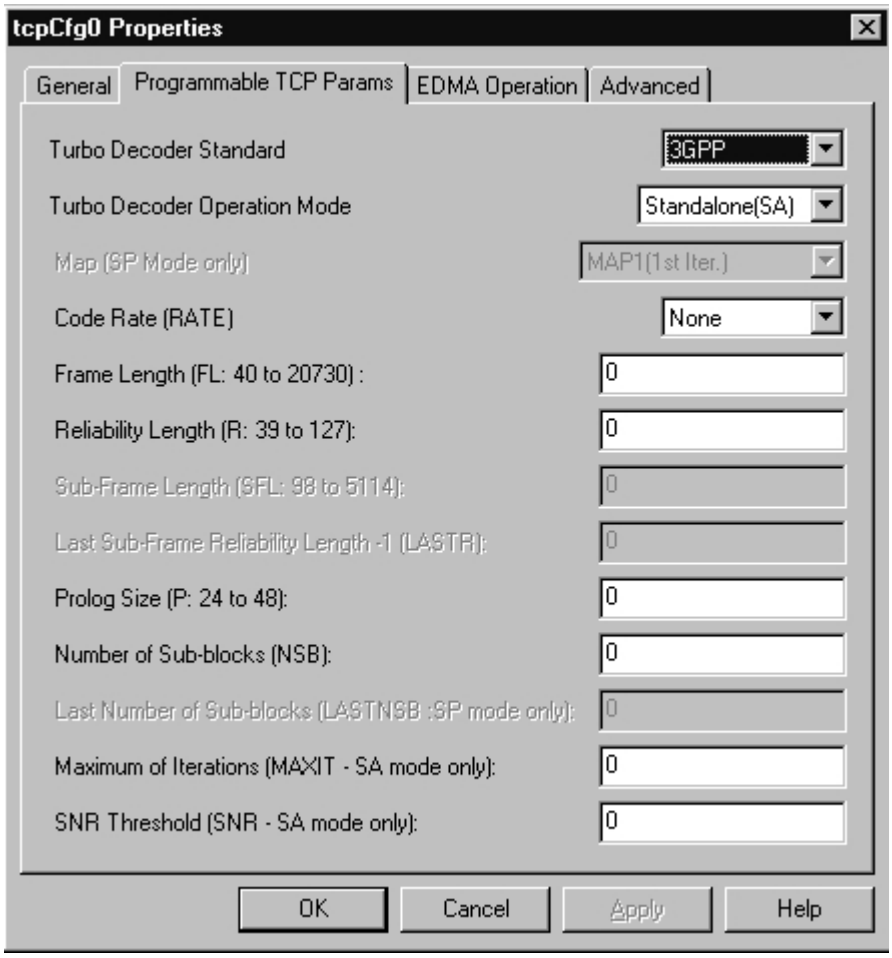
C.10.2.3 Properties Page of the Object

The Properties pages allow you to set the memory-mapped registers related to the TCP. The configuration options are divided into the following tab pages:

- Programmable TCP Params: TCP parameters dedicated to TCP functionality.
- EDMA Operation: TCP parameters related to EDMA operations.
- Advanced Page: This page contains the full hexadecimal register values and reflects the setting of the previous tab pages. Also, the full register values can be entered directly and the new options will be mirrored on the related tab pages automatically.

Figure C–59 shows the TCP configuration properties page dialog box.

Figure C–59. TCP Configuration Properties Page



Each tab page is composed of several options that are set to a default value (at device reset).

The options represent the fields of the TCP registers; the associated field name is shown in parenthesis. For further details on the fields and registers, refer to the *External Memory Interface* chapter in the *TMS320C6000 Peripherals Reference Guide* (SPRU190).

C.10.3 TCP Resource Manager

The TCP Resource Manager allows you to generate three different CSL functions with the associated pre-defined objects as parameters in order to configure the TCP module.

C.10.3.1 Properties Page

The Properties page is accessible by right-clicking on the drop-down menu option, Properties.

- **Enable Parameters Setting** generates the `TCP_genParams()` function.

  This function uses the Basic Parameters object (TCP_BaseParams) predefined and returns a TCP Parameter configuration structure which can be used by other TCP APIs.

  The first time you open the Property window, TCP_PARAMNULL is used to indicate that there is no Basic Parameter structure selected. “NULL” is used to indicate the user to enter a new name for the TCP Parameter object (type TCP_Params).

  The function will be generated if these three conditions are met:

  - Enable Parameter Setting check box is checked.
  - TCP_BaseParams object is other than “TCP_PARAMNULL”.
  - TCP_Params configuration name is other than “NULL”.

- **Set TCP Params Values to the IcConfig Object** generates `TCP_setParams()`.

  This function will use the TCP Parameter Object generated by `TCP_genParams()` to set the Input control values and return them under the IC configuration structure pre-defined (type TCP_ConfigIc). `TCP_setParams()` will be generated if these five conditions are met:

  - Enable Parameter Setting check box is checked.
  - TCP_BaseParams object is other than “TCP_PARAMNULL”.
  - Set TCP Params Values to the IcConfig Object checked.
  - TCP_Params configuration name is other than “NULL”.
  - TCP_ConfigIc configuration object name is other than “TCP_NOTHING”.

- **Enable Pre-initialization** generates the `TCP_icConfig()` function with the pre-defined configuration as parameter. This function allows the user to configure the TCP device.

  The first time the Properties page appears, TCP_NOTHING is used to indicate that there is no configuration object selected for this peripheral.

  To enable Enable Pre-Initialization, first create a new configuration object (see section C.10.2). When a configuration object is available, Enable Pre-Initialization can be checked and you can select a configuration object from the drop-down list.

If TCP_NOTHING remains selected, no configuration object will be generated.

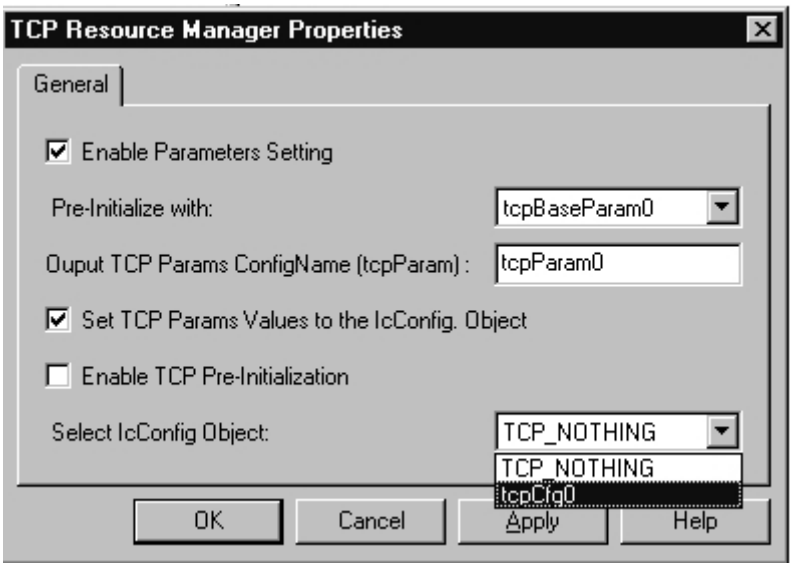
Note:

It is possible to set all IC registers values directly without using the TCP Parameters object. To do this, the "Set TCP Params Values to the IcConfig Object" check box must remain unchecked. The `TCP_setParams()` function will not be generated.

In summary, the user may generate several TCP_BaseParam objects, only one TCP_Param object, and several TCP_ConfigIc objects.

In the example shown in Figure C–60, the Enable Parameters Setting option is checked, so `TCP_genParams()` will be generated with tcpBaseParam0 as the parameter. In addition, the Set TCP Params Values to the IcConfig Object option is checked, which will cause `TCP_setParams()` to be generated with the Output TCP Params ConfigName, tcpParam0, as the parameter. A configuration object, tcpCfg0, is available for pre-initialization, but the Enable TCP Pre-Initialization option is not checked, so the `TCP_icConfig()` function will not be generated (see section C.10.4).

Figure C–60. TCP Resource Manager Dialog Box



C.10.4 C Code Generation for TCP Module

The two C files generated from the Configuration Tool are a header file and a source file.

C.10.4.1 Header File

The generated header file cdbnamecfg.h includes the csl header files required for the TCP device such as csl.h and csl_tcp.h It contains the declaration of the created objects under the TCP module : TCP_BaseParams and VCP_ConfigIC object types.

```
extern far TCP_BaseParams tcpBaseParam0;
extern far TCP_ConfigIc tcpCfg0;
```

Example C–28. TCP Header File

```
extern far TCP_ConfigIc tcpCfg0;
```

C.10.4.2 Source File

The source file includes the declaration of the configuration structures (values of the memory-mapped registers).

Example C–29. TCP Source File (Declaration Section)

```
... /*  Config Structures */
... TCP_BaseParams tcpBaseParam0 = {
     ..... TCP_STANDARD_3GPP,        /*   Decoder Standard  */
     ..... TCP_RATE_1_3,        /*   Rate  */
     ..... 40,        /*   Frame Length (FL: 40 to 20730)   */
     ..... 24,        /*   Prolog Size (P: 24 to 48)  */
      .... 8,        /*   Maximum of Iterations (MAXIT – SA mode only)*/
      .... 0,        /*   SNR Threshold (SNR – SA mode only)  */
      .... 1,        /*   Interleaver Write Flag   */
      .... 1         /*   Output Parameters Read Flag   */
... };

... TCP_ConfigIc tcpCfg0 = {
     ..... 0x00283000,        /*  Input Configuration Reg 0 (IC0)  */
     ..... 0x00270000,        /*  Input Configuration Reg 1 (IC1)  */
     ..... 0x00080118,        /*  Input Configuration Reg 2 (IC2)  */
     ..... 0x00000000,        /*  Input Configuration Reg 3 (IC3)  */
     ..... 0x00000000,        /*  Input Configuration Reg 4 (IC4)  */
     ..... 0x00000000,        /*  Input Configuration Reg 5 (IC5)  */
     ..... 0x00000000,        /*  Input Configuration Reg 6 (IC6)  */
     ..... 0x00000000,        /*  Input Configuration Reg 7 (IC7)  */
     ..... 0x00000000,        /*  Input Configuration Reg 8 (IC8)  */
     ..... 0x00000000,        /*  Input Configuration Reg 9 (IC9)  */
     ..... 0x00000000,        /*  Input Configuration Reg 10 (IC10)  */
     ..... 0x00000000         /*  Input Configuration Reg 11 (IC11)  */
... };
... /*  Handles  */
... TCP_Params tcpParam0;
```

Object created under the TCP Parameters Manager .

Object created under the TCP Configuration Manager .

TCP_Param Object name entered under the Resource Manager Properties page

The source file contains the Configuration Pre-Initialization TCP API, `TCP_config()`. This function is encapsulated in a unique function, `CSL_cfgInit()`.

TCP_config() will be generated only if Enable-Pre-Initialization is checked under the TCP Resource Manager Properties page (with a selected configuration other than TCP_NOTHING).

Example C–30. TCP Source File (Body Section)

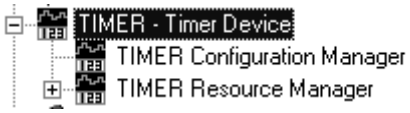
```
void CSL_cfgInit()
(
    TCP_genParams(&tcpBaseParam0, &tcpParam0);
    TCP_setParams(&tcpParam0,&tcpCfg0);
    TCP_icConfig(&tcpCfg0);
}
```

C.11 Configuring the TIMER Module Using CSL GUI

The TMS320C6000™ Chip Support Library (CSL) graphical user interface is part of the DSP/BIOS™ Configuration Tool integrated in Code Composer Studio™ IDE. This GUI reduces C-code typing and offers an easy way of using on-chip peripherals by programming the associated memory-mapped registers through the Properties pages.

The sections that follow provide explanations and screen captures to show you how to use the DSP/BIOS Configuration Tool.

Figure C–61. TIMER Sections Menu



The TIMER includes the following two sections:

- **TIMER Configuration Manager** allows you to create configuration objects (no predefined configuration objects).
- **TIMER Resource Manager** allows you to select a device that will be used and to associate a configuration object with that device. Three handle objects are predefined.

C.11.1 TIMER Configuration Manager

The TIMER Configuration Manager allows you to create device configurations through the Properties page and to generate the configuration objects.

C.11.1.1 Inserting a Configuration Object

There is no predefined configuration object available.

To configure a TIMER device through the memory-mapped registers, use the drop-down menu to insert a new configuration object by right-clicking on the TIMER Configuration Manager and selecting Insert timerCfg. The configuration objects can be renamed. Their use depends on the on-chip device resources. This means that if only two TIMERs are supported, a maximum of two configurations can be used.

Note:

The number of configuration objects is unlimited. Several configurations can be created and you can select the right one for a specific device and change the configuration later just by selecting a new one under the TIMER Resource Manager. This feature provides you with more flexibility and reduces the time required to modify register values.

C.11.1.2 Deleting/Renaming a Configuration Object

To delete or to rename an object, use the drop-down menu by clicking on the configuration object to be deleted or renamed.

If a configuration object is used by one of the predefined handle objects of the TIMER Resource Manager (see section C.11.2), the Delete and Rename options are grayed-out and are non-usable. The Show Dependency option is accessible and shows which device is using the configuration object. See section C.2, *Introduction to DSP/BIOS Configuration Tool: CSL Tree*, on page C-3.

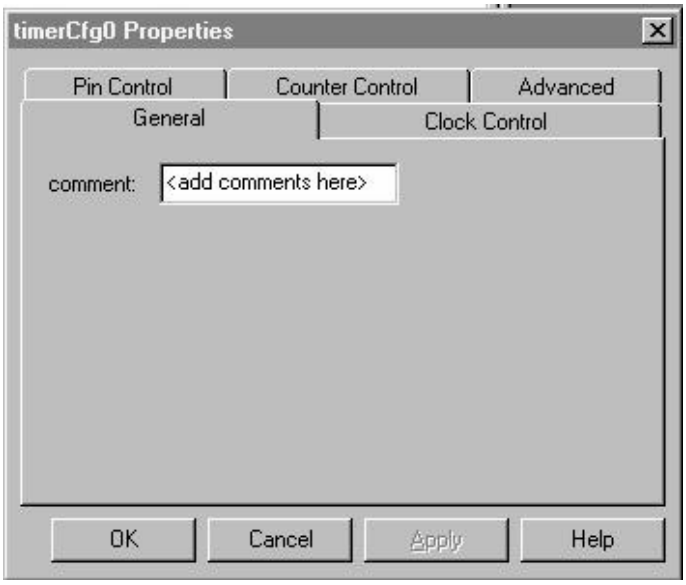
C.11.1.3 Properties Page of the Object

The Properties pages allow you to configure the memory-mapped registers related to the TIMER. The configuration options are divided into the following Tab pages:

- Clock Control: Clock configuration
- Counter Control: Counter configuration
- Pin Control: General-Purpose I/O pins
- Advanced Page: Summary of the previous three pages
- This page contains the full hexadecimal register values and reflects the setting of the three pages
- The full register values can be entered directly and the new options will be mirrored on the previous three pages automatically

Figure C–62 shows the TIMER Properties page dialog box.

Figure C–62. TIMER Properties Page



Each tab page is composed of several options that are set to a default value (at device reset).

The options represent the fields of the TIMER registers; the associated field name is shown in parenthesis. For further details on the fields and registers, refer to the *Timers* chapter in the *TMS320C6000 Peripherals Reference Guide* (SPRU190).

C.11.2 TIMER Resource Manager

The TIMER Resource Manager allows you to generate the `TIMER_open()` and the `TIMER_config()` CSL functions.

Figure C–63. TIMER Resource Manager Menu

C.11.2.1 Predefined Objects

Three handle objects are predefined and each of them is associated with a supported on-chip TIMER device.

(The third TIMER, TIMER_Device2, is supported by the TMS320C64x™ DSP only and is available for CCS releases 2.1 and above.)

- **TIMER_Device0** – Default handle name: hTimer0
- **TIMER_Device1** – Default handle name: hTimer1
- **TIMER_Device2** – Default handle name: hTimer2

Note: The above objects cannot be deleted but can be renamed.

A configuration is enabled if at least one configuration object was defined previously in section C.11.1.

C.11.2.2 Properties Page

The Properties page is accessible by right-clicking on the drop-down menu option Properties.

The first time the Properties page appears, only the Open TIMER Device check-box can be selected.

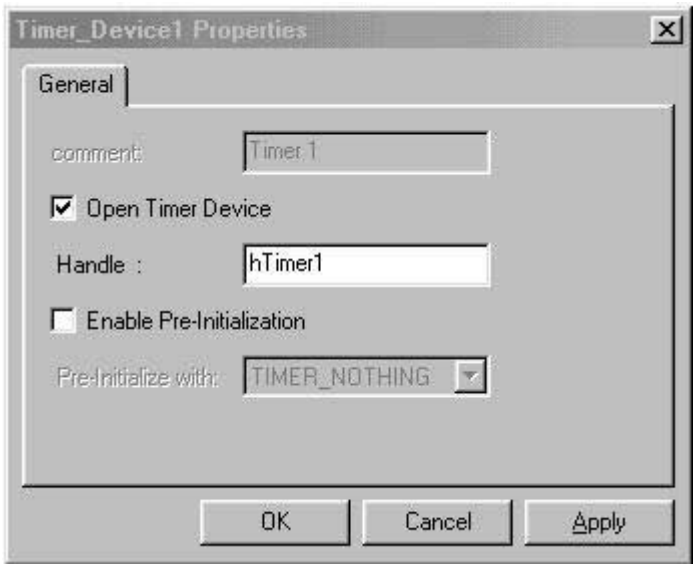
TIMER_NOTHING is used to indicate that there is no configuration object selected for this device.

To enable Enable Pre-Initialization, create a new configuration object first (see section C.11.1). When a configuration object is available, Enable Pre-Initialization can be checked and you can select a configuration object on the drop-down list.

If TIMER_NOTHING remains selected, no configuration object will be generated for the related TIMER handle (see section C.11.3).

In Figure C–64, the Open TIMER Device option is checked and the handle object hTimer0 is now accessible (renaming allowed). The `TIMER_open( )` function will be generated with hTimer0 containing the return handle address.

Figure C–64. TIMER Properties Page With Handle Object Accessible



C.11.3 C Code Generation for TIMER Module

The two C files generated from the configuration tool are a header file and a source file.

C.11.3.1 Header File

The generated header file *cdbnamecfg.h* includes the csl header files required for the TIMER module: csl.h, and csl_timer.h. It contains the declaration of the TIMER handles and configuration objects.

Example C–31. TIMER Header File

```
extern far TIMER_Config timerCfg1;
extern far TIMER_Handle hTimer1
```

C.11.3.2 Source File

The source file includes the declaration of the handle object and the configuration structures.

Example C–32. TIMER Source File (Declaration Section)

```
/*  Config Structures */
TIMER_Config timerCfg1 = {
    0x00000000,        /*  Control Register (CTL)   */
    0x0000FFFF,        /*  Period Register (PRD)   */
    0x00000000         /*  Counter Register (CNT)   */
};

/*  Handles  */
TIMER_Handle hTimer1;
```

The source file contains the Handle and Configuration Pre-Initialization using CSL TIMER APIs `TIMER_open()` and `TIMER_config()`. These two functions are encapsulated into a unique function, `CSL_cfgInit()`.

`TIMER_open()` and `TIMER_config()` will be generated only if both Open TIMER Device and Enable-Pre-Initialization (with timerCfg0) are checked on the TIMER Resource Manager Properties page.

Example C–33. TIMER Source File (Body Section)

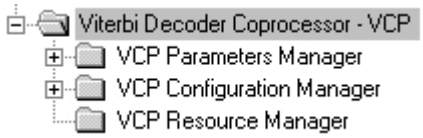
```
    void CSL_cfgInit()
    {
       hTimer1 = TIMER_open(TIMER_DEV1, TIMER_OPEN_RESET);
       TIMER_config(hTimer1, &timerCfg1);

    }
```

C.12 Configuring the VCP Module Using CSL GUI

The TMS320C6000™ Chip Support Library (CSL) graphical interface is part of the DSP/BIOS™ configuration tool integrated in Code Composer Studio™ IDE. This graphical user interface benefits you by reducing your C-code typing and offering you an easy way of using on-chip peripherals by programming the associated memory-mapped registers through the Properties pages.

The sections that follow provide explanations and screen captures to show you how to use the DSP/BIOS Configuration Tool.

Figure C–65. VCP Sections Menu



The VCP includes the following three sections:

- **VCP Parameters Manager** allows you to create a configuration object of VCP Parameters.
- **VCP Configuration Manager** allows you to create configuration objects by setting the VCP programmable parameters via Input Control registers (IC0–3).
- **VCP Resource Manager** allows you to associate a pre-configuration object to the VCP module.

C.12.1 VCP Parameters Manager

The Parameter Manager allows you to create a VCP configuration object of VCP_BaseParams type through the Properties page.

C.12.1.1 Inserting a Configuration Object

There is no predefined Base Param object. To insert a new configuration object, access the drop-down menu, click on the VCP Parameters Manager, and select Insert vcpBaseParam. The Base Param object can be renamed. Their use depends on the on-chip device resources. Because there is only one VCP, only one set of parameters can be used at a time.

Note:

The number of Base Param objects is unlimited. Several sets of basic parameters can be created and the user can use one for configuring the VCP device by selecting the Base Param object under the VCP Resource Manager. The goal is to provide more flexibility. The user can generate several Base Param objects and use them at different times in his or her own program main.c.

C.12.1.2 Deleting/Renaming a Configuration Object

To delete or rename an object, use the drop-down menu by clicking on the Base Param object to be deleted or renamed.

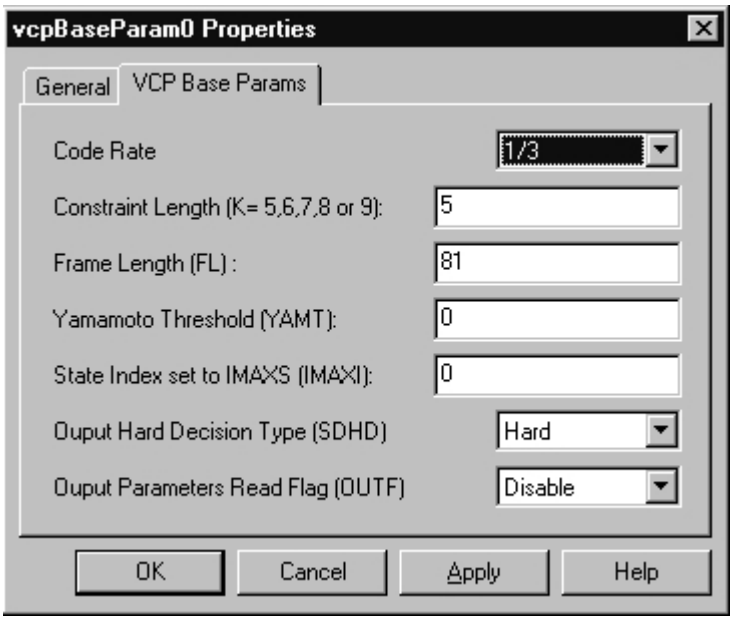
If a Base Param object is selected in the VCP Resource Manager (see section C.12.3), the Delete and Rename options are grayed-out and are non-usable for this object. Instead, the Show Dependency option is accessible and shows that the VCP resource manager is using the configuration object. See section C.2, *Introduction to DSP/BIOS Configuration Tool: CSL Tree*, on page C-3 for more details.

C.12.1.3 Properties Page of the Object

The Properties page allows the user to set some required VCP parameters under a basic parameters structure (VCP_BaseParams type). See Figure C–66.

```
typedef struct {
   VCP_Rate     rate;
   Uint8        constLen;
   Uint16       frameLen;
   Uint16       yamTh;
   Uint8        stateNum;
   Uint8        decision;
   Uint8        readFlag;
} VCP_BaseParams;
```

Figure C–66. VCP Basic Parameter Properties Dialog



C.12.2 VCP Configuration Manager

The VCP Configuration Manager allows you to create VCP configurations through the Properties page and generate the configuration objects.

C.12.2.1 Inserting a Configuration Object

There is no predefined configuration object. To insert a new configuration object, access the drop-down menu, click on the VCP Configuration Manager, and select Insert VCPCfg. The configuration objects can be renamed. Their use depends on the on-chip device resources. Because there is only one VCP, only one configuration will be used.

Note: The number of configuration objects is unlimited. Several configurations can be created and the user can select one for VCP and can change the configuration later just by selecting another configuration under the VCP Resource Manager. The goal is to provide more flexibility and to reduce the time required to modify register values.

C.12.2.2 Deleting/Renaming a Configuration Object

To delete or rename an object, use the drop-down menu by clicking on the configuration object to be deleted or renamed.

If a configuration object is selected in the VCP Resource Manager (see section C.12.3). the Delete and Rename options are grayed-out and non-usable for this object. The Show Dependency option is accessible and shows that the VCP resource manager is using the configuration object. See Section 2.1, *Introduction to DSP/BIOS Configuration Tool: CSL Tree*, on page 2-2.

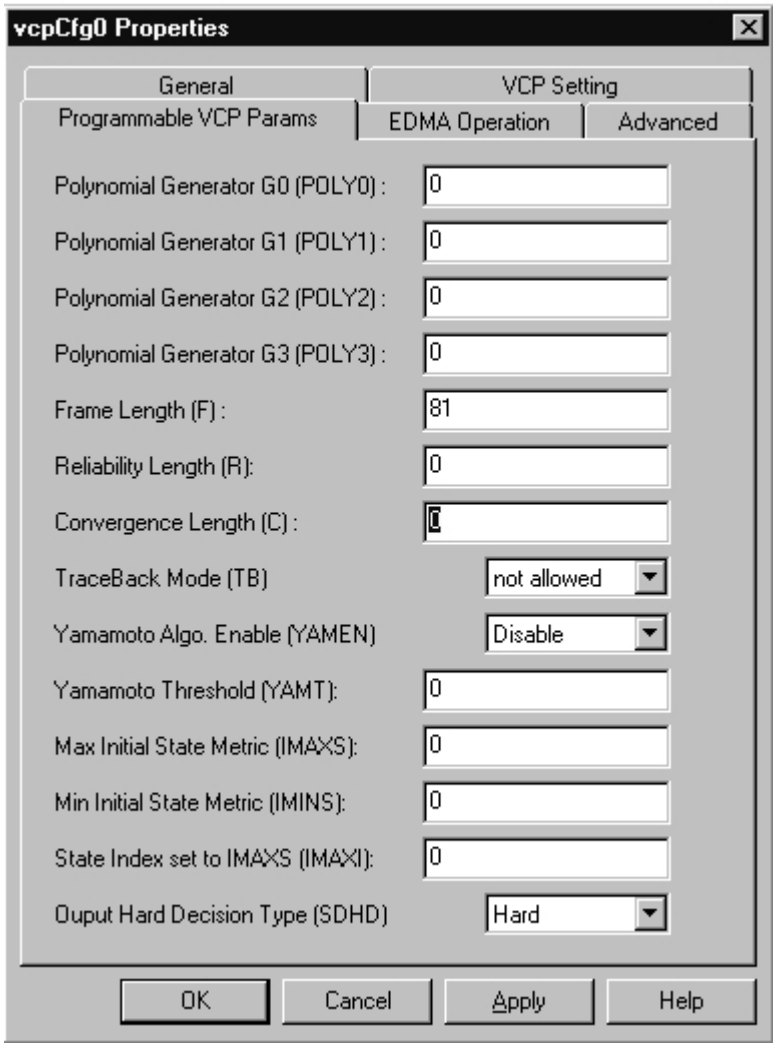
C.12.2.3 Properties Page of the Object

The Properties pages allow you to set the memory-mapped registers related to the VCP. The configuration options are divided into the following tab pages:

- Programmable VCP Params: VCP parameters dedicated to VCP functionality
- EDMA Operation : VCP parameters related to EDMA operations
- Advanced Page: This page contains the full hexadecimal register values and reflects the setting of the previous tab pages. Also, the full register values can be entered directly and the new options will be mirrored on the related tab pages automatically.

Figure C–67 shows the VCP Configuration Properties page dialog box.

Figure C–67. VCP Configuration Properties Page



Each tab page is composed of several options that are set to a default value (at device reset).

The options represent the fields of the VCP registers; the associated field name is shown in parenthesis. For further details on the fields and registers, refer to the *External Memory Interface* chapter in the *TMS320C6000 Peripherals Reference Guide* (SPRU190).

C.12.3 VCP Resource Manager

The VCP Resource Manager allows you to generate three different CSL functions with the associated pre-defined objects as parameters in order to configure the VCP module.

C.12.3.1 Properties Page

The Properties page is accessible by right-clicking on the drop-down menu option Properties.

- **Enable Parameter Setting** generates the `VCP_genParams()` function.

  This function uses the Basic Parameters object (VCP_BaseParams) predefined and returns a VCP Parameter configuration structure which can be used by others VCP APIs.

  The first time the user opens the Properties window, VCP_PARAMNULL is used to indicate that there is no Basic Parameter structure selected, and "NULL" is used to indicate the user to enter a new name for the TCP Parameter object (type VCP_Params).

  `VCP_genParams()` is a powerful function which will calculate the polynomial constant values and other required VCP parameters, such as reliability length and trace back field. The values are returned under the VCP_Param object.

  The function will be generated if these three conditions are met:

  - Enable Parameter Setting check box is checked.
  - VCP_BaseParams object is something other than "VCP_PARAMNULL".
  - VCP_Params configuration name is something other than "NULL".

- **Set VCP Params Values to the IcConfig Object** generates `VCP_genIc()`.

  This function will use the TCP Parameter Object generated by VCP_genParams to set the Input control values and returns them under the IC configuration structure pre-defined (type VCP_ConfigIc). `VCP_genIc()` will be generated if these five conditions are met:

  - Enable Parameter Setting check box is checked.
  - VCP_BaseParams object is something other than "VCP_PARAMNULL".
  - Set VCP Params Values to the IcConfig Object is checked.
  - VCP_Params configuration name is something other than "NULL".

- VCP_ConfigIc configuration object name is something other than “VCP_NOTHING”.

❑ **Enable Pre-initialization**: will generate the `VCP_icConfig()` function with the pre-defined configuration as parameters. This function allows you to configure the VCP device.

The first time the Properties page appears, VCP_NOTHING is used to indicate that there is no configuration object selected for this peripheral.

To enable Enable Pre-Initialization, create a new configuration object first (see section C.12.2). When a configuration object is available, Enable Pre-Initialization can be checked and you can select a configuration object from the drop-down list.

If VCP_NOTHING remains selected, no configuration object will be generated.

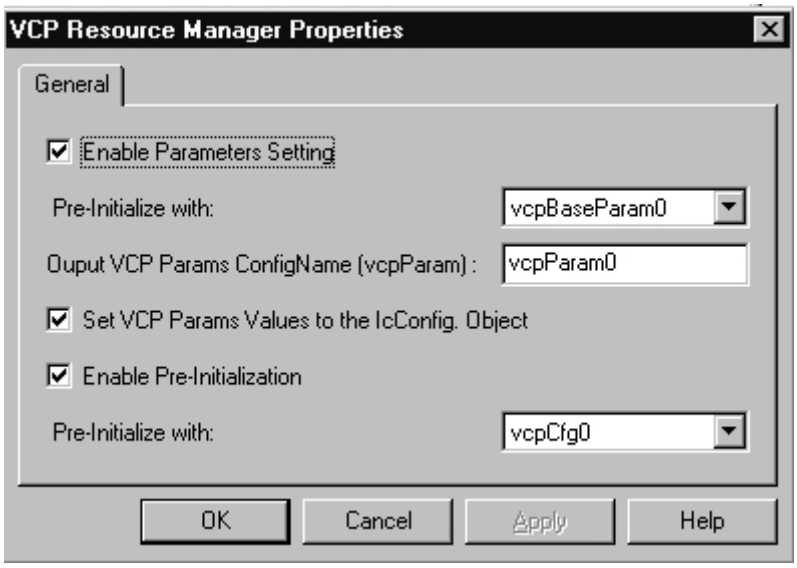
Note:

It is possible to set all IC registers values directly without using the VCP Parameters object.To do this, the “Set VCP Params Values to the IcConfig Object” check box must remain unchecked. The `VCP_genIc()` function will not be generated.

In summary, the user may generate several VCP_BaseParam objects, only one VCP_Param object, and several TCP_ConfigIc objects.

In the example shown in Figure C–68 the Enable Parameters Setting option is checked, so `VCP_genParams()` will be generated with vcpBaseParam0 as the parameter. In addition, the Set VCP Params Values to the IcConfig Object option is checked, which will cause `VCP_genIc()` to be generated with the Output VCP Params ConfigName, vcpParam0, as the parameter. A configuration object is available for pre-initialization, and the Enable VCP Pre-Initialization option is checked, so the `VCP_icConfig()` function will be generated with vcpCfg0 as the parameter (see section C.12.4).

Figure C–68. VCP Resource Manager Dialog Box



C.12.4 C Code Generation for VCP Module

The two C files generated from the configuration tool are a header file and a source file.

C.12.4.1 Header File

The generated header file *cdbnamecfg.h* includes the csl header files required for the VCP device, such as csl.h and csl_vcp.h. It also contains the declaration of the created objects under the VCP module: VCP_BaseParams and VCP_ConfigIC object types.

Example C–34. VCP Header File

```
extern far VCP_BaseParams vcpBaseParam0;
extern far VCP_ConfigIc vcpCfg0;
```

C.12.4.2 Source File

The source file includes the declaration of the configuration structures (values of the memory-mapped registers).

Example C–35. VCP Source File (Declaration Section)

```
... /*  Config Structures */
... VCP_BaseParams vcpBaseParam0 = {
...      3,        /*   Rate  */
...      9,        /*   Constraint Length (K= 5,6,7,8 or 9)  */
...      81,        /*   Frame Length (FL)   */
...      0,        /*   Yamamoto Threshold (YAMT)  */
...      0,        /*   State Index set to IMAXS (IMAXI)  */
...      0,        /*   Output Hard Decision Type   */
...      0         /*   Output Parameters Read Flag   */
... };
... VCP_ConfigIc vcpCfg0 = {
...     0x00C9B36F,        /*  Input Configuration Reg 0 (IC0)  */
...     0x00000000,        /*  Input Configuration Reg 1 (IC1)  */
...     0x00000051,        /*  Input Configuration Reg 2 (IC2)  */
...     0x00000018,        /*  Input Configuration Reg 3 (IC3)  */
...     0x00000018,        /*  Input Configuration Reg 4 (IC4)  */
...     0x41080000         /*  Input Configuration Reg 5 (IC5)  */
... };
... /*  Handles  */
... VCP_Params vcpParam0;
```

Object created under the VCP Parameters Manager

Object created under the VCP Configuration Manager

VCP_Param Object name entered under the Resource Manager Properties page

The source file body contains a unique function, `CSL_cfginit()`, which encapsulates the VCP functions generated by the DSP/BIOS Configuration Tool. As mentioned under VCP Resource Manager, the three functions `VCP_genParams()`, `VCP_genIc()`, and `VCP_icConfig()` will appear if the corresponding check boxes were checked.

Example C–36. VCP Source File (Body Section)

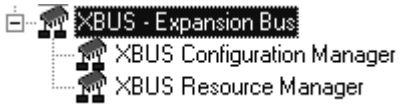
```
void CSL_cfgInit()
(
  VCP_genParams(&vcpBaseParam0, &vcpParam0);
  VCP_genIc(&vcpParam0,&vcpCfg0);
  VCP_icConfig(&vcpCfg0);


}
```

C.13 Configuring the XBUS Module Using CSL GUI

The TMS320C6000™ Chip Support Library (CSL) graphical interface is part of the DSP/BIOS™ configuration tool integrated in Code Composer Studio™ IDE. This graphical user interface benefits you by reducing your C-code typing and offering you an easy way of using on-chip peripherals by programming the associated memory-mapped registers through the Properties pages.

The sections that follow provide explanations and screen captures to show you how to use the DSP/BIOS configuration tool.

Figure C–69. XBUS Sections Menu



The XBUS includes the following two sections:

- **XBUS Configuration Manager** allows you to create configuration objects by setting the memory-mapped registers related to the XBUS.
- **XBUS Resource Manager** allows you to associate a preconfiguration object to the XBUS.

C.13.1 XBUS Configuration Manager

The XBUS Configuration Manager allows you to create XBUS configurations through the Properties page and to generate the configuration objects.

C.13.1.1 Inserting a Configuration Object

There is no predefined configuration object. To insert a new configuration object, access the drop-down menu, click on the XBUS Configuration Manager, and select Insert xbusCfg. The configuration objects can be renamed. Their use depends on the on-chip device resources. Because there is only one XBUS, only one configuration will be used.

Note:

The number of configuration objects is unlimited. Several configurations can be created and the user can select one for XBUS and can change the configuration later just by selecting another configuration under the XBUS Resource Manager. This feature allows you more flexibility and reduces the time required to modify register values.

C.13.1.2 Deleting/Renaming a Configuration Object

To delete or to rename an object, use the drop-down menu by clicking on the configuration object to be deleted or renamed. If a configuration object is selected in the XBUS Resource Manager (see section C.13.2), the Delete and Rename options are grayed-out and non-usable for this object. The Show Dependency option is accessible and shows that the XBUS resource manager is using the configuration object. See section C.2, *Introduction to DSP/BIOS Configuration Tool: CSL Tree*, on page C-3.

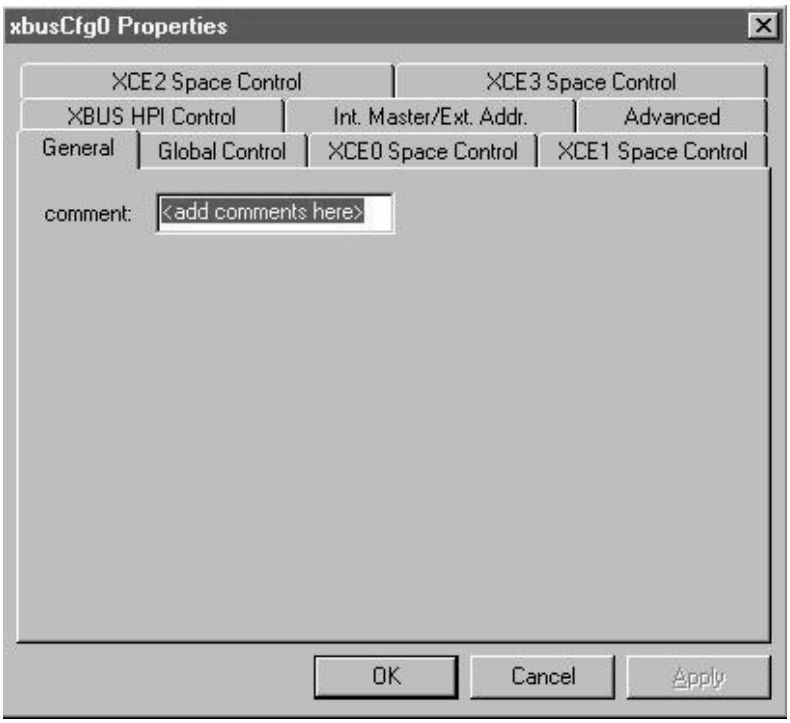
C.13.1.3 Properties Page of the Object

The Properties pages allow you to set the memory-mapped registers related to the XBUS. The configuration options are divided into the following tab pages:

- Global Control: FIFO clock enable and FIFO clock rate
- XCE0 Space Control: configuration of XCE0 memory space (XCE0CTL register)
- XCE1 Space Control: configuration of XCE1 memory space (XCE1CTL register)
- XCE2 Space Control: configuration of XCE2 memory space (XCE2CTL register)
- XCE3 Space Control: configuration of XCE3 memory space (XCE3CTL register)
- XBUS HPI Control: Allows you to configure expansion bus host port parameters.
- Int. Master/Ext. Addr.: Allows you to set values for expansion bus internal master address and expansion bus external address register.
- Advanced Page: This page contains the full hexadecimal register values and reflects the setting of the previous tab pages. The full register values can be entered directly and the new options will be mirrored on the related tab pages automatically.

Figure C–70 shows the XBUS Properties page dialog box.

Figure C–70. XBUS Properties Page



Each tab page is composed of several options that are set to a default value (at device reset).

The options represent the fields of the XBUS registers; the associated field name is shown in parenthesis. For further details of the fields and registers, refer to the *Expansion Bus* chapter of the *TMS320C6000 Peripheral References Guide* (SPRU190).

C.13.2 XBUS Resource Manager

The XBUS Resource Manager allows you to generate the XBUS_`config()` CSL function with the predefined configuration as parameter. Because only one XBUS is supported, only one resource is available and used as the default.

C.13.2.1 Properties Page

The Properties page is accessible by right-clicking on the drop-down menu option Properties.

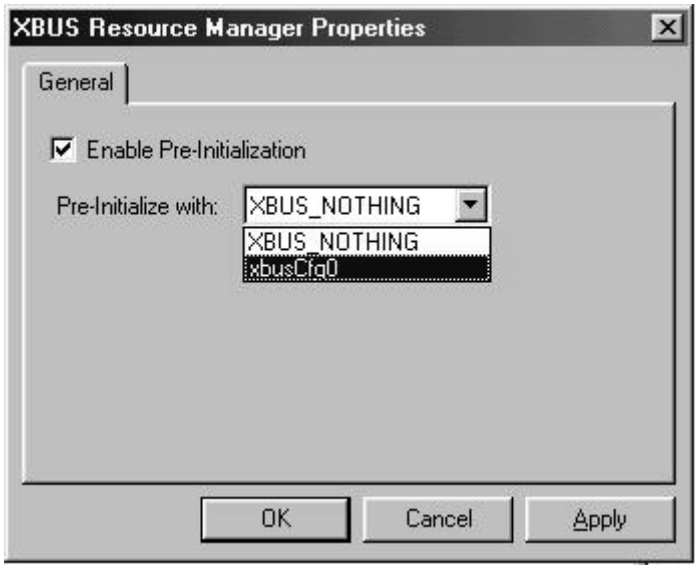
The first time the Properties page appears, XBUS_NOTHING is used to indicate that there is no configuration object selected for this peripheral.

To enable Enable Pre-Initialization, create a new configuration object first (see section C.13.1). When a configuration object is available, Enable Pre-Initialization can be checked and you can select a configuration object on the drop-down list.

If XBUS_NOTHING remains selected, no configuration object will be generated .

In Figure C–71, *XBUS Resource Manager Dialog Box*, the Enable Pre-Initialization option is checked, the XBUS_`config()` function will be generated with xbusCfg0 as the parameter (see section C.13.3).

Figure C–71. XBUS Resource Manager Dialog Box



C.13.3 C Code Generation for XBUS Module

The two C files generated from the configuration tool are a header file and a source file.

C.13.3.1 Header File

The generated header file *cdbnamecfg.h* includes the csl header files required for the XBUS module: csl.h, and csl_xbus.h. It contains the declaration of the XBUS handle objects.

Example C–37. XBUS Header File

```
extern far XBUS_Config xbusCfg0;
```

C.13.3.2 Source File

The source file includes the declaration of the configuration structures (values of the memory-mapped registers).

Example C–38. XBUS Source File (Declaration Section)

```
/*  Config Structures */
XBUS_Config xbusCfg0 = {
    0x00000000,  /* Global Control Register(XBGC) */
    0xFFFF3F23,  /* XCE0 Space Control Register(XCE0CTL)  */
    0xFFFF3F23,  /* XCE0 Space Control Register(XCE0CTL)  */
    0xFFFF3F23,  /* XCE0 Space Control Register(XCE0CTL)  */
    0xFFFF3F23,  /* XCE0 Space Control Register(XCE0CTL)  */
    0x00000000,  /* XBUS HPI Control Register(XBHC) */
    0x00000000,  /* XBUS Internal Master Addr Register(XBIMA) */
    0x00000000   /* XBUS External Address Register(XBEA) */
};
```

The source file contains the Configuration Pre-Initialization XBUS API, `XBUS_config()`. This function is encapsulated in a unique function, `CSL_cfgInit()`.

`XBUS_config()` will be generated only if Enable Pre-Initialization is checked under the XBUS Resource Manager Properties page (with a selected configuration other than XBUS_NOTHING). See Figure C–71.

Example C–39. XBUS Source File (Body Section)

```
void CSL_cfgInit()
{
    XBUS_config(&xbusCfg0);

}
```

Appendix D

Old and New Cache APIs

The L2 cache register names and the CSL cache coherence APIs have been renamed to better reflect the actual operation. All users are encouraged to switch from the old APIs to the new ones. The old APIs will still work, but will no longer be updated. Also, the old CSL version does not support some new C64x cache operations. Table D–1 and Table D–2 show the correct function calls for the new APIs, to replace the old ones. Table D–3 shows the mapping of the old L2 register names to the new L2 register names. Table D–4 shows the mapping of the new L2ALLOCx bit field names to the old bit field names (C64x only).

Table D–1. CSL APIs for L2 Cache Operations

Scope	Operation	Old API	New API†
Block	L2 Invalidate (C64x only)	N/A	CACHE_invL2(start address, byte count, CACHE_WAIT)
	L2 Writeback	CACHE_flush(CACHE_L2, start address, word count)	CACHE_wbL2(start address, byte count, CACHE_WAIT)
	L2 Writeback–Invalidate	CACHE_clean(CACHE_L2, start address, word count)	CACHE_wbInvL2(start address, byte count, CACHE_WAIT)
All L2 Cache	L2 Writeback All	CACHE_flush(CACHE_L2ALL, [ignored], [ignored])	CACHE_wbAllL2(CACHE_WAIT)
	L2 Writeback–Invalidate All	CACHE_clean(CACHE_L2ALL, [ignored], [ignored])	CACHE_wbInvAllL2(CACHE_WAIT)

† Refer CACHE chapter for the complete description of API.

Table D–2. CSL APIs for L1 Cache Operations

Scope	Operation	Old CSL Commands	New CSL Commands
Block	L1D Invalidate (C64x only)	N/A	CACHE_invL1d(start address, byte count, CACHE_WAIT)
	L1D Writeback–Invalidate	CACHE_flush(CACHE_L1D, start address, word count)	CACHE_wbInvL1d(start address, byte count, CACHE_WAIT)
	L1P Invalidate	CACHE_invalidate(CACHE_L1P, start address, word count)	CACHE_invL1p(start address, byte count, CACHE_WAIT)
All	L1P Invalidate	CACHE_invalidate(CACHE_L1PALL, [ignored], [ignored])	CACHE_invAllL1p()

Table D–3. Mapping of Old L2 Register Names to New L2 Register Names

Old Register Name	New Register Name	Description
L2CLEAN	L2WBINV	L2 Writeback–Invalidate All
L2FLUSH	L2WB	L2 Writeback All
L2CBAR	L2WIBAR	L2 Writeback–Invalidate Base Address Register
L2CWC	L2WIWC	L2 Writeback–Invalidate Word Count
L2FBAR	L2WBAR	L2 Writeback Base Address Register
L2FWC	L2WWC	L2 Writeback Word Count
L2IBAR	L2IBAR	L2 Invalidate Base Address Register (C64x only)
L2IWC	L2IWC	L2 Invalidate Word Count (C64x only)
L1DFBAR	L1DWIBAR	L1D Writeback–Invalidate Base Address Register
L1DFWC	L1DWIWC	L1D Writeback–Invalidate Word Count
L1DIBAR	L1DIBAR	L1D Invalidate Base Address Register (C64x only)
L1DIWC	L1DIWC	L1D Invalidate Word Count (C64x only)
L1PFBAR	L1PIBAR	L1P Invalidate Base Address Register
L1PFWC	L1PIWC	L1P Invalidate Word Count

Table D–4. Mapping of New L2ALLOCx Bit Field Names to Old Bit Field Names (C64x only)

Register	Old Bit Field Names	New Bit Field Names	Description
L2ALLOC1	L2ALLOC	Q1CNT	L2 allocation priority queue 1
L2ALLOC2	L2ALLOC	Q2CNT	L2 allocation priority queue 2
L2ALLOC3	L2ALLOC	Q3CNT	L2 allocation priority queue 3
L2ALLOC4	L2ALLOC	Q4CNT	L2 allocation priority queue 4

Appendix E

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 (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

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

PCI: *Peripheral component interconnect interface; see PCI module*.

PCI module: PCI is an API module that includes APIs which are dedicated to DSP-PCI Master transfers, EEPROM operations, and power management

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 micro-programmed 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

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.

X

xbus:

XBUS module: The XBUS module is an API module used for configuring the tXBUS registers.

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X