TMS320C54x Chip Support Library API Reference Guide

SPRU420D
May 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 that 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.

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Preface

Read This First

About This Manual

The TMS320C54x™ DSP Chip Support Library (CSL) provides C-program functions to configure and control on-chip peripherals, which makes it easier for algorithms to run in a real system. The CSL provides peripheral ease of use, shortened development time, portability, and hardware abstraction, along with some level of standardization and compatibility among devices. A version of the CSL is available for all TMS320C54x™ DSP devices.

How to Use This Manual

The contents of the TMS320C5000™ DSP Chip Support Library (CSL) are as follows:

- Chapter 1, *CSL Overview*, provides an overview of the CSL, includes tables showing CSL module support for various C5000 devices, and lists the CSL modules.
- Chapter 2, *How To Use CSL*, provides basic examples of how to use CSL functions with or without using the DSP/BIOS™ Configuration Tool, and shows how to define Build options for the CSL in the Code Composer Studio™ environment.
- Chapters 3-16 provide basic examples, functions, macros, and CSL GUI configurations for the individual CSL modules.
- Appendix A provides examples of how to use CSL C5000 Registers.

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.
- TMS320C54x™ DSP devices are referred to throughout this reference guide as C5401, C5402, etc.

Related Documentation From Texas Instruments

The following books describe the TMS320C54x™ DSP 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 are located on the internet at http://www.ti.com.

TMS320C54x Assembly Language Tools User's Guide (literature number SPRU102) 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 C54x generation of devices.

TMS320C54x Optimizing C Compiler User's Guide (literature number SPRU103) describes the C54x C compiler. This C compiler accepts ANSI standard C source code and produces TMS320 assembly language source code for the C54x generation of devices.

TMS320C54x Simulator Getting Started (literature number SPRU137) describes how to install the TMS320C54x simulator and the C source debugger for the C54x. The installation for MS-DOS™, PC-DOS™, SunOS™, Solaris™, and HP-UX™ systems is covered.

TMS320C54x Evaluation Module Technical Reference (literature number SPRU135) describes the C54x evaluation module, its features, design details and external interfaces.

TMS320C54x Simulator Getting Started Guide (literature number SPRU137) describes how to install the TMS320C54x simulator and the C source debugger for the C54x. The installation for Windows 3.1, SunOS™, and HP-UX™ systems is covered.

TMS320C54x Code Generation Tools Getting Started Guide (literature number SPRU147) describes how to install the TMS320C54x assembly language tools and the C compiler for the C54x devices. The installation for MS-DOS™, OS/2™, SunOS™, Solaris™, and HP-UX™ 9.0x systems is covered.

TMS320C54x Simulator Addendum (literature number SPRU170) tells you how to define and use a memory map to simulate ports for the C54x. This addendum to the *TMS320C5xx C Source Debugger User's Guide* discusses standard serial ports, buffered serial ports, and time division multiplexed (TDM) serial ports.

Trademarks

Contents

Figures

Tables

Examples

Chapter 1

CSL Overview

This chapter introduces the Chip Support Library(CSL), briefly describes its architecture, and provides a generic overview of the collection of functions, macros, and constants that are needed to program DSP peripherals.

Topic **Page**

1.1 Introduction to the CSL

The Chip Support Library(CSL) is a collection of functions, macros, and symbols used to configure and control on-chip peripherals. The goal is peripheral ease of use, shortened development time, portability, hardware abstraction, and some level of standardization and compatibility among TI devices.

The CSL is a fully scalable component of DSP/BIOS™, however, it does not require the use of other DSP/BIOS components to operate.

1.1.1 Benefits of the CSL

- **Standard Protocol to Program Peripherals**

  The CSL provides you with a standard protocol to use each time you program on-chip peripherals. This protocol includes specific data types and macros to define peripheral configurations, and standard functions to implement the various operations of each peripheral.

- **Automated Peripheral Pre-initialization via the CSL GUI**

  The CSL integrates a graphical user interface (GUI) into the DSP/BIOS configuration tool. The CSL GUI pre-initializes peripherals by generating correct peripheral register values and C files. The C files initialize peripherals by using functions provided in the CSL.

  Section 2.2, *Using CSL with the CSL GUI*, details the available CSL modules found in the CSL DSP/BIOS Configuration Tool.

- **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.

- **Symbol Peripheral Descriptions**

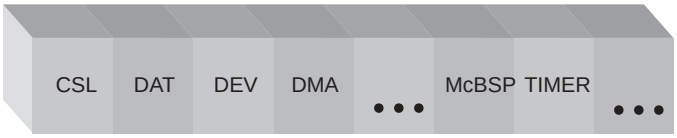
  As a side benefit to the creation of CSL, a complete symbolic description of all peripheral registers and register fields has been created. It is suggested that you use the higher level protocols described in the first two benefits, as these are less device specific, thus making it easier to migrate your code to newer versions of DSPs.

1.1.2 CSL Architecture

The CSL consists of modules that are built and archived into a library file. Each peripheral is covered by a single module while additional modules provide general programming support.

Figure 1–1 illustrates the individual CSL modules. This architecture allows for future expansion because new modules can be added as new peripherals emerge.

Figure 1–1. API Modules



Although each CSL module provides a unique set of functions, some interdependency exists between the modules. For example, the DMA module depends on the IRQ module because of DMA interrupts; as a result, when you link code that uses the DMA module, a portion of the IRQ module is linked automatically.

Each module has a compile-time support symbol that denotes whether or not the module is supported for a given device. For example, the symbol _DMA_SUPPORT has a value of 1 if the current device supports it and a value of 0 otherwise. The available symbols are located in Table 1–1. You can use these support symbols in your application code to make decisions.

Table 1–1 lists general and peripheral modules with their associated include file, the module support symbol, and a column that specifies which modules are/are not supported by the CSL GUI. 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	CSL GUI Support
CHIP	General device module	csl_chip.h	_CHIP_SUPPORT	NO
DAA	Digital Access Arrangement	csl_daa.h	_DAA_SUPPORT	NO
DAT	A data copy/fill module based on the DMA	csl_dat.h	_DAT_SUPPORT	NO
DMA	DMA Peripheral	csl_dma.h	_DMA_SUPPORT	YES
EBUS	External bus interface	csl_ebus.h	_EBUS_SUPPORT	YES
GPIO	Non-multiplexed general purpose I/O	csl_gpio.h	_GPIO_SUPPORT	YES
HPI	HPI peripheral	csl_hpi.h	_HPI_SUPPORT	NO
IRQ	Interrupt controller	csl_irq.h	_IRQ_SUPPORT	NO
MCBSP	Multi-channel buffered serial port	csl_mcbsp.h	_MCBSP_SUPPORT	YES
PLL	PLL	csl_pll.h	_PLL_SUPPORT	YES
PWR	Power savings control	csl_pwr.h	_PWR_SUPPORT	NO
TIMER	Timer peripheral	csl_timer.h	_TIMER_SUPPORT	YES
UART	Universal Asynchronous Receiver/Transmitter	csl_uart.h	_UART_SUPPORT	YES
WDTIM	Watchdog Timer	csl_wdtim.h	_WDT_SUPPORT	YES

Table 1–2 lists the C54x devices that the CSL supports and the far and near-mode libraries included in the CSL. The device support symbol must be used with the compiler (–d option), for the correct peripheral configuration to be used in your code.

Note:

Devices C541 to C549 are NOT supported by CSL.

Table 1–2. CSL Device Support

Device	Near-Mode Library	Far-Mode Library	Device Support Symbol
C5401	csl5401.lib	csl5401x.lib	CHIP_5401
C5402	csl5402.lib	csl5402x.lib	CHIP_5402
C5404	csl5404.lib	csl5404x.lib	CHIP_5404
C5407	csl5407.lib	csl5407x.lib	CHIP_5407
C5409	csl5409.lib	csl5409x.lib	CHIP_5409
C5409A	csl5409A.lib	csl5409Ax.lib	CHIP_5409A
C5410	csl5410.lib	csl5410x.lib	CHIP_5410
C5410A	csl5410A.lib	csl5410Ax.lib	CHIP_5410A
C5416	csl5416.lib	csl5416x.lib	CHIP_5416
C5420	csl5420.lib	csl5420x.lib	CHIP_5420
C5421	csl5421.lib	csl5421x.lib	CHIP_5421
C5440	csl5440.lib	csl5440x.lib	CHIP_55440
C5441	csl5441.lib	csl5441x.lib	CHIP_5441
C5471	csl5471.lib	csl5471x.lib	CHIP_5471
C54cst	cslcst.lib	cslcstx.lib	CHIP_54CST

1.2 Naming Conventions

The following conventions are used when naming CSL functions, macros and data types.

Table 1–3. 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, macros and data types start with PER_ (where PER—all in capital letters—is the Peripheral module name listed in Table 1–1).
- Function names use all small letters. Capital letters are used only if the function name consists of two separate words (e.g., PER_getConfig()).
- Macro names use all capital letters (e.g., DMA_DMPREC_RMK).
- Data types begin with a capital letter, followed by small letters (e.g., DMA_Handle).

1.3 Data Types

The CSL provides its own set of data types. Table 1–4 lists the CSL data types as defined in the *stdinc*.h file.

Table 1–4. CSL Data Types

Data Type	Description
CSLBool	unsigned short
*PER*_Handle	void *
Int16	short
Int32	long
Uchar	unsigned char
Uint16	unsigned short
Uint32	unsigned long
DMA_AdrPtr	void (*DMA_AdrPtr)() pointer to a void function

1.4 Functions

Table 1–5 provides a generic description of the most common CSL functions where *PER* indicates a peripheral module as listed in Table 1–1.

Note:

Not all of the peripheral functions are available for all the modules. See the specific module chapter for specific module information. Also, each peripheral module may offer additional peripheral specific functions.

The following conventions are used in Table 1–5:

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

CSL functions provide a way to program peripherals by *direct register initialization* using the PER_config() or PER_configArgs() functions (see section 1.4.1). This method is used by the CSL GUI.

Table 1–5. 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 that is used 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*_**start**(*[handle,])* *[txrx,]* *[delay]* *)*	Starts the peripheral after using PER_config() or PER_configArgs(). [txrx] and [delay] apply only to MCBSP.
*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 generic 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** allows you to initialize a configuration structure with the appropriate register values and pass the address of that structure to the function, which then writes the values to the register. The CSL GUI uses this function to initialize peripherals. Example 1–1 shows an example of this method.
- ***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 1–1 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 offers you a GUI that produces these register values. CSL also provides the PER_REG_RMK (make) macros, which form merged values from a list of field arguments. Macros are covered in section 1.5, *CSL Macros*.

Example 1–1. Using PER_config or PER_configArgs

```
PER_Config MyConfig = {
  reg0,
  reg1,
  …
};
main() {
…
PER_config(&MyConfig);
…
;or...
;PER_configArgs (reg0, reg1, ...);
}
```

1.5 Macros

Table 1–6 provides a generic description of the most common CSL macros. The following naming conventions are used:

- *PER* indicates a peripheral module as listed in Table 1–1.
- *REG* indicates a register name (without the channel number).
- *REG#* indicates, if applicable, a register with the channel number. (e.g., DMPREC, DMSRC1, ...)
- *FIELD* indicates a field in a register.
- *regval* indicates an integer constant, an integer variable, a symbolic constant (*PER_REG*_DEFAULT), or a merged field value created with the *PER*_REG_RMK() macro.
- *fieldval* indicates an integer constant, integer variable, macro, or symbolic constant (*PER_REG_FIELD_SYMVAL*) as explained in section 1.6; all field values are right justified.

CSL also offers equivalent macros to those listed in Table 1–6, 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. The equivalent macros are shown in Table 1–7. Please note that REG is the register name without the channel number.

Table 1–6. Generic CSL Macros

Macro	Description
*PER*_REG_**RMK**(*fieldval_15,* . . . *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: ❑ Define only when more than one field exists in a register ❑ 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 you to initialize few fields in REG as an alternative to the _RMK macro that requires that ALL the fields in the register 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*_**ADDR**(*REG#*)	If applicable, retrieves the memory address (or sub-address) of the peripheral register REG#.

Table 1–7. Generic CSL Macros (Handle-based)

Macro	Description
*PER*_**RGETH**(handle, *REG*)	Returns the value of the peripheral register REG associated with Handle.
*PER*_**RSETH**(handle, *REG,* *regval*)	Writes the value to the peripheral register REG associated with Handle.
*PER*_**ADDRH**(handle, *REG*)	If applicable, retrieves the memory address (or sub-address) of the peripheral register REG associated with Handle.
*PER*_**FGETH**(handle, *REG,* *FIELD*)	Returns the value of the specified *FIELD* in the peripheral register REG associated with Handle.
*PER*_**FSETH**(handle, *REG*, *FIELD,* fieldval)	Sets the value of the specified *FIELD* in the peripheral register REG to fieldval.

1.6 Symbolic Constant Values

To initialize values in your application code, the CSL provides symbolic constants for registers and writable field values as described in Table 1–8. The following naming conventions are used:

- *PER* indicates a peripheral module as listed in Table 1–1.
- *REG* indicates a peripheral register.
- *FIELD* indicates a field in the register.
- *SYMVAL* indicates the symbolic value of a register field as listed in Appendix A.

Table 1–8. 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

PER_REG_FIELD_SYMVAL	Symbolic constant to specify values for individual fields in the indicated peripheral register. See Appendix A 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 and the Use of CSL Handles

The CSL provides limited support for resource management in applications that involve multiple threads, reusing the same multichannel peripheral device.

Resource management in the CSL is achieved through calls to the PER_open and PER_close functions. The PER_open function normally takes a channel/port number as the primary argument. It then returns a pointer to a Handle structure that contains information about which channel (DMA) or port (MCBSP) was opened.

When given a specific channel/port number, the open function checks a global flag to determine its availability. If the port/channel is available, it returns a pointer to a predefined Handle structure for this device.

If the device has already been opened by another process, an invalid Handle is returned with a value equal to the CSL symbolic constant, INV.

Calling PER_close frees a port/channel for use by other processes. PER_close clears the in_use flag and resets the port/channel.

Note:

All CSL modules that support multiple ports or channels, such as MCBSP, TIMER, DAT, and DMA, require a device Handle as primary argument to most functions. For these functions, the definition of a PER_Handle object is required.

1.7.1 Using CSL Handles

CSL Handle objects are used to uniquely identify an opened 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 argument.
For example:

```
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:

```
DMA_start(myDma);          /* Begin transfer */
.
.
.
DMA_close(myDma);           /* Free DMA channel */
```

1.8 Support for Device-Specific Features

Not all C54x peripherals offer the same type of features across the C54x devices. Table 1–9 lists specific features that are not common across the C54x family and the devices that support these features. References to Table 1–9 will be found across the CSL documentation.

Table 1–9. Device-Specific Features Support

(a) DMA Module-Channel Reload

Individual Channel Register Reload Support	Global Channel Register Reload Support
5416, 5421, 5409a, 5410a, 5440, 5441	All other C54x supported devices

(b) DMA Module-Extended Data Reach

Individual Channel Extended Data Memory Support	Global Extended Data Memory Support	No Extended Data Memory Support
5440, 5441	5409, 5416, 5421, 5409a, 5410a	5402, 5404, 5407, 5410, 5420, 5455

(c) MCBSP Module-Channel Support

MCBSP 128–Channel Support	MCBSP 32-Channel Support
5416, 5421, 5440, 5409a, 5410a, 5441	All other C54x supported devices

(d) Watchdog Module

Watchdog Timer Support	No Watchdog Timer Support
5440, 5441	All other C54x supported devices

(e) Timer Module

Timer Extended Pre-Scaler Support	No Timer Extended Pre-Scaler Support
5471, 5441	All other C54x supported devices

(f) Chip Module

Device ID Support	No Device ID Support
5416, 5409A, 5410A, 5441, 5421	All other C54x supported devices

Chapter 2

How To Use CSL

This chapter provides instructions on how to:

- Use the TMS320C54x CSL Graphic User Interface (GUI) integrated into the DSP/BIOS Configuration Tool. It describes a detailed work-flow for setting and accessing the CSL GUI data via the generation of C code files. The examples shown in this chapter include a complete procedure for creating configuration objects and generating peripheral pre-initialization.
- Use the CSL library without the use of the CSL GUI (see section 2.3).

Topic **Page**

2.1 Overview

There are two methods to configure peripherals when using CSL:

- ❏ Manually configure the peripherals by declaring/initializing objects and invoking the CSL functions inside your C source code.
- ❏ Configure peripherals by using the CSL GUI.

The CSL GUI, integrated under the DSP/BIOS configuration tool, provides the benefit of a visual tool that allows you to view the chosen register settings, and determine which flags/options have been set by a particular mode selection. With CSL GUI, code for the configuration settings is automatically created and stored in a C source file that is integrated directly into your application.

It is not possible to configure all peripherals via the CSL GUI. Only peripherals requiring initial static configurations are supported. Please refer to Table 1–1, on page 1-4, for the list of peripherals supported by the CSL GUI.

2.2 Using CSL with the CSL GUI

2.2.1 CSL GUI: The CSL Tree

The CSL GUI, integrated under the DSP/BIOS Configuration tool tree, allows you to configure some, but not all, of the on-chip peripherals. Each peripheral is represented as a subdirectory of the CSL Tree as shown in Figure 2–1.

Figure 2–1. CSL Tree (for 5402)

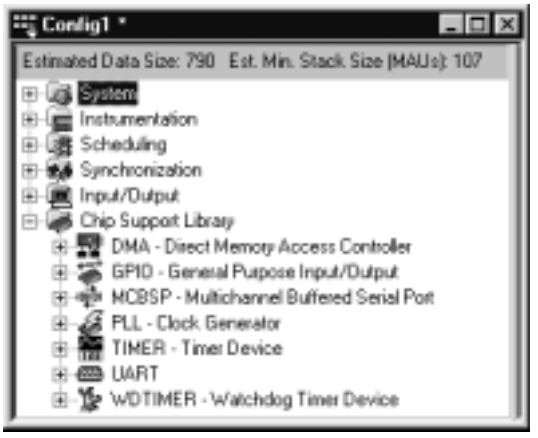
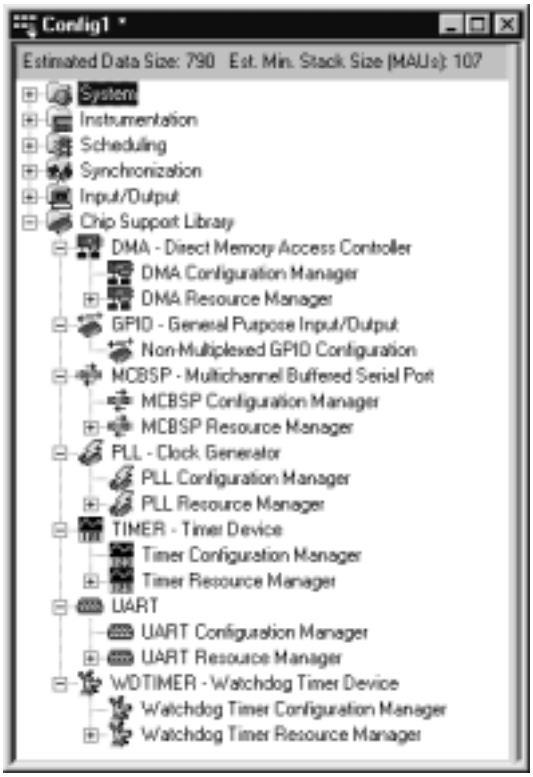


Figure 2–2 shows an example of the expanded CSL Tree.

Figure 2–2. Expanded CSL Tree (for C5402)

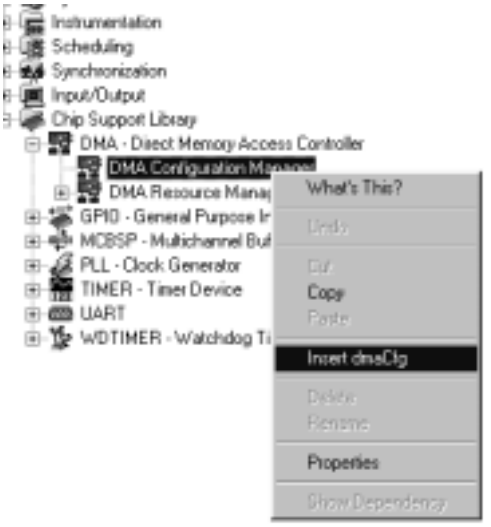


Each peripheral is organized into two sections (see Figure 2–2):

- *PERIPHERAL* **Configuration Manager** allows you to build and configure peripheral objects using the CSL PER_Config structure (see Table 1–5 on page 1-9). You can set the peripheral register values by selecting the options through the Properties pages. Several configuration objects can be created by selecting the Insert *PER*Cfg option from the right-click menu (see Figure 2–3). The menu options allow you to rename and delete the configuration object (see Figure 2–4), and to display the Dependency Dialog box that allows you to determine which peripheral is using the configuration (see Figure 2–5).

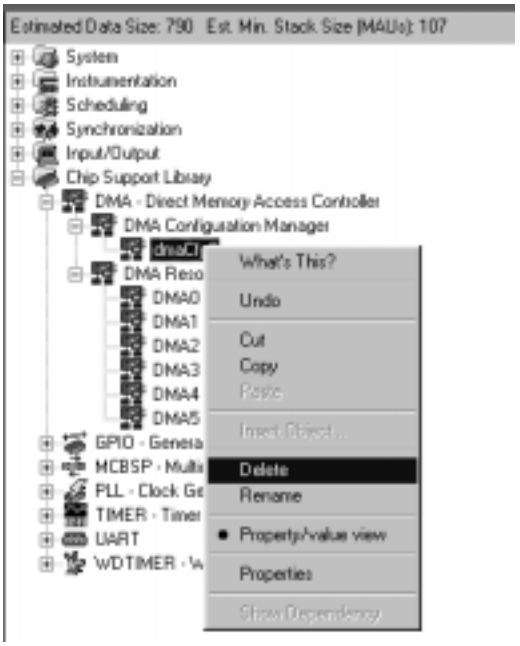
- *PERIPHERAL* **Resource Manager** allows you to bind a peripheral configuration to a specific on-chip device, i.e., a DMA channel, a MCBSP port , or a TIMER device. The devices are displayed as pre-defined objects and cannot be deleted or renamed. However, the Handles to these objects can be renamed.

Figure 2–3. Insert Configuration Object



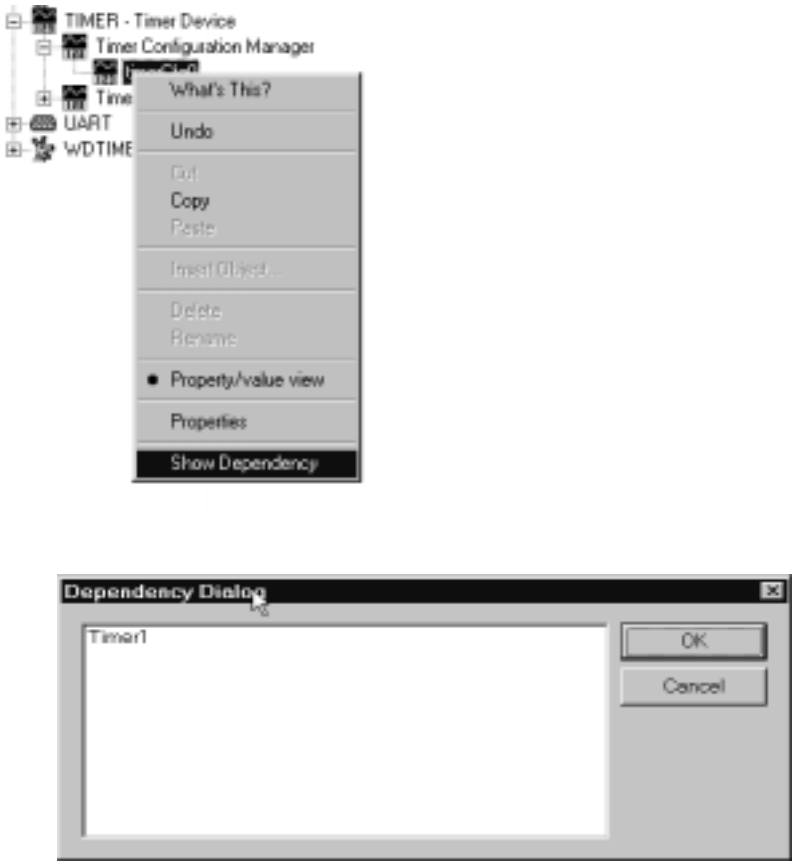
The Insert option allows the user to create the configuration objects. Once the object is created, you can access the Properties Page of this configuration, which allows you to set the register values.

Figure 2–4. Delete/Rename Options



The Delete option allows you to delete a configuration object. The Delete option is not accessible if the configuration is used by a physical peripheral of the associated Resource manager.

Figure 2–5. Show Dependency Option



The Show Dependency menu option shows which device/channel is using what specific configuration. Once a configuration has been bound to a peripheral, the Show Dependency option can be used to indicate which peripheral is using what particular configuration. In Figure 2–5, the “timerCfg0” configuration is bound to the physical timer device “Timer1” at design time (in the configuration tool).

Note:

A configuration object cannot be deleted if there is a dependency present.

2.2.2 Steps to configuring a peripheral using CSL GUI

This section provides an example using the 5402 device that demonstrates how to create, open and define a configuration for a TIMER device using the CSL GUI.

When configuring a peripheral using the CSL GUI, the steps below must be followed:

Step 1: Create the DSP/BIOS configuration file (.cdb file). In Code Composer Studio, select File → New → DSP/BIOS Configuration (select *template.cdb*). Default name, *Config1.cdb*.

Step 2: Configure the on-chip peripherals using the CSL hierarchy tree. (see section 2.2.3)

Step 3: Save the configuration file (.cdb file). Select File → Save as: enter the .cdb name; for example, *mytimer.cdb*. This automatically generates several files requested to build your project. (see section 2.2.4)

Step 4: Add the .cdb file to your project:

Click on Project → Add Files to Project. Select the mytimer.*cdb* file.

Step 5: Add the linker command file associated with the .cdb file:

Click on Project → Add Files to Project. Select the mytimer.*cmd* file.

Figure 2–6 shows the project layout before and Figure 2–7 shows the project layout after a .cdb file is created and the *mytimer*.cmd, *mytimer*.s54, and *mytimer_c*.c files have been added to the project.

Step 6: Modify your application code (main.c) (see section 2.2.5)

Figure 2–6. Code Composer Studio IDE Project Window

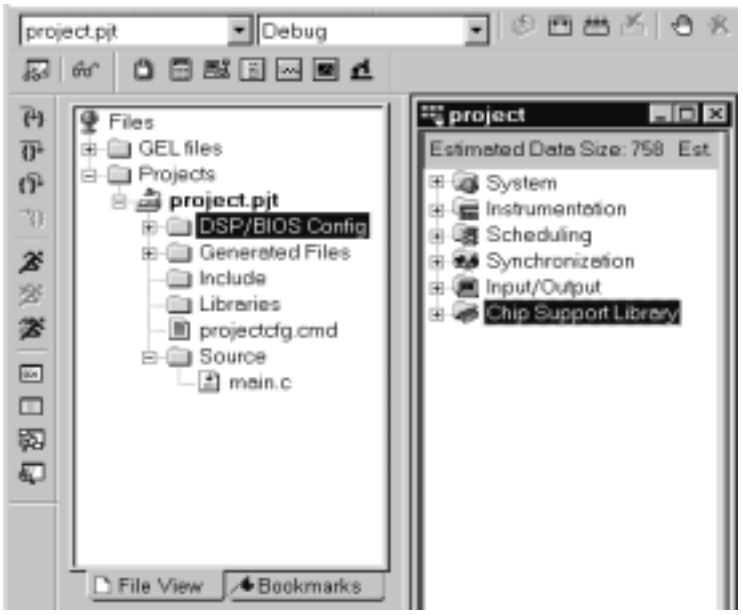


Figure 2–7. Code Composer Studio IDE Project Window with .cdb Project File Added

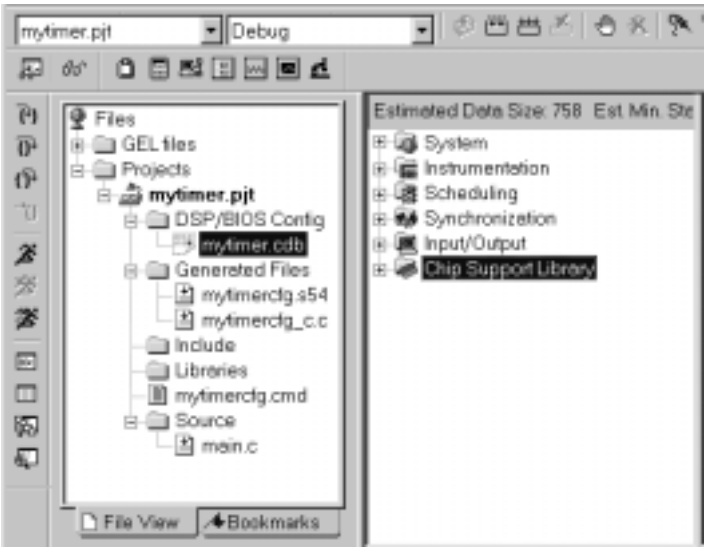
2.2.3 TIMER Configuration Example

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 ensures 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 2.2.2).

In the CCStudio Project View window (see Figure 2–8) open *mytimer*.cdb, and go to the sub-folder TIMER module (CSL Folder).

Figure 2–8. CCStudio IDE Project View



Using Figure 2–9 follow these steps:

Step 1: Right-click on the TIMER Configuration Manager, insert a new configuration object.

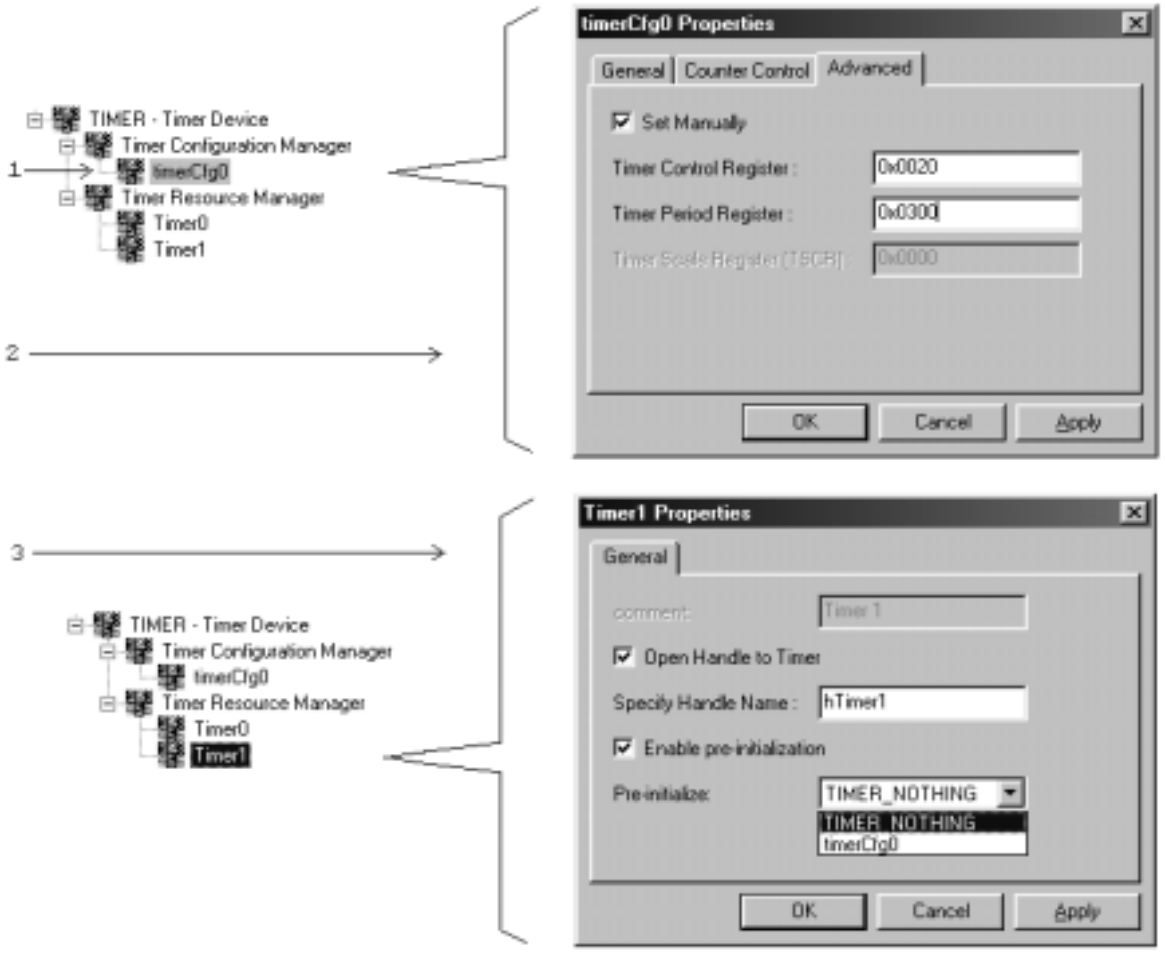
Step 2: Right-click on timerCfg0 and select Properties to open the timerCfg0 Properties window (as shown in Figure 2–9). Set the configuration by clicking on any of the tabs.

Step 3: Under the Timer Resource Manager, right-click on Timer1 and select Properties to open the Timer1 Properties window (see Figure 2–9).

- Check the Open Handle to Timer and Enable pre-initialization

- From the pre-initialize drop-down list, select the configuration, timerCfg0.

Figure 2–9. Configuring the TIMER1 Device



2.2.4 Files Generated by the CSL GUI

After saving the configuration file *mytimer***.cdb**, there are a number of files generated by the configuration tool:

- *mytimer*.**cdb** - An updated configuration database containing all inserted objects and all current property settings.
- *mytimer***cfg.h** - An extern declaration of all created objects, definition of CHIP_XXXX, plus #include of CSL module headers.
- *mytimer***cfg_c.c** - Definition and initialization of created objects. Contains function calls to TIMER_open and TIMER_config for pre-opened/pre-initialized objects.
- *mytimer***cfg.sXX** - Defines which DSP/BIOS elements are present and provides initialization for those elements.
- *mytimer***cfg.cmd** - A Linker command file. Includes the CSL library and .csldata placements.

Two of these files are dedicated to the CSL and use the CSL API:

- Header file: *mytimer***cfg.h**
- Source file: *mytimer***cfg_c.c**

In this example, mytimer is your .cdb file name. The bold characters are attached automatically.

2.2.4.1 Header File: mytimercfg.h

This 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.

This header file contains several elements:

- The definition of the chip:

  ```
  #define CHIP_5402 1
  ```

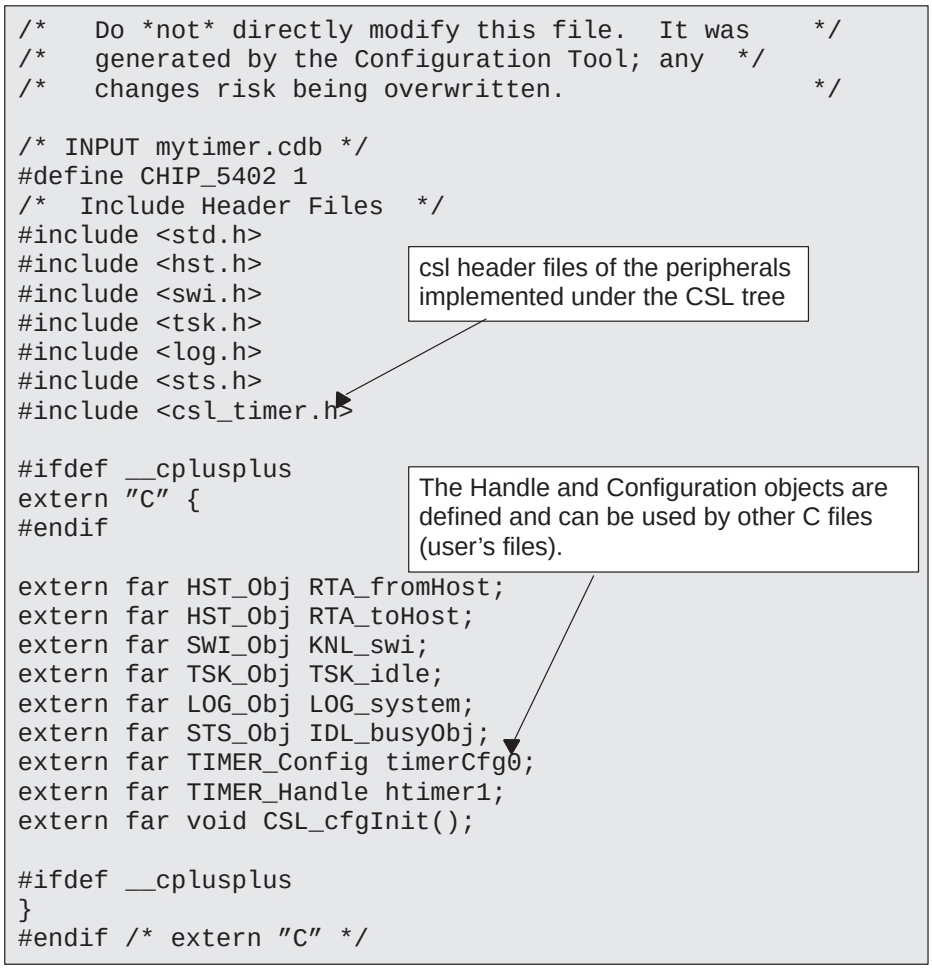
- The csl header files used by the CSL tree
- The declaration list of the CSL handle objects and configuration names defined in the *mytimer_c.c*

  These are declared external, and can be applied in the user's code:

  ```
  extern TIMER_Config timerCfg1;
  extern TIMER_Handle hTimer1;
  #include <projectcfg.h>
  ```

Figure 2–10 illustrates the setup of *mytimer***cfg_c.c**

Figure 2–10. Header File mytimercfg.h



2.2.4.2 Source File: mytimercfg_c.c

This source file consists of the Include section, the Declaration section, and the Code section:

- **Include** section

  This section defines the project header file. This allows *mytimer***cfg_c.c** access to the data declared in the header file.

  ```
  #include <mytimercfg.h>
  ```

Note:

This line is added before the other CSL header files (csl_emif.h, csl_timer.h, ...). This is done so that you are not required to specify the device number under the Project option (that –dCHIP_54xx is not required).

- **Declaration** section

  This section defines the peripheral registration configuration structures and the Handle objects previously defined in the CSL GUI.

- **Code** section

  The code section is composed of a unique function, **CSL_cfgInit()**, as shown in Figure 2–11, which is automatically called by the DSP/BIOS boot routine if the project.cdb is added to your project. Otherwise, you can call it later in your main.c file.

  CSL_cfgInit() prompts the tool to implement the open and configuration options you specify in the CSL GUI, via calls to the TIMER_open and TIMER_config functions.

  Calls to TIMER_open() and TIMER_config() are generated when the *Open Handle to Peripheral and Enable pre-initialization* options are checked in the Properties page of the related Resource Manager (see the timer example illustrated in Figure 2–11).

Note:

A device can be allocated/opened without being configured.

In the example shown in Figure 2–11,

- If Enable pre-initialization is checked, a call to the *TIMER_config()* function is generated.
- If Enable Pre-initialization is unchecked, a call to the*TIMER_config()* is not generated, but the configuration structure *timerCfg1* is created and available for you to use at a later time.

Figure 2–11. Resource Manager Properties Page

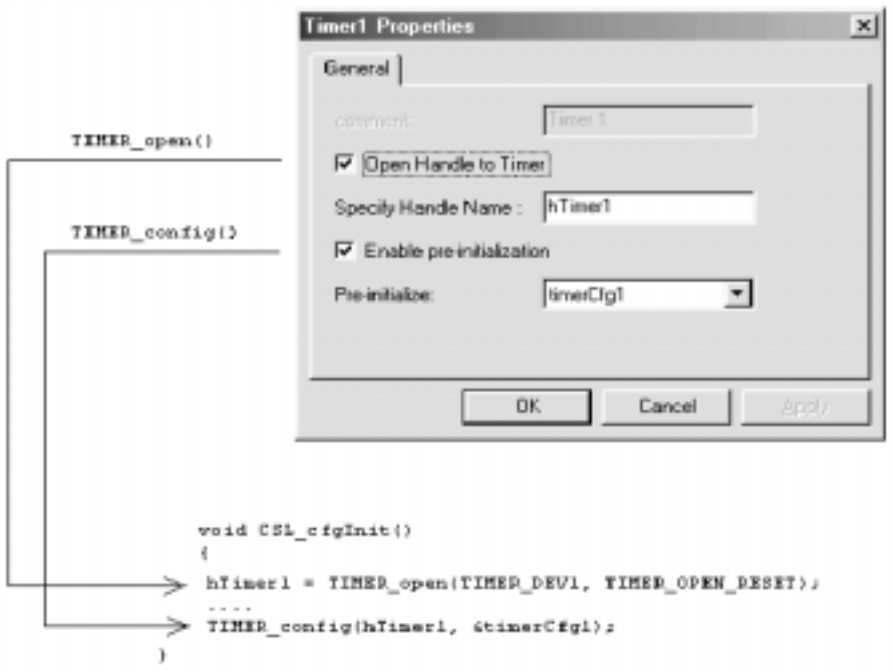
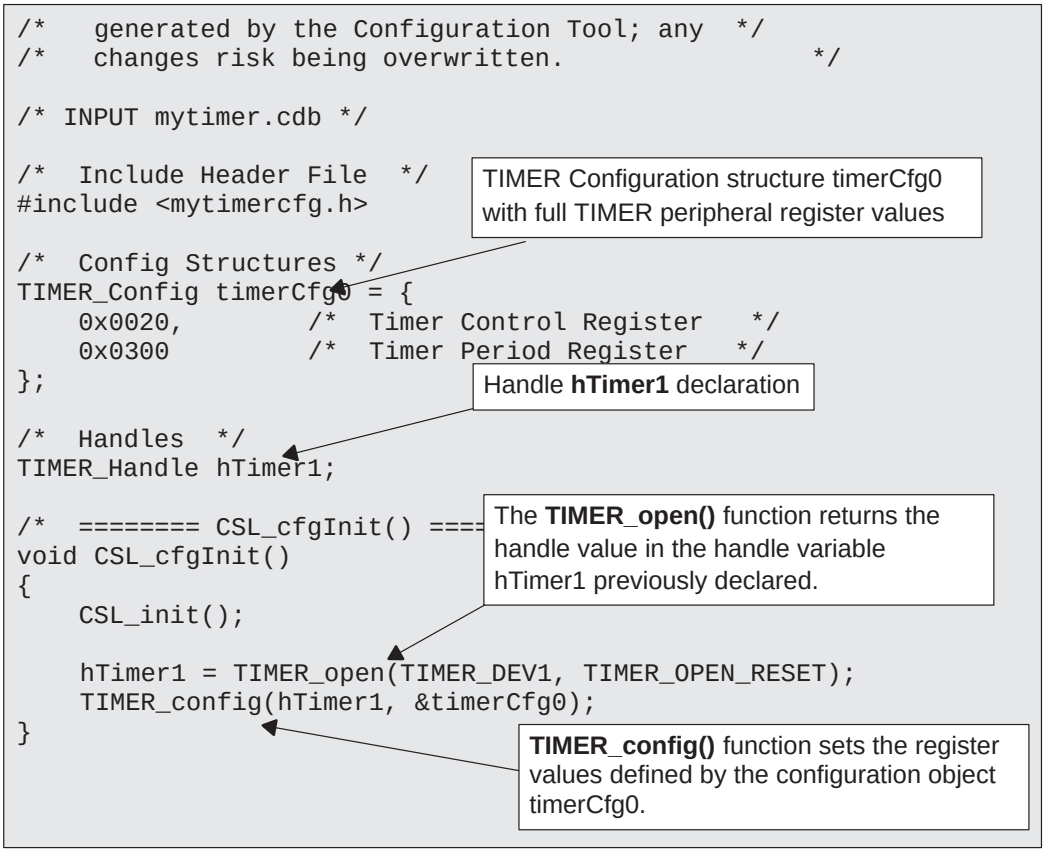


Figure 2–12 illustrates the mytimercfg_c.c file.

Figure 2–12. Source File mytimercfg_c.c



2.2.5 Modifying Your Application Code (main.c)

The following line of code is required to access the CSL objects you predefined with the CSL GUI:

```
#include  “mytimercfg.h”
```

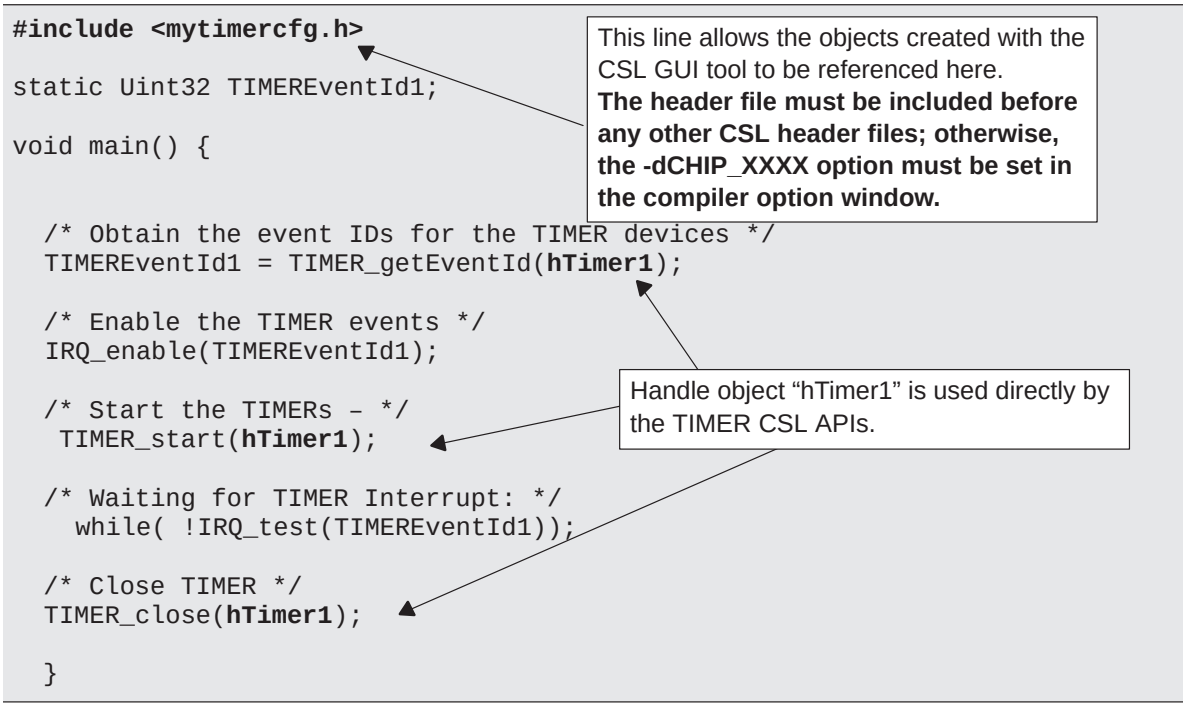
This includes CSL handle and configuration declarations to your code.

mytimercfg.h automatically includes the required *csl.h* and *csl_timer.h* files.

This header file must be included before other CSL files are added.

Figure 2–13 illustrates how to access the files generated by the CSL GUI in your application code (main.c)

Figure 2–13. Example of main.c File Using Data Generated by the CSL GUI



2.3 Using CSL without the CSL GUI

You may choose to manually declare and initialize the peripheral configuration objects within the C source. This means that you are not required to use the CSL GUI when working with the CSL library. However, it is highly recommended that you use the CSL GUI, particularly if your application involves interrupt management, and there are other components of DSP/BIOS present.

This section provides an example of using CSL without the CSL GUI and without a .cdb file. The CSL GUI merely assists in generation of the CSL configuration structures and provides the option to open and configure peripherals. This can equally be done using the CSL macros and functions directly.

There are two ways to program peripherals using CSL:

- **Register-based configuration (PER_config())**: Configures peripherals by setting the full values of memory-map registers. Compared to functional parameter-based configurations, register-based configurations require less cycles and code size, but are not abstracted.
- **Functional parameter-based configuration (PER_setup())**: Configures peripherals via a set of parameters. Compared to register-based configurations, functional parameter-based configurations require more cycles and code size, but are more abstracted.

The following example illustrates the use of CSL to initialize DMA channel 0 and to copy a table from address 0x3000 to address 0x2000 using the register based configuration (DMA_config())

Source address:	2000h in data space
Destination address:	3000h in data space
Transfer size:	Sixteen 16-bit single words

2.3.1 Using the DMA_config() function

The steps below use the DMA_config() function to initialize the registers:

Step 1: Include the csl.h and the header file of the module/peripheral you will use <csl_dma.h>. The different header files are shown in Table 1–1.

```
#include <csl.h>
#include <csl_dma.h>

// Example-specific initialization
#define N 16        // block size to transfer
#pragma DATA_SECTION(src,"table1")
/* scr data table address */
Uint16 src[N] = {
      0xBEEFu,  0xBEEFu,  0xBEEFu,  0xBEEFu,
      0xBEEFu,  0xBEEFu,  0xBEEFu,  0xBEEFu,
      0xBEEFu,  0xBEEFu,  0xBEEFu,  0xBEEFu,
      0xBEEFu,  0xBEEFu,  0xBEEFu,  0xBEEFu
};
#pragma DATA_SECTION(dst, "table2")
/* dst data table address */
Uint16 dst[N];
```

Step 2: Define and initialize the DMA channel configuration structure.

```
DMA_Config myconfig = {
    DMA_DMACSDP_RMK(0 , 0 ,0 ,0 , 0, 0 ,1),      /* DMACSDP  */
    DMA_DMACCR_RMK(1, 1, 0, 0, 0, 0,  0, 0, 0),  /* DMACCR   */
    DMA_DMACICR_RMK(1, 1,1 , 1, 1, 1),           /* DMACICR  */
    (DMA_AdrPtr) &src,                           /* DMACSSAL */
    0,                                           /* DMACSSAU */
    (DMA_AdrPtr)&dst,                            /* DMACDSAL */
    0,                                           /* DMACDSAU */
    N,                                           /* DMACEN   */
    1,                                           /* DMACFN   */
    0,                                           /* DMACFI   */
    0};                                          /* DMACEI   */
```

Step 3: Define a DMA_Handle pointer. DMA_open will initialize this handle when a DMA channel is opened.

```
DMA_Handle myhDma;
void main(void) {
// .....
```

Step 4: Initialize the CSL Library. A one-time only initialization of the CSL library must be done before calling any CSL module API:

```
CSL_init();   /* Init CSL */
```

Step 5: For multi-resource peripherals such as McBSP and DMA, call PER_open to reserve resources (MCBSP_open(), DMA_open()...):

```
myhDma = DMA_open(DMA_CHA0, 0);/* Open DMA Channel 0 */
```

By default, the TMS320C54xx compiler assigns all data symbols word addresses. The DMA however, expects all addresses to be byte addresses. Therefore, you must shift the address by 2 in order to change the word address to a byte address for the DMA transfer.

Step 6: Configure the DMA channel by calling DMA_config() function:

```
myconfig.dmacssal =
(DMA_AdrPtr)(((Uint16)(myconfig.dmacssal)<<1)&0xFFFF);
myconfig.dmacdsal =
(   DMA_AdrPtr)(((Uint16)(myconfig.dmacdsal)<<1)&0xFFFF);
DMA_config(myhDma, &myConfig);    /* Configure Channel  */
```

Step 7: Call DMA_start() to begin DMA transfers:

```
DMA_start(myhDma);  /*  Begin Transfer  */
```

Step 8: Close DMA channel

```
DMA_close(myhDma);   /* Close channel (Optional) */
}
```

2.3.2 Compiling and Linking With CSL using Code Composer Studio (CCStudio)

To compile and link with CSL, you must configure the CCStudio project environment. To complete this process, follow these steps:

Step 1: Specify the target device. (Refer to section 2.3.2.1)

Step 2: Determine whether or not you are using small or large memory model and specify the CSL and RTS libraries you require. (Refer to section 2.3.2.3)

Step 3: Create the linker command file (with a special .csldata section) and add the file to the project. (Refer to section 2.3.2.4)

Step 4: Determine if you must enable inlining. (Refer to section 2.3.2.5)

The remaining sections in this chapter will provide more details and explanations for the steps above.

Note:

CCStudio will automatically define the search paths for include files and libraries as defined in Table 2–1. You are not required to set the –i option.

Table 2–1. CSL Directory Structure

This CSL component...	Is located in this directory...
Libraries	c:\ti\c5400\bios\lib
Source Library	c:\ti\c5400\bios\lib
Include files	c:\ti\c5400\bios\include
Examples	c:\ti\examples\<target>\csl
Documentation	c:\ti\docs

2.3.2.1 Specifying Your Target Device

Use the following steps to specify the target device you are configuring:

Step 1: In Code Composer Studio, select Project → Options.

Step 2: In the Build Options dialog box, select the Compiler tab (see Figure 2–14).

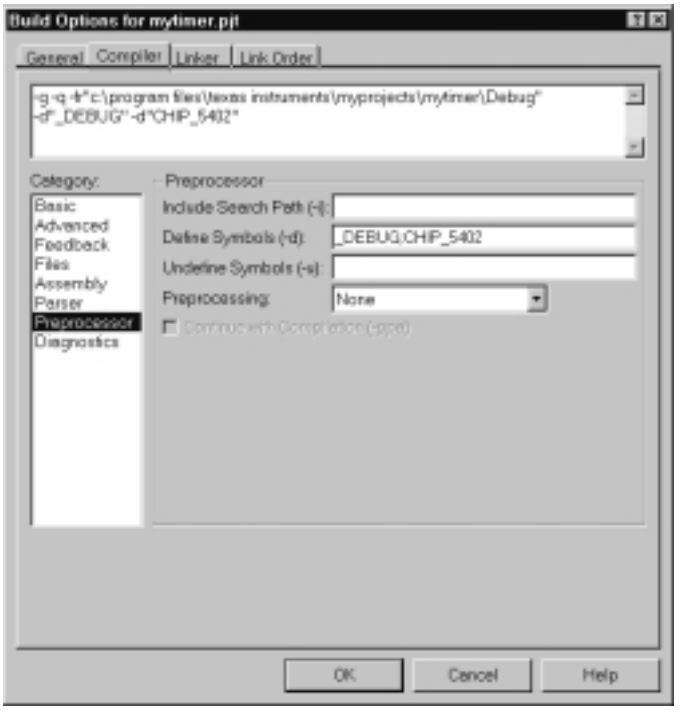
Step 3: In the Category list box, highlight Preprocessor.

Step 4: In the Define Symbols field, enter one of the device support symbols in Table 1–2, on page 1-5.

For example, if you are using the 5402PG1.2 device, enter CHIP_5402PG1_2.

Step 5: Click OK.

Figure 2–14. Defining the Target Device in the Build Options Dialog



2.3.2.2 Selecting Far/Near Mode

If you use an far-mode libraries, define far-mode for the compiler, and link with the far-mode library (rts_ext.lib), use the following steps. Use Figure 2–15 to complete these steps.

Step 1: In Code Composer Studio, select Project → Options

Step 2: In the Build Options dialog box, select the Compiler Tab.

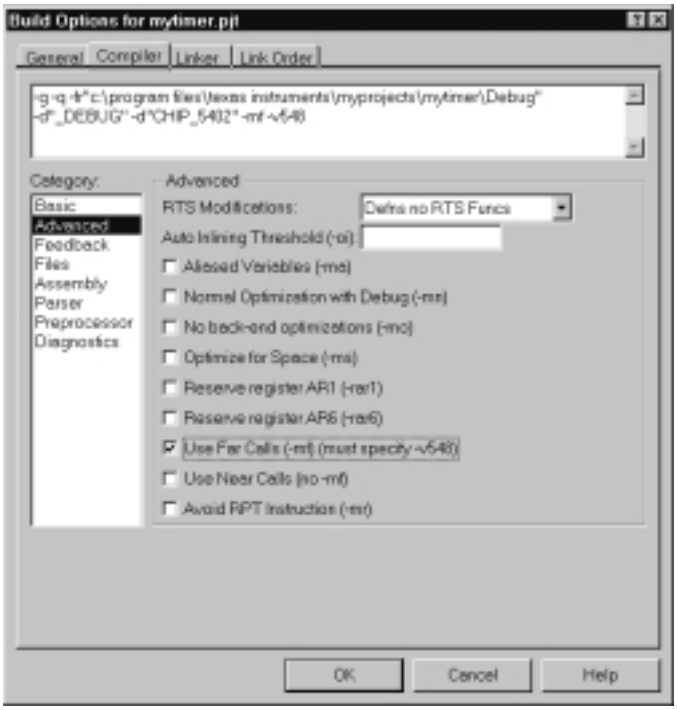
Step 3: In the Category list box, highlight advanced.

Step 4: Select Use Far Calls.

Step 5: In the Processor Version (-v) field, type 548.

Step 6: Click OK.

Figure 2–15. Defining Far Mode



Step 7: If you use any far-mode libraries, define far mode for the compiler and link with the far mode runtime library (rts_ext.lib). Then, you must specify which CSL and RTS libraries will be linked in your project.

Step 8: In Code Composer Studio, select Project → Options

Step 9: In the Build Options dialog box, Select the Linker Tab (see Figure 2–17).

Step 10: In the Category list, highlight Basic.

The Library search Path field (-l), should show:
c:\ti\c5400\bios\lib (automatically configured by CCS)

Step 11: In the Include Libraries (-l) field, enter the correct library from Table 1–2, on page 1-5.

For example, if you are using the 5402 device, enter csl5402.lib for near mode or csl5402x.lib for far mode. In addition, you must include the corresponding rts.lib or rts_ext.lib compiler runtime support libraries.

Step 12: Click OK.

2.3.2.3 Large/Small Memory Model Selection

If you use any large memory model libraries, define the -ml option for the compiler and link with the large memory model runtime library (rts54x.lib) using the following steps:

Step 1: In Code Composer Studio, select Project → Options.

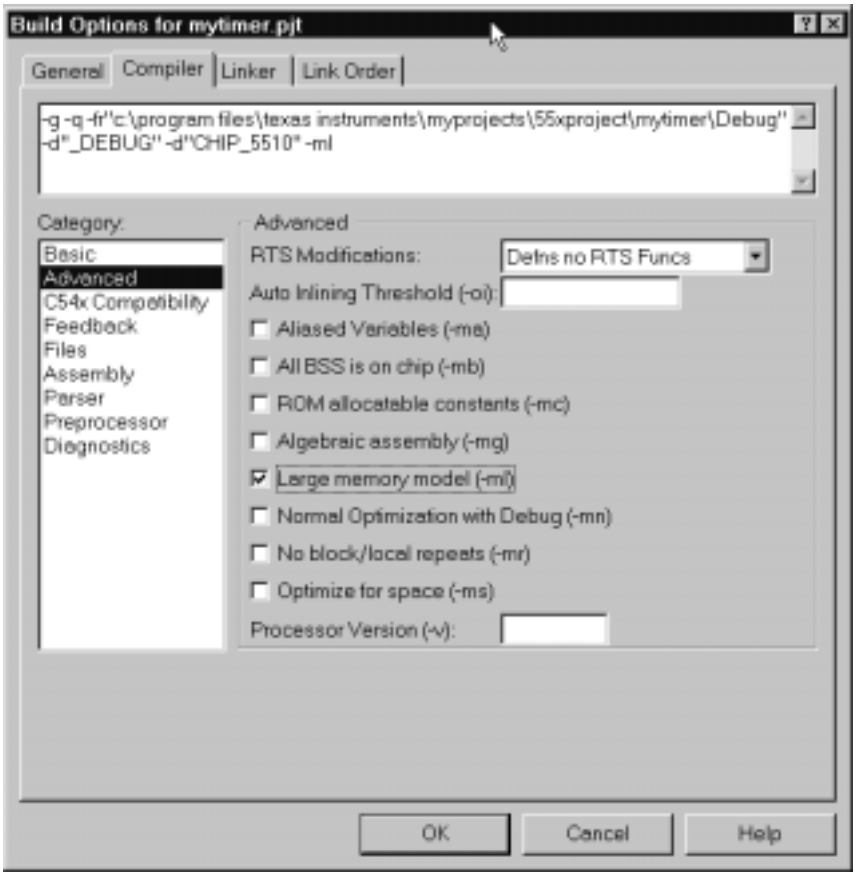
Step 2: In the Build Options dialog box, select the Compiler Tab (Figure 2–16).

Step 3: In the Category list box, highlight advanced.

Step 4: Select Use Large memory model (-ml).

Step 5: Click OK.

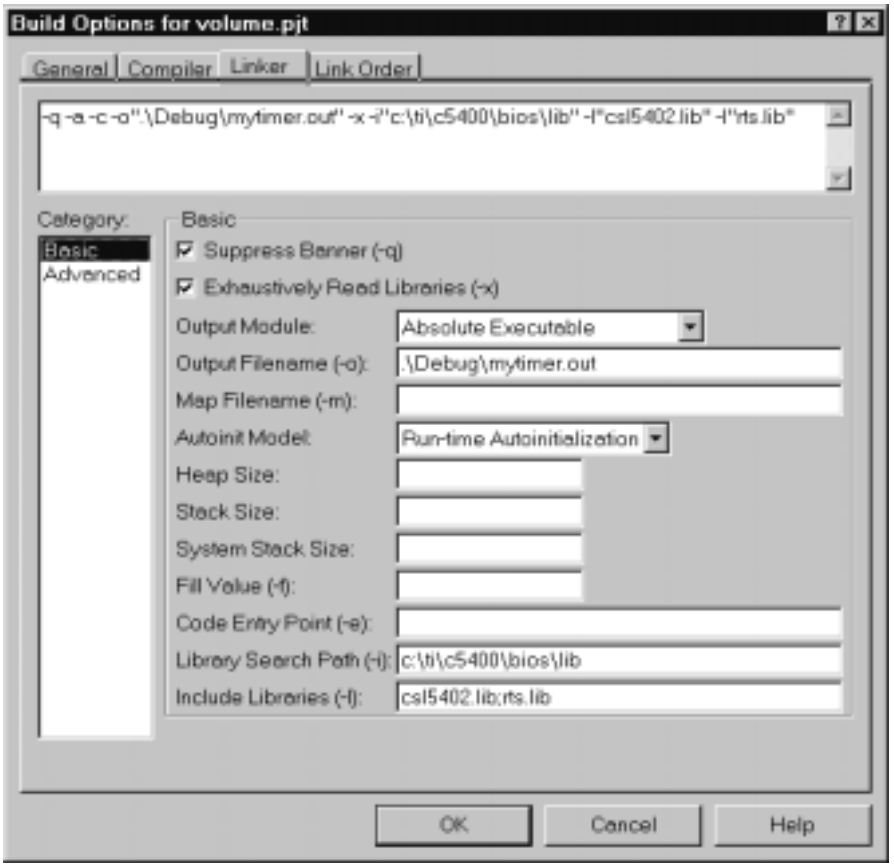
Figure 2–16. Defining Large Memory Model



Then, you must specify which CSL and RTS libraries will be linked in your project.

- ❑ In Code Composer Studio, select Project → Options
- ❑ In the Build Options dialog box, Select the Linker Tab (see Figure 2–17).
- ❑ In the Category list, highlight Basic.
- ❑ The Library search Path field (-I), should show: c:\ti\c5400\bios\lib (automatically configured by CCStudio)
- ❑ In the Include Libraries (-l) field, enter the correct library from Table 1–2, on page 1-5.
- ❑ For example, if you are using the 5402 device, enter csl5402.lib for near mode or csl5402x.lib for far mode. In addition, you must include the corresponding rts54.lib or rts54x.lib compiler runtime support libraries.
- ❑ Click OK.

Figure 2–17. Defining Library Paths



2.3.2.4 Creating a Linker Command File

The CSL has two requirements for the linker command file:

- **You must allocate the .csldata section.**

  CSL creates a .csl data section to maintain global data that CSL uses to implement functions with configurable data. You must allocate this section within the base 64K address space of the data space.

- **You must reserve address 0x7b in scratch pad memory**

  The CSL uses address 0x7b in the data space as a pointer to the .csldata section, which is initialized during the execution of *CSL_init()*. For this reason, you must call *CSL_init()* before calling any other CSL functions. Overwriting memory location 0x7b can cause the CSL functions to fail.

Example 2–1 illustrates these requirements which must be included in the linker command file.

Example 2–1. Using a Linker Command File

```
MEMORY
{
       PROG0:    origin =  8000h, length = 0D000h
       PROG1:    origin = 18000h, length = 08000h

       DATA:    origin =  1000h, length = 04000h
}

SECTIONS
{
   .text    > PROG0
   .cinit   > PROG0
   .switch  > PROG0
   .data    > DATA
   .bss     > DATA
   .const   > DATA
   .sysmem  > DATA
   .stack   > DATA
   .csldata > DATA

   table1 : load =  6000h
   table2 : load =  4000h
}
```

2.3.2.5 Using Function Inlining

Because some CSL functions are short (they may set only a single bit field), incurring the overhead of a C function call is not always necessary. If you enable inline, the CSL declares these functions as *static inline*. Using this technique helps you improve code performance.

2.4 Rebuilding CSL

All CSL source code is archived in the file csl54xx.src located in the \C5400\bios\lib\ folder. For example, to rebuild csl5402x.lib, type the following on the command line:

```
mk500  csl54xx.src -dCHIP_5402 -v548 -mf
```

Chapter 3

CHIP Module

The CSL CHIP module offers general CPU functions and macros for C54x register accesses. The CHIP module is not handle-based.

Topic **Page**

3.1 Overview

The CSL CHIP module offers general CPU functions. The CHIP module is not handle-based.

Table 3–1 lists the functions available as part of the CHIP module.

Table 3–1. CHIP Functions

Function	Purpose	See page ...
CHIP_getCpuId	Returns the CPU ID field of the CSIDR register.	3-3
CHIP_getMapMode	Returns the current MAP mode of the device.	3-3
CHIP_getRevId	Returns the CPU revision ID.	3-4
CHIP_getSubsysId	Returns sub-system ID (or core) for a multi-core device.	3-4

3.2 Functions

This section lists the functions in the CHIP module.

CHIP_getCpuId — *Get CPU ID (C5416, C5421, C5440, C5441 only)*

Function Uint32 CHIP_getCpuId();

Arguments None

Return Value CPU ID — Returns the CPU (CHIP) ID Field

Description This function returns the CPU (CHIP) ID field of the CSIDR register.

Example

```
Uint32 CpuId;
CpuId = CHIP_getCpuId();
```

CHIP_getMapMode — *Reads the map mode bits*

Function Uint16 CHIP_getMapMode();

Arguments None

Return Value map mode — Returns current device MAP mode, which will be one of the following:

- ❏ CHIP_MAP_0: MP/MC DROM and OVLY bits are OFF
- ❏ CHIP_MAP_1: DROM bit is on
- ❏ CHIP_MAP_2: OVLY bit is on
- ❏ CHIP_MAP_3: Both DROM and OVLY Bits are on
- ❏ CHIP_MAP_4: MP/MC bit is on
- ❏ CHIP_MAP_5: MP/MC and DROM are on
- ❏ CHIP_MAP_6: MP/MC and OVLY bits are on
- ❏ CHIP_MAP_7: MP/MC, DROM, and OVLY bits are on

Description Reads the map mode bits (OVLY, DROM, MPMC) from the device. In devices not supported by a specific map-mode bit, the value returned is invalid. See the specific device data sheet for the availability of map mode bits. This function is useful for debugging purposes.

Example

```
Uint16 MapMode;
…
MapMode = CHIP_getMapMode();
if (MapMode == CHIP_MAP_0) {
  /* do map 0 tasks /
} else {
  /* do map 1 tasks */
}
```

CHIP_getRevId *Get revision ID (C5416, C5421 only)*

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 CSIDR register.

Example

```
Uint32 RevId;
RevId = CHIP_getRevId();
```

CHIP_getSubsysId *Get subsystem ID (C5421, C5440, C5441 only)*

Function Uint32 CHIP_getSubsysId();

Arguments None

Return Value Subsytem ID

Description Get the sub-system ID (or core) from a multi-core device from the CSIDR register.

Example

```
Uint32 RevId;
RevId = CHIP_getSubsysId();
```

Chapter 4

DAA Module

This chapter contains general descriptions for the DAA (Data Access Arrangement) configuration structures, functions, and macros.

Topic **Page**

4.1 Overview

The on-chip DAA (Data Access Arrangement) is the system (digital) side of a two-chip integrated DAA solution that provides a programmable line interface to meet global telephone line requirements. The DAA has built in A/D and D/A converters for telephone line data and these converters interface to the DSP core through a dedicated DSP serial port (McBSP #2).

Programming of the DAA registers is implemented through the same serial port. This eliminates the need for an isolation transformer, relays, opto–isolators, and a 2 to 4-wire hybrid, and reduces the cost of discrete components required to achieve compliance with global regulatory requirements.

Note:

This DAA chip requires the line (analog) side DAA (Silicon Labs Si3016) for operation.

4.1.1Configuration Structures

Table 4–1. DAA Configuration Structures

Configuration Structure	Purpose	See page ...
DAA_CircBufCtrl	Circular buffer control structure	4-4
DAA_Params	Parameters structure	4-4
DAA_PrivateObject/DAA/ DAA_Handle	Private object/Handle structure	4-5
DAA_DevSetup	Device setup structure	4-6
DAA_Setup	DAA setup structure	4-6

4.1.2 Functions

Table 4–2. DAA Functions

Functions	Purpose	See page ...
DAA_setup	Sets up DAA devices	4-7
DAA_close	Stops a DAA function	4-8
DAA_reset	Hardware reset function for the on-chip DAA	4-8
DAA_readWrite	DAA read/write function	4-9
DAA_availability	Returns available words in the circular buffer	4-9
DAA_resetInternalBuffer	Resets the pointers of the internal circular buffer	4-9
DAA_delay	Implements a delay count of McBSP frame syncs	4-10
DAA_isr	DAA interrupt service routine	4-10

4.1.3 Macros

Table 4–3. DAA Macros

Macros	Purpose	See page ...
DAA_RGETH	Macro to read DAA register	4-11
DAA_RSETH	Macro to write a value to a DAA register	4-11
DAA_ADDR	Returns a DAA register address	4-12
DAA_FMK	Returns a 16-bit register value for **REG** with **FIELD** set to **VAL**	4-12

4.2 Configuration Structures

(Example of structures are included in the Functions section)

DAA_CircBufCtrl *DAA Circular Buffer Control Structure*

Members

```
Int16  *pBuffer     /* pointer to the beginning of the circular buffer */
Uint16 bufSize      /* size of the circular buffer */
Int16  *pHead       /* pointer for reading data from Rx Interrupt */
Int16  *pTail       /* pointer for writing data from Tx Interrupt */
Int16  *pCurrent    /* pointer for reading/writing data for the user */
Uint16 isOverflow   /* private flag for overflow */
Uint16 circBufOffset /* initial circular buffer offset */
```

Description The user must create this circular buffer control structure, and include it in the DAA private object structure (DAA_PrivateObject). The structure is initialized by the DAA_setup(…) function. One circular buffer control structure must be created for each DAA device.

DAA_Params *DAA Parameters Structure*

Members

```
Uint16 txAttenuation       /* analog transmit attenuation, valid attenuation
                           /* values are: */
DAA_GCR_ATX_ATT_0db
DAA_GCR_ATX_ATT_3db
DAA_GCR_ATX_ATT_6db
DAA_GCR_ATX_ATT_9db
DAA_GCR_ATX_ATT_12db
Uint16 rxGain              /* cid receive attenuation/analog receive gain,
                           /* valid */
                           /* attenuation/gain values are: */
DAA_GCR_ARX_ATT_0db
DAA_GCR_ARX_ATT_1db
DAA_GCR_ARX_ATT_2.2db
DAA_GCR_ARX_ATT_3.5db
DAA_GCR_ARX_ATT_5db
DAA_GCR_ARX_GAIN_0db
DAA_GCR_ARX_GAIN_3db
DAA_GCR_ARX_GAIN_6db
DAA_GCR_ARX_GAIN_9db
DAA_GCR_ARX_GAIN_12db
Uint16 sampleRateReg7      /* sample rate control – register 7,
                           /* valid values are: */
DAA_SRCTRL_SRC_7200
```

```
DAA_SRCTRL_SRC_8000
DAA_SRCTRL_SRC_8229
DAA_SRCTRL_SRC_8400
DAA_SRCTRL_SRC_9000
DAA_SRCTRL_SRC_9600
DAA_SRCTRL_SRC_10286
Uint16 sampleRateReg8      /* sample rate control – register 8 */
Uint16 sampleRateReg9      /* sample rate control – register 9 */
Uint16 sampleRateReg10     /* sample rate control – register 10*/
Uint16 ictrl1              /* international control register 1 value */
Uint16 ictrl2              /* international control register 2 value */
Uint16 ictrl3              /* international control register 3 value */
                           /* pre-defined country-specific macros for the three */
                           /* international control registers are: */
DAA_AUSTRALIA
DAA_BULGARIA
DAA_CHINA
DAA_CTR21 (Europe)
DAA_ CZECH_REPUBLIC
DAA_FCC (same as DAA_USA)
DAA_ HUNGARY
DAA_JAPAN
DAA_ MALAYSIA
DAA_ NEW_ZEALAND
DAA_ PHILIPPINES
DAA_ POLAND
DAA_ SINGAPORE
DAA_ SLOVAKIA
DAA_SLOVENIA
DAA_SOUTH_AFRICA
DAA_SOUTH_KOREA
DAA_USA
```

Description The user must create this DAA parameters structure, initialize all members, and include it in the DAA single device setup structure (DAA_DevSetup). One parameter structure may be used for multiple DAA devices.

DAA_PrivateObject/DAA_Handle *Private Object/Handle Structure*

Members

```
MCBSP_Handle mcbspHandle       /* McBSP 2 Handle */
Int16  state                   /* current control and read/write state */
Uint16 regRequest              /* DAA register request – read or write */
CSLBool      isIORegReady      /* Flag to indicate register IO complete */
Uint16 regValue                /* Value of DAA register value just read or to
                               /* be written */
Uint16 regIndex                /* Index of DAA register value just read or to
                               /* be written */
Uint16 buffLength              /* buffer length for DAA read/write function */
```

```
Int16  delayCount                /* counter used to implement delay based on
                                 /* sampling rate */
Void   *pID                      /* void pointer to channel ID */
DAA_CallBack  dataCallBack       /* DAA data callback function pointer */
DAA_CallBack  ctrlCallBack       /* DAA control callback function pointer */
DAA_CircBufCtrl daaCircBufCtrl   /* circular buffer ctrl structure */
```

Description The user must create this private object/handle structure, initialize all members, and include it in the DAA single device setup structure (DAA_DevSetup). One private object structure must be created for each DAA device. This private object/handle serves as the state machine for the DAA device.

DAA_DevSetup *DAA Single Device Setup Structure*

Members

```
DAA_Params    *params       /* DAA parameters (initial register values) */
DAA_Handle    daaHandle     /* pointer to DAA private object created by user */
Uint16        mcbspPort     /* MCBSP port the DAA is connected to */
                            /* MCBSP_PORT2 (for internal DAA) */
Int16         *pCircBuf     /* pointer to the user-allocated circular buffer */
Uint16        circBufSize   /* circular buffer size */
Uint16        circBufOffset /* Initial circular buffer offset */
Uint16        dataLength    /* Length of data buffer (callback size) */
Void          *pID          /* void pointer to device ID */
DAA_CallBack dataCallBack   /* pointer to user-defined data callback function */
DAA_CallBack ctrlCallBack   /* pointer to user-defined control callback function */
DAA_RstFxn    reset         /* pointer to DAA hardware reset control function */
                            /* for internal DAA, use &DAA_reset */
```

Description The user must create this single device setup structure, initialize all members, and include it in the DAA setup structure (DAA_Setup). One device structure must be created for each DAA device. The DAA setup structure is passed to the function DAA_setup(..).

DAA_Setup *DAA Setup Structure*

Members

```
Uint16        numDevs       /* number of devices to be set up, must be greater
                            /* that 0 */
DAA_DevSetup  **dev         /* pointer to array of device setup structure
                            /* pointers */
```

Description The user must create DAA setup structure, initialize all members, and pass it to the DAA setup function, DAA_setup(..).

4.3 Functions

void DAA_setup *Pointer to DAA setup structure*

Arguments DAA_Setup *

Return Value None

Description This function sets up a number of DAA devices according to the value of numDevs in the setup structure and initializes the state machines. The function enables the IMR fields for the DAA interrupts. However, it is the responsibility of the user to plug the corresponding DAA ISRs into the vector table. Each of these user ISRs must call the function DAA_isr(…). Note that this function takes a pointer to a DAA private object as it's argument. This function makes calls to the CSL MCBSP module.

Important: The DAA_DevSetup structure passed through the DAA_Setup structure to this function defines a data callback length.As soon as interrupts are globally enabled, following a call to this function, the state machine starts invoking data callbacks whenever it has the specified number of words in the circular buffer.

Example

```
DAA_DevSetup setupStruct ={
&daaParams,
&daaPrivateObject,
MCBSP_PORT2,
circBuff,
40,
5,
10,
(void *)0,
(DAA_CallBack)&dataCB,
(DAA_CallBack)&ctrlCB,
&DAA_reset
};
DAA_DevSetup *devArray[] = {&devSetup};
DAA_Setup setup = {
1,
devArray
```

```
};
   DAA_setup(&setup);
void dataCB(void* pID, Arg task, Uns result)
{
/* submit next readwrite request */
DAA_readWrite(handle, dataBuff, 10);
/* Post buffer processing function */
}
```

DAA_close — *Handle to DAA device*

Arguments DAA_Handle

Return Value None

Description This function stops the DAA device operation. The function also closes the corresponding McBSP port.

Example

```
DAA_Handle handle;
DAA_close(handle);
```

DAA_reset — *DAA flag to select hardware*

Arguments Uint16 Flag to select hardware ***reset (1) /unreset (0)***.

Return Value None

Description Flag to select hardware ***reset (1) /unreset (0)***. It is used during the setup operation (DAA_setup(…)). This address of this function should be passed to the setup function for the on-chip DAA.

Example See example in 2.0.

DAA_readWrite

Copies size words from the data read/write buffer

Arguments

DAA_Handle	Handle to DAA device
Int16 *	Pointer to data read/write buffer
Uint16	Size of data read/write buffer

Return Value

Int16 Number of received words in buffer. Zero is returned if there is an error in dataBuff or size arguments.

Description

This function copies ***size*** words from the data read/write buffer into the internal circular buffer for transmission and fills the buffer with ***size*** received data from the circular buffer. Note: This function does not check for how much data is in the circular buffer. Therefore, the user must call DAA_availability to determine how much data is available in the circular buffer before calling the function.

Example

```
DAA_Handle handle;
Int16 buff[20];
Uint16 size = 20;
DAA_readWrite(handle, buff, size);
```

DAA_availabilty

Returns available words in the circular buffer

Arguments

DAA_Handle Handle to DAA device

Return Value

Int16 Number of available words in circular buffer.

Description

This function returns the number of available (received) words in the circular buffer.

Example

```
DAA_Handle handle;
Int16 bufCount;
BufCount = DAA_availability(handle);
```

DAA_resetInternalBuffer

Resets the pointers of the internal circular buffer

Arguments

DAA_Handle Handle to DAA device

Return Value

None.

Description

This function resets the pointers of the internal circular buffer. This action is equivalent to clearing out the buffer.

Example

```
DAA_Handle handle;
DAA_resetInternalBuffer(handle);
```

DAA_delay

Implements a delay of count McBSP frame syncs

Arguments

DAA_Handle	Handle to DAA device
Uint16	Number of McBSP frame-syncs to delay for

Return Value

CSLBool — TRUE Indicates “Accepted” and FALSE “rejected”.

Description

This function requests a delay of length count McBSP frame syncs. The function returns immediately with an indication of whether the request was accepted (TRUE) or rejected (FALSE). If accepted, the user defined control callback will be invoked when the delay is done with a flag of“_DAA_DELAY”. If not, the user must keep invoking the function until it is accepted.

Example

```
   Void main(void)
{
      DAA_Handle handle;
   Uint16 delay;
While(!DAA_delay(handle, delay));
…
…
   }
      void ctrlCB(void* pID, Uint16 task, Uint16 arg)
   {
         if (task & _DAA_DELAY)
         {
            printf(“delay is done!\n”);
      }
      }
```

DAA_isr

DAA interrupt service routine

Arguments

DAA_Handle	Handle to DAA device

Return Value

None.

Description

This is the DAA state machine and must be invoked on every DAA receive interrupt.

Example

```
DAA_Handle handle;
   Interrupt void myDAAIsr(void)
   {
      DAA_isr(handle);
   }
```

4.4 Macros

DAA_RGETH

Macros used to read a DAA register

Arguments

DAA_Handle	Handle to DAA device
Reg	DAA register name

Return Type CSLBool — TRUE Indicates “Accepted” and FALSE “rejected”.

Description This macro reads a DAA register. The macro returns immediately with an indication of whether the request was accepted (TRUE) or rejected (FALSE). If accepted, the user defined control callback will be invoked when the register read is done with a flag of “_DAA_REG_READ”. If not, the user must keep invoking the macro until it is accepted.

Example

```
Void main(void)
{
   DAA_Handle handle;
While(!DAA_RGETH(handle, DAACTRL1));
…
…
}
   void ctrlCB(void* pID, Uint16 task, Uint16 arg)
{
      if (task & _DAA_REG_READ)
      {
         printf(“DAA control 1 register has a value of
         %d\n”, &arg);
   }
   }
```

DAA_RSETH

Macro used to write a value to a DAA register

Arguments

DAA_Handle	Handle to DAA device
REG	DAA register name
VAL	Register value to be written

Return Type CSLBool — TRUE Indicates “Accepted” and FALSE “rejected”.

Description This macro writes a value to a DAA register. The macro returns immediately with an indication of whether the request was accepted (TRUE) or rejected (FALSE). If accepted, the user defined control callback will be invoked when the register write is done with a flag of “_DAA_REG_WRITE”. If not, the user must keep invoking the macro until it is accepted.

Example

```
Void main(void)
{
   DAA_Handle handle;
   Uint16 val = DAA_DAACTRL1_OH_OFFHOOK;
/* Take DAA off-hook */
While(!DAA_RSETH(handle, DAACTRL1, val));
…
…
}
   void ctrlCB(void* pID, Uint16 task, Uint16 arg)
{
      if (task & _DAA_REG_WRITE)
      {
         printf(“DAA control 1 register write
         complete!\n”);
   }
   }
```

DAA_ADDR

Macro used to return the address of a DAA register

Arguments	REG	DAA register name e.g SRCTRL
Return Type	REGADDR	Register address
Description	This macro returns the address of a DAA register.	
Example	`reg5Addr = DAA_ADDR(DAACTRL1);`	

DAA_FMK

Macro used to return 16-bit register values

Arguments	REG	DAA register name
	FIELD	Field name
	VAL	Field value
Return Type	REGADDR	Register address
Description	This macro returns a 16-bit register value for ***REG*** with ***FIELD*** set to ***VAL***.	
Example	`offHook = DAA_FMK(DAACTRL1, OH, 1);`	

Chapter 5

DAT Module

The handle-based DAT (data) module allows you to use DMA hardware to move data.

Topic **Page**

5.1 Overview

The handle-based DAT (data) module allows you to use DMA hardware to move data. This module works the same for all devices that support DMA regardless of the type of DMA controller; therefore, any application code using the DAT module is compatible across all devices as long as the DMA supports the specific address reach and memory space.

The DAT copy operations occur on dedicated DMA hardware independent of the CPU. Because of this asynchronous nature, you can submit an operation to be performed in the background while the CPU performs other tasks in the foreground. Then you can use the DAT_wait() function to block completion of the operation before moving to the next task.

Since the DAT module uses the DMA peripheral, it cannot use a DMA channel that is already allocated by the application. To ensure this does not happen, you must call the DAT_open() function to allocate a DMA channel for exclusive use. When the module is no longer needed, you can free the DMA resource by calling DAT_close().

Table 5–1 lists the functions for use with the DAT modules. The functions are presented in the order that they will typically be used in an application.

Note:

1) **Multiplexing Across Different Devices:**
   To simplify the Interrupt multiplexing across different devices, the C54x DAT module uses only DMA channels 2 and 3.

2) **Memory Spaces:**
   The DAT module contains functions to copy data from one location to another and to fill a region of memory in program, data, or I/O space valid for the specific device (Refer to the C54x data sheets). CSL does not perform any searches for invalid memory addresses.

Table 5–1. DAT Functions

Function	Purpose	See page ...
DAT_close()	Closes a DAT channel	5-4
DAT_copy()	Copies a linear block of data from src to dst using DMA hardware	5-4
DAT_copy2D()	Copies 2D data from src to dst using DMA hardware	5-6
DAT_fill()	Fills a linear block of memory with the specified fill value using DMA hardware	5-8
DAT_open()	Opens a DAT channel	5-9
DAT_wait()	Waits for a previous transfer to complete	5-10

5.2 Functions

This section describes, in alphabetical order, the functions in the DAT module.

DAT_close — *Closes the DAT module*

Function

```
void DAT_close(
    DAT_Handle hDat
);
```

Arguments

hDat Handle to a DAT channel (obtained via DAT_open)

Return Value

None

Description

Closes a DAT channel previously opened with DAT_open(). Any pending requests are first allowed to complete.

Example

```
DAT_close(hDat);
```

DAT_copy — *Copies linear block of data from src to dst*

Function

```
Uint16 DAT_copy(
    DAT_Handle hDat,
    Uint32 src,
    Uint32 dst,
    Uint16 ElemCnt
);
```

Arguments

hDat Handle to a DAT channel (obtained via DAT_open)

src Source address ORed with any of the following memory space symbols:

- ❏ DAT_PROGRAM_SPACE (Not valid for devices 5402, 5471, and 5472)
- ❏ DAT_DATA_SPACE
- ❏ DAT_IO_SPACE (Not valid for devices 5402, 5471, and 5472)

For example:

- ❏ 0x10000 | DAT_PROGRAM_SPACE indicates address 0x10000 in program space
- ❏ 0x10000 | DAT_DATA_SPACE indicates address 0x10000 in data space
- ❏ 0x100 | DAT_IO_SPACE indicates address 0x100 in I/O space;

dst	Destination address ORed with a memory space symbol
ElemCnt	Number of 16-bit words to copy

Return Value

DMA status — Returns status of data transfer at the moment of exiting the routine:
0: transfer complete
1: on-going transfer

Description

Copies a linear block of data from src to dst using DMA hardware.

You must open the DAT channel with DAT_open() before calling this function. You can use the DAT_wait() function to poll for the completed transfer of data.

Example

```
#define DATA_SIZE 256 // number of 16-bit elements to transfer
Uint16 BuffA[DATA_SIZE];
Uint16 BuffB[DATA_SIZE];
DAT_Handle hDat;
main() {
…
   hDat = DAT_open(DAT_CHA_ANY,DAT_PRI_LOW,0);
   DAT_copy(
      hDat,
      (Uint32) (&BuffA) | DAT_DATA_SPACE,
      (Uint32) (&BuffB) | DAT_DATA_SPACE,
      DATA_SIZE
   );
…
}
```

DAT_copy2D *Copies data from src to dst*

Function

```
Uint16 DAT_copy2D(
    DAT_Handle hDat,
    Uint16 Type,
    Uint32 src,
    Uint32 dst,
    Uint16 LineLen,
    Uint16 LineCnt,
    Uint16 LinePitch
);
```

Arguments

hDat — Handle to a DAT channel (obtained via DAT_open)

Type — Type of 2D DMA transfer, must be one of the following:

- ❏ DAT_1D2D : 1D to 2D transfer
- ❏ DAT_2D1D : 2D to 1D transfer
- ❏ DAT_2D2D : 2D to 2D transfer

src — Pointer to source ORed with any of the following memory space symbols:

- ❏ DAT_PROGRAM_SPACE (Not valid for devices 5402, 5471, and 5472)
- ❏ DAT_DATA_SPACE
- ❏ DAT_IO_SPACE (Not valid for devices 5402, 5471, and 5472)

For example:

- ❏ 0x10000 | DAT_PROGRAM_SPACE indicates address 0x10000 in program space;
- ❏ 0x10000 | DAT_DATA_SPACE indicates address 0x10000 in data space;
- ❏ 0x100 | DAT_IO_SPACE indicates address 0x100 in I/Ospace;

dst — Pointer to destination address ORed with a memory space symbol

LineLen — Number of 16-bit words to copy for each line

LineCnt — Number of lines to copy

LinePitch — Pitch of each line, number of 16-bit words

Return Value

DMA status — Returns status of data transfer at the moment of exiting the

routine:
0: transfer complete
1: on-going transfer

Description

This function copies 2D data from src to dst using DMA hardware.

You must open the DAT channel with DAT_open() before calling this function. You can use the DAT_wait() function to poll for the completed transfer of data.

Example

```
#define DATA_SIZE 256
Uint16 BuffA[DATA_SIZE];
Uint16 BuffB[DATA_SIZE];
DAT_Handle hDat;
main(){
…
   hDat = DAT_open(DAT_CHA_ANY,DAT_PRI_LOW,0);
   DAT_copy2D(
      hDat,
      DAT_2D2D,
      (Uint32) (&BuffA) | DAT_DATA_SPACE,
      (Uint32) (&BuffB) | DAT_DATA_SPACE,
      10,20,10
   );
…
}
```

DAT_fill

Fills linear block of memory with specified fill value

Function

```
Uint16 DAT_fill(
    DAT_Handle hDat;
    Uint32 dst,
    Uint16 ElemCnt,
    Uint32 Value
);
```

Arguments

hDat — Handle to a DAT channel

dst space — Destination address ORed with any of the following memory symbols:

- ❑ DAT_PROGRAM_SPACE
- ❑ DAT_DATA_SPACE
- ❑ DAT_IO_SPACE

For example:

- ❑ 0x10000 | DAT_PROGRAM_SPACE indicates address 0x1000 in program space;
- ❑ 0x10000 | DAT_DATA_SPACE indicates address 0x10000 in data space;
- ❑ 0x100 | DAT_IO_SPACE indicates address 0x100 in I/Ospace;

ElemCnt — Number of bytes to fill (must be power of 2)

Value — fill value

Return Value

DMA status — Returns status of data transfer at the moment of exiting the routine:
0: transfer complete
1: on-going transfer

Description

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

You must open the DAT channel with DAT_open() before calling this function. You can use the DAT_wait() function to poll for the completed transfer of data.

Example

```
#define BUFF_SIZE 256;
Uint16 Buff[BUFF_SIZE];
Uint32 FillValue = 0xA5A5;
DAT_Handle hDat;
…
   hDat = DAT_open(DAT_CHA_ANY,DAT_PRI_LOW,0);
   DAT_fill(
      hDat, (Uint32)(&Buff) | DAT_DATA_SPACE, BUFF_SIZE,
FillValue
);
```

DAT_open

Opens a DAT module

Function

```
DAT_Handle DAT_open(
    int ChaNum,
    int Priority,
    Uint32 Flags
);
```

Arguments

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

- ❏ DAT_CHA_ANY (allocates Channel 2 or 3)
- ❏ DAT_CHA2
- ❏ DAT_CHA3

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

- ❏ DAT_PRI_LOW sets the DMA channel for low priority level
- ❏ DAT_PRI_HIGH sets the DMA channel for high priority level

Flags — Miscellaneous open flags (currently None available).

Return Value

Handle for DAT channel. If the requested DMA channel is currently being used, an INV(-1) value is returned.

Description

Opens the DAT module. You must call this function before using any of the other DAT API functions. The ChaNum argument specifies which DMA channel to open for exclusive use by the DAT module. Currently, no flags are defined and the argument should be set to zero.

Example 1

To open a DAT channel using any available DMA channel (2 or 3 only) in low priority mode:

```
DAT_Handle hdat;
hdat = DAT_open(DAT_CHA_ANY,DAT_PRI_LOW,0);
```

Example 2

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

```
DAT_Handle hdat;
hdat = DAT_open(DAT_CHA2,DAT_PRI_HIGH,0);
```

DAT_wait *Waits for previous transfer to complete*

Function

```
void DAT_wait(
    DAT_Handle hDat
);
```

Arguments

hDat Handle to a DAT channel

Return Value

None

Description

This function polls the IFR flag to see if the DMA channel has completed a transfer. If the transfer is already completed, the function returns immediately. If the transfer is not complete, the function waits for completion of the transfer as identified by the handle; interrupts are not disabled during the wait.

Example

```
Uint16 TransferStat;
DAT_Handle hDat;
main(){
…
   hDat = DAT_open(DAT_CHA_ANY, DAT_PRI_LOW, 0);
   …
   TransferStat = DAT_copy(hDat, src,dst,len);
   /* custom DAT configuration */
   if (TransferStat)
   DAT_wait(hDat);
...
}
```

Chapter 6

DMA Module

This chapter describes the structure, functions, and macros of the DMA module.

Topic **Page**

6.1 Overview

The DMA module is a handle-based module that requires a call to DMA_open() to obtain a handle before any other functions are called.

The CSL module is not the same for all C54x devices. The differences mainly relate to:

- ❏ Individual channel register reload support
- ❏ Extended Data Memory Support

For more information regarding the DMA support in the C54x family, refer to Table 1–9 on page 1-17.

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

Table 6–1. DMA Configuration Structure

Structure	Purpose	See page ...
DMA_Config	DMA structure that contains all local registers required to set up a specific DMA channel.	6-4
DMA_GblConfig	Global DMA structure that contains all global registers that you may need to initialize a DMA channel	6-5

Table 6–2. DMA Functions

(a) DMA Primary Functions

Function	Purpose	See page ...
DMA_close()	Closes a DMA channel	6-7
DMA_config()	Sets up the DMA channel using the configuration structure	6-7
DMA_configArgs()	Sets up the DMA channel using the register values passed in	6-8
DMA_open()	Opens a DMA channel	6-10
DMA_pause()	Pauses a DMA channel. Identical to DMA_stop().	6-11
DMA_reset()	Resets DMA channel register to their power-on reset value	6-11
DMA_start()	Starts a DMA channel	6-11
DMA_stop()	Disables a DMA channel	6-12

Table 6–2. DMA Functions (Continued)

(b) DMA Global Register Function

Function	Purpose	See page ...
DMA_globalAlloc()	Allocates a global DMA register	6-12
DMA_globalConfig()	Sets up the DMA channel using the configuration structure	6-14
DMA_globalConfigArgs()	Sets up the DMA channel using the register values passed in	6-15
DMA_globalFree()	Frees a global DMA register that was previously allocated	6-16
DMA_globalGetConfig()	Gets DMA global register configuration	6-17
DMA_resetGbl()	Resets the DMA global registers	6-17

(c) DMA Auxiliary Functions

Function	Purpose	See page ...
DMA_autostart()	Enables a DMA channel	6-18
DMA_getConfig()	Get DMA channel configuration	6-18
DMA_getEventId()	Returns the IRQ Event ID for the DMA completion interrupt	6-19

6.2 Configuration Structure

Because the DMA has both local and global registers to 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_GblConfig (global configuration structure)** contains all the global registers that you may need to initialize a DMA channel. These global registers are resources shared across the different DMA channels and include element/frame indexes, 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 channel configuration structure*

Structure DMA_Config

Members

Uint16 priority	DMA channel priority

For devices supporting individual channel reload registers (see note) add:

Uint16 autoix	Provided for compatibility with older versions of DMA

For all devices add:

Uint16 dmmcr	DMA transfer mode control register
Uint16 dmsfc	DMA sync select and frame count register
DMA_AdrPtr dmsrc	DMA source address register
DMA_AdrPtr dmdst	DMA destination address register
Uint16 dmctr	DMA element count register

For devices supporting individual channel reload registers (see note) add:

DMA_AdrPtr dmgsa	DMA source address reload
DMA_AdrPtr dmgda	DMA destination address reload
Uint16 dmgcr	DMA element count reload
Uint16 dmgfr	DMA frame count reload

For devices supporting individual channel extended data memory addressing (see note), add:

Uint16 dmsrcdp	data page for src
Uint16 dmdstdp	data page for dst

Note:

For more information concerning these devices, see section 1.8 *Device-Specific Features Support*.

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

Example

```
DMA_Config MyConfig = {
0,                    /* priority  */
0x0000,               /* xfrctrl   */
0x0000,               /* syncframe */
(DMA_AdrPtr) 0x0300,  /* src       */
(DMA_AdrPtr) 0x0400,  /* dst       */
0x00FF                /* xfrcnt    */
};
```

DMA_GblConfig *DMA global configuration structure*

Structure DMA_GblConfig

Members Uint16 free run free under emulation control

For devices supporting individual channel reload registers (see note) add:
Uint16 autoix For compatibility with older versions of DMA

For all devices add:
Uint16 gbldmsrcp global program page for src
Uint16 gbldmdstp global program page for dst
Uint16 gbldmidx0 global element index 0
Uint16 gbldmfri0 global frame index 0
Uint16 gbldmidx1 global element index 1
Uint16 gbldmfri1 global frame index 1

For devices not offering global channel reload registers (see note), add:
DMA_AdrPtr gbldmgsa global src address reload
DMA_AdrPtr gbldmgda global dst address reload
Uint16 gbldmgcr global element count reload
Uint16 gbldmgfr global frame count reload

For devices supporting global extended data memory addressing (see note), add:
Uint16 gbldmsrcdp; global data page for src
Uint16 gbldmdstdp; global data page for dst

> **Note:**
>
> For more information concerning these devices, see section 1.8, *Device-Specific Features Support*.

Description

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

Example 1

```
   DMA_GblConfig MyGblConfig = {
                        0,   /* stop under emulation control */
                        10,  /* src program page             */
                        20,  /* dst program page             */
                        0x1, /* index 0                      */
                        0x4  /* frame index 0                */
                        0,   /* index 1                      */
                        0,   /* frame index 1                */
                        0,   /* src data page                */
                        0    /* dst data page                */
}
```

In this example, source and destination pages are hard-coded.

For a complete example, see Example 2 in section 6.6.

Example 2

```
extern DMA_AdrPtr mySrc;
extern myDst;
DMA_gblConfig myGblConfig = {
                        0,
                        0,
                        0,
                        0x1,
                        0x4
                        0,
                        0
                    };
....
myGblConfig.gbldmsrcp = (((Uint32)(&mySrc)>>16)&0xFFFFu);
myGblConfig.gbldmdstp = (((Uint32)(&myDst)>>16)&0xFFFFu);
```

6.3 Functions

This section describes the functions in the DMA CSL module.

6.3.1 DMA Primary Functions

DMA_close *Closes DMA channel*

Function

```
void DMA_close(
    DMA_Handle hDma
);
```

Arguments

hDma Handle to DMA channel; see DMA_open()..

Return Value

None

Description

Closes a DMA channel previously opened with DMA_open(). The registers for the DMA channel are set to their power-on reset defaults, then 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 (See DMA_Config)

Return Value

None

Description

Sets up the DMA channel using the configuration structure. The values of the structure are written to the DMA registers. To start the DMA channel, call the DMA_start() function. DMA_Config() initializes the DMA channel register, but **does not** start the DMA channel.

Example

```
DMA_Config MyConfig = {
   0x0,                 /*priority */
   0x0000,               /* mcr     */
   0x0000,               /* sfc     */
   (DMA_AdrPtr) 0x0300,  /* src     */
   (DMA_AdrPtr) 0x0400,  /* dst     */
   0x00FF                /* ctr     */
};
   …
   DMA_config(hDma,&MyConfig);
```

For complete examples, please refer to section 6.6, *Examples*.

DMA_configArgs *Sets up DMA channel with register values*

Function

void DMA_configArgs(
    DMA_Handle hDma,
    Uint16 priority,
For devices supporting individual channel reload registers (see note) add:
    Uint16 autoix (Provided for compatibility with older versions of DMA)
For all devices add:
    Uint16 dmmcr,
    Uint16 dmsfc,
    Uint16 dmsrc,
    Uint16 dmdst,
    Uint16 dmctr,

For devices supporting individual channel reload registers (see note), add:
    Uint16 dmgsa,
    Uint16 dmgda,
    Uint16 dmgcr,
    Uint16 dmgfr,

For devices supporting individual channel extended data memory addressing (see note), add:
    Uint16 dmsrcdp,
    Uint16 dmdstdp
);

Note:

For more information concerning these devices, see section 1.8, *Device-Specific Features Support*.

Arguments

hDma Handle to DMA channel; see DMA_open()
priority DMA channel priority

For devices supporting individual channel reload registers (see note) add:
Uint16 autoix (Provided for compatibility with older versions of DMA)

dmmcr DMA transfer mode control register value
dmsfc DMA sync select and frame count register value
dmsrc DMA source address register value
dmdst DMA destination address register value
dmctr DMA element count register value

For devices supporting individual channel reload registers (see note):
dmgsa Pointer to DMA source address reload value
dmgda Pointer to DMA destination address reload value
dmgcr DMA element count reload value
dmgfr DMA frame count reload value

For devices supporting individual channel extended data memory addressing (see note), add:
Uint16 dmsrcdp data page for src
Uint16 dmdstdp data page for dst

Return Value

None

Description

Sets up the DMA channel with the register values passed to the function. The register values are written to the DMA registers. To start the DMA channel, you must call the DMA_start() function. DMA_Config() initializes the DMA channel register, but **does not** start the DMA channel.

You may use literal values for the arguments; or for readability, you may use the *MK macros* to create the register values based on field values.

Example

```
DMA_configArgs(hDma,
  0x0000, /* channel priority  */
  0x0000, /* mcr */
  0x0000, /* sfc */
  0x0300, /* src */
  0x0400, /* dst */
  0x00FF  /* ctr */
);
```

For a complete example, see Section 5.4, Example 1B.

DMA_open — *Opens DMA channel*

Function

```
DMA_Handle DMA_open(
    int ChaNum,
    Uint32 Flags
);
```

Arguments

ChaNum	DMA channel to open: DMA_CHA_ANY DMA_CHA0 DMA_CHA1 DMA_CHA2 DMA_CHA3 DMA_CHA4 DMA_CHA5
Flags	Open flags (logical OR of any of the following): DMA_OPEN_RESET

Return Value

Device handle	Handle to newly opened device

Description

Opens a DMA channel. Before a DMA channel can be used, you must first call this function to open the channel. Once opened, it cannot be opened again before you call DMA_close(). The return value is a unique device handle for use in subsequent DMA API calls. If the open fails, INV is returned.

You can use this function in either of the following ways:

- ❏ Specify exactly which physical channel to open.
- ❏ Use DMA_CHA_ANY to allow the library pick an unused channel.

If you specify the DMA_OPEN_RESET flag, the DMA channel registers are set to the power-on reset defaults and the channel interrupt is disabled and cleared. Use this flag when the DMA channel has been running to clean previously set status and interrupt flags.

Example

```
DMA_Handle hDma;
    ...
   hDma = DMA_open(DMA_CHA_ANY,DMA_OPEN_RESET);
```

DMA_pause — *Pauses DMA channel*

Function

```
void DMA_pause(
    DMA_Handle hDma
);
```

Arguments

hDma — Handle to DMA channel; see DMA_open().

Return Value

None

Description

Identical to DMA_stop(). This is provided for compatibility with other TMS320 devices only.

Example

```
DMA_pause(hDma);
```

DMA_reset — *Resets DMA channel*

Function

```
void DMA_reset(
    DMA_Handle hDma
);
```

Arguments

hDma — Handle to DMA channel; see DMA_open().. or INV (If you want to reset all DMA channel registers)

Return Value

None

Description

Resets the DMA channel by setting its registers to the power-on defaults and disables and clears the channel interrupt. You can 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*

Function

```
void DMA_start(
    DMA_Handle hDma
);
```

Arguments

hDma — Handle to DMA channel; see DMA_open().

Return Value

None

Description

Starts a DMA channel by setting the enable channel bits in the DMA priority and enable control register (DMPREC) accordingly to 1. See DMA_stop().

Example

```
DMA_start(hDma);
```

DMA_stop *Disables DMA channel*

Function

```
void DMA_stop(
    DMA_Handle hDma
);
```

Arguments hDma Handle to DMA channel; see DMA_open().

Return Value None

Description Disables the DMA channel by resetting the enable channel bits in the DMA priority and enable control (DMPREC) register accordingly. See DMA_start() .

Example

```
DMA_stop(hDma);
```

6.3.2 DMA Global Register Function

DMA_globalAlloc *Performs global register allocation*

Function

```
Uint16 DMA_globalAlloc (
    Uint16 RegMask
);
```

Arguments RegMask Mask that indicates which global registers you want to use; must be one of the following:
DMA_GBL_DMIDXANY (any global index register)
DMA_GBL_DMIDX0 (global index 0)
DMA_GBL_DMIDX1 (global index 1)
DMA_GBL_DMFRI0 (global frame index 0)
DMA_GBL_DMFRI1 (global frame index 1)
DMA_GBL_RLDR (global reload registers)
DMA_GBL_SRCP (global program page for src)
DMA_GBL_DSTP (global program page for dst)
DMA_GBL_SRCDP (global data page for src)
DMA_GBL_DSTDP (global data page for dst)
DMA_GBL_ALL **(all global registers)**

Note:

In the C54x, the DMA_GBL_DMFRIx and DMA_GBL_DMIDXx masks should used in pairs. For example, when you use DMA_GBL_DMFRI0, you should also use DMA_GBL_DMIDX0. Similarly both DMA_GBL_DMFRI1 and DMA_GBL_DMIDX1 should be used. If you do not follow this guideline, the function allocates all registers (DMA_GBL_DMFRI0, DMA_GBL_DMFRI1, DMA_GBL_DMIDX0, DMA_GBL_DMIDX1). If you use DMA_GBL_DMIDXANY, the function allocates any of the available DMA_GBL_DMFRIx/DMA_GBL_DMIDXx pairs.

Return Value

RegMaskalloc — Mask that indicates the global registers that are being allocated as a response to the current RegMask requests. This mask does NOT include registers requested via previous calls to DMA_globalAlloc().

If ANY of the RegMask requests cannot be fulfilled, then RegMaskAlloc equals zero.

Description

Performs Global register allocation. This function returns a mask that indicates to the DMA_global Config/ConfigArgs functions which global registers are being allocated for the DMA channel. If you request via RegMask a global register that has been previously allocated the function returns a zero.

The use of this function is considered optional. It can be used to prevent double allocation of registers to DMA channels. If not used, you can pass off the DMA_GBL_ALL (0xffff value) as the RegMaskAlloc parameter for the DMA_global Config/Args functions.

Example

```
#define NOTUSED 0

    DMA_GblConfig MyGblConfig = {
      0,              /* free emulator control */
      10,             /* src program page      */
      20,             /* dst program page      */
      0x1,            /* index 0               */
      0x4             /* frame index 0         */
      NOTUSED,        /* index 1               */
      NOTUSED         /* frame index 1         */
       };

.....

      mask = DMA_globalAlloc (DMA_GBL_DMIDX1|DMA_GBL_DMFRI1);
      DMA_globalConfig (mask, &MyGblConfig);
```

For a complete example, see Section 6.6, Example 2.

DMA_globalConfig *Sets up DMA global registers using configuration structure*

Function

```
void DMA_globalConfig (
    Uint16 RegMaskAlloc,
    DMA_GblConfig *Config
    );
```

Arguments

RegMaskAlloc	Mask to indicate global registers to initialize. This argument is produced by the DMA_GlobalAlloc function. A value of DMA_GBL_ALL(0xffff value) allocates all the global registers specified in Config.
Config	Pointer to an initialized global configuration structure

Return Value

None

Description

Sets up the DMA global registers using the global configuration structure. The values of the structure are written to the DMA global registers. Since the DMA global registers are shared, this function will ONLY initialize the registers that have been allocated via a DMA_globalAlloc routine and passed to this function via the RegMaskAlloc value. See DMA_globalAlloc.

This function is optional. It may not be necessary to use this function if no global resource register initialization (element/frame indexes, reload registers, and src/dst page registers) is required for the DMA transfer.

Example

```
#define NOTUSED 0

    DMA_GblConfig MyGblConfig = {
      0,                          /* free emulator control */
      10,                         /* src program page */
      20,                         /* dst program page */
      0x1,                        /* index 0 */
      0x4                         /* frame index 0 */
      NOTUSED,                    /* index 1 */
      NOTUSED,                    /* frame index 1 */
      (DMA_AdrPtr) 100,           /* src data page */
      (DMA_AdrPtr) 101,           /* dst data page */
       }

   .....

      mask = DMA_globalAlloc (DMA_GBL_DMIDX1|DMA_GBL_DMFRI1);
      DMA_globalConfig (mask, &MyGblConfig);
```

For a complete example, see Section 6.6, Example 2.

DMA_globalConfigArgs *Sets up DMA global registers using arguments*

Function

```
void DMA_globalConfigArgs(
    Uint16 RegMask,
    Uint16 free,
```

For devices supporting individual channel reload registers (see note) add:

```
    Uint16 autoix, (Provided for compatibility with older versions of DMA)
```

For all devices add:

```
    Uint16 intosel,
    Uint16 dmidx0,
    Uint16 dmfri0,
    Uint16 dmidx1,
    Uint16 dmfri1,
```

For devices not supporting global channel reload registers,(see section 1.8) add:

```
    Uint16 dmgsa,
    Uint16 dmgda,
    Uint16 dmgc,
    Uint16 dmgcr,
    Uint16 dmgfr,
```

For all devices, add:

```
    Uint16 dmsrcp,
    Uint16 dmdstp,
```

For devices supporting extended DMA data support, (see section 1.8) add:

```
    Uint16 dmsrcdp,
    Uint16 dmdstdp
```

Arguments

RegMask Mask to indicate global registers to initialize. This argument is produced by the DMA_GlobalAlloc function. A value of 0xffff (DMA_GBL_ALL) allocates all the global registers specified in Config.

For devices supporting individual channel reload registers (see note) add:

autoix (Provided for compatibility with older versions of DMA)

free;	Response to emulation control
dmidx0;	Global element index 0
dmfri0;	Global frame index 0
dmidx1;	Global element index 1
dmfri1;	Global frame index 1

For devices supporting global channel reload registers,(see section 1.8):

dmgsa;	Pointer to global src address reload
dmgda;	Pointer to global dst address reload
dmgcr;	Global element count reload
dmgfr;	Global frame count reload

For all devices:

dmsrcp;	Global program page for src
dmdstp;	Global program page for dst

For devices supporting extended data addressing (see section 1.8):

dmsrcdp;	Global data page for src
dmdstdp;	Global data page for dst

Return Value None

Description Sets up the DMA global registers with the register values passed to the function. The register values are written to the DMA global registers. Since the DMA global registers are shared, this function will ONLY initialize the registers that have been allocated via a DMA_globalAlloc routine and passed to this function via the RegMaskAlloc value. See DMA_globalAlloc().

Example None

DMA_globalFree *Frees global DMA register that was previously allocated*

Function

```
void DMA_globalFree(
    Uint16 regMask
);
```

Arguments

regMask	Global register mask that can be obtained from DMA_globalAlloc( ). A value of 0xffff (DMA_GBL_ALL) frees all of the global DMA registers.

Return Value None

Description Frees global DMA registers that were previously allocated by calling DMA_globalAlloc(). Once freed, the register is again available for allocation.

Example

```
Uint16 RegMask;
    …
      RegMask = DMA_globalAlloc(DMA_GBL_DMIDX0, DMA_GBL_DMIDX0);
    …
    /* some time later on when you're done with it */
    DMA_globalFree(RegMask);
```

DMA_globalGetConfig *Gets a DMA global configuration register*

Function

```
void DMA_globalGetConfig (
    Uint16 RegMaskAlloc,
    DMA_GblConfig *Config
    );
```

Arguments

RegMaskAlloc	Mask that indicates which global register to get. Refer to DMA_globalAlloc for valid values. DMA_GBL_ALL will get all global registers
Config	Pointer to an un-initialized global configuration structure

Return Value

None

Description

Specifies the current configuration for the DMA global registers specified by RegMask. This is accomplished by reading the actual DMA global registers and fields and storing them back in the config structure.

Example

```
DMA_GblConfig  ConfigRead;
...
DMA_globalGetConfig (DMA_GBL_ALL, &ConfigRead);
```

DMA_resetGbl *Resets a DMA global register*

Function

```
void DMA_resetGbl(
    DMA_Handle hDma
);
```

Arguments

hDma	Handle to DMA channel; see DMA_open(), Or INV (-1) if you want to reset all DMA channel registers.

Return Value

None

Description

Resets the DMA global register by setting all global registers to the power-on defaults. You must use INV (-1) as the device handle to reset all the global registers.

Example

```
DMA_resetGbl(hDma);

/* or */
DMA_resetGbl(INV);
```

6.3.3 DMA Auxiliary Functions

DMA_autostart *Enables the specified DMA channel and sets the AUTOINIT bit*

Function

```
void DMA_autostart(
    DMA_Handle hDma
);
```

Arguments

hDma — Handle to DMA channel; see DMA_open().

Return Value

None

Description

Enables the specified DMA channel and sets the AUTOINIT bit.

Example

```
DMA_autostart(hDma);
```

DMA_getConfig *Gets a DMA channel configuration*

Function

```
void DMA_getConfig(
    DMA_Handle hDma
    DMA_Config  *Config
    );
```

Arguments

hDma — Handle to DMA channel; see DMA_open().

Config — Pointer to an un-initialized configuration structure (see DMA_Config)

Return Value

None

Description

Gets the current configuration for the DMA channel used by handle. This is accomplished by reading the actual DMA channel registers and fields and storing them back in the Config structure.

Example

```
DMA_Config    ConfigRead;

...
myHdma = DMA_open (DMA_CHA0, 0);
DMA_getConfig (myHdma, &ConfigRead);
```

DMA_getEventId — *Returns IRQ Event ID for DMA completion interrupt*

Function

```
Uint16 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);
```

For a complete example, see Section 6.6, Example 2.

DMA_getStatus — *Gets DMA channel status*

Function

```
Uint16 DMA_getStatus (
    DMA_Handle hDma
);
```

Arguments

hDma

Return Value

1: if DMA channel is still running
0: if DMA channel has stopped (transfer completed)

Description

Returns the status of the DMA channel used by handle. Use as a indication of transfer complete.

Example

```
while (DMA_getStatus(myHdma)); /*wait for transfer to complete */
```

For a complete example of DMA_getStatus, see Section 5.4 (Example 1a)

6.4 Macros

As covered in section 1.5, the CSL offers a collection of macros that allow individual access to the peripheral registers and fields. To use the DMA macros include “csl_dma.h” in your project.

Because the DMA has several channels, the macros identify the channel used by either the channel number or the handle used. Table 6–3 lists the macros available for a DMA channel, using the channel number as part of the register name. Table 6–4 lists the macros available for a DMA channel using its corresponding handle.

Table 6–3. DMA CSL Macros (using channel number)

(a) Macros to read/write DMA register values
DMA_RGET()
DMA_RSET()
(b) Macros to read/write DMA register field values(Applicable only to registers with more than one field)
DMA_FGET()
DMA_FSET()
(c) Macros to create value to write to a DMA register and fields (Applicable only to registers with more than one field)
DMA_REG_RMK()
DMA_FMK()
(d) Macros to read a register address
DMA_ADDR()

Table 6–4. DMA CSL Macros (using handles)

(a) Macros to read/write DMA register values
DMA_RGETH()
DMA_RSETH()
(b) Macros to read/write DMA register field values(Applicable only to registers with more than 1-field)
DMA_FGETH()
DMA_FSETH()
(c) Macros to read a register address
DMA_ADDRH()

DMA_RGET *Get value of DMA register*

Macro Uint16 DMA_RGET (REG)

Arguments REG LOCALREG# or GLOBALREG, where:

- LOCALREG# Local register name with channel number (#), where # = 0, 1, 2 ,3, 4, 5,
  DMSRC#
  DMDST#
  DMCTR#
  DMSFC#
  DMMCR#

  For devices supporting individual channel reload registers, add:
  DMGSA#
  DMGDA#
  DMGCR#
  DMGFR#

  For devices supporting individual channel extended data memory space support, add:
  DMSRCDP#
  DMDSTDP#

- GLOBALREG Global register name
  DMPREC
  DMSRCP
  DMDSTP

  For devices not supporting individual channel extended data memory space support, add:
  DMSRCDP
  DMDSTDP

  For devices supporting global channel reload registers, add:
  DMGSA
  DMGDA
  DMGCR
  DMGFR

  For devices supporting global extended data memory space support, add:
  DMSRCDP#
  DMDSTDP#

Return Value value of register

Description Returns the DMA register value

Example 1 For local registers:

```
Uint16 myvar;
myVar = DMA_RGET(DMSRC1);   /* read DMSRC for channel 1 */
```

Example 2 For global registers:

```
Uint16 myVar;
   ...
   myVar = DMA_RGET(DMPREC);
```

DMA_RSET *Set value of DMA register*

Macro Void DMA_RSET (REG, Uint16 regval)

Arguments

REG LOCALREG# or GLOBALREG, as listed in DMA_RGET() macro

regval register value that wants to write to register REG

Return Value value of register

Description Set the DMA register REG value to regval

Example 1 For local registers:

```
DMA_RSET(DMSRC1, 0x8000);   /*DMSRC for channel 1 = 0x8000 */
```

Example 2 For global registers:

```
DMA_RSET(DMSRCP, 3);   /* DMSRCP = 3 */
```

DMA_REG_RMK *Creates register value based on individual field values*

Macro

Uint16 DMA_REG_RMK (fieldval_n,...,fieldval_0)

Arguments

REG — Only writable registers containing more than one field are supported by this macro. Also notice that the channel number is not used as part of the register name.
For example:
DMSFC
DMMCR
DMPREC

fieldval — Field values to be assigned to the writable register fields.
Rules to follow:

- ❏ Only writable fields are allowed
- ❏ Start from Most-significant field first
- ❏ Value should be a right-justified contant. If fieldval_n value exceeds the number of bits allowed for that field, fieldval_n is truncated accordingly.

Return Value

Value of register that corresponds to the concatenation of values passed for the fields.

Description

Returns the DMA register value given specific field values. You can use constants or the CSL symbolic constants covered in Section 1.6.

Example

```
Uint16 myregval;
myregval = DMA_DMSFC_RMK (0,0,3); /* dsyn,dblw,framecount fiel
ds */
```

or you can use the PER_REG_FIELD_SYMVAL symbolic constants provided in CSL (see section 1.6).

```
myregval=DMA_DMSFC_RMK
(DMA_DMSFC_DSYN_NONE, DMA_DMSFC_DBLW_OFF, 3);
```

DMA_REG_RMK are typically used to initialize a DMA configuration structure used for the DMA_config() function (see section 6.6).

DMA_FMK — *Creates register value based on individual field values*

Macro

Uint16 DMA_FMK (REG, FIELD, fieldval)

Arguments

REG — Only writable registers containing more than one field are supported by this macro. Also notice that for local registers, the channel number is not used as part of the register name.
For example:
DMPREC
DMSFC
DMMCR

FIELD — Symbolic name for field of register REG Possible values: Field names as listed in the C54x Register Reference Guide. (see Appendix A) Only writable fields are allowed.

fieldval — Field values to be assigned to the writable register fields.
Rules to follow:

- ❏ Only writable fields are allowed
- ❏ Start from Most-significant field first
- ❏ Value should be a right-justified contant. If fieldval_n value exceeds the number of bits allowed for that field, fieldval_n is truncated accordingly.

Return Value

Shifted version of fieldval. fieldval is shifted to the bit numbering appropriate for FIELD.

Description

Returns the shifted version of fieldval. Fieldval is shifted to the bit numbering appropriate for FIELD within register REG. This macro allows the user to initialize few fields in REG as an alternative to the DMA_REG_RMK() macro that requires ALL the fields in the register to be initialized. The returned value could be ORed with the result of other _FMK macros, as show below.

Example

```
Uint16 myregval;
myregval = DMA_FMK (DMSFC, DBLW, 1) | DMA_FMK (DMSFC, DSYN,
2);
```

DMA_FGET *Get value of register field*

Macro Uint16 DMA_FGET (REG, FIELD)

Arguments

REG Only writable registers containing more than one field are supported by this macro. Also notice that for local registers, the channel number is used as part of the register name.
For example:
DMPREC
DMSFC#
DMMCR#

FIELD Symbolic name for field of register REG. Possible values: Field names as listed in the C54x Register Reference Guide (see Appendix A). Only writable fields are allowed.

Return Value Value of register field

Description Gets the DMA register field value

Example 1 For local registers:

```
Uint16 myvar;
...
myregval = DMA_FGET (DMMCR1, CTMOD);
```

Example 2 For global registers:

```
Uint16 myvar;
...
myregval = DMA_FGET (DMPREC, INTOSEL);
```

DMA_FSET *Set value of register field*

Macro Void DMA_FSET (REG, FIELD, fieldval)

Arguments

REG Only writable registers containing more than one field are supported by this macro. Also notice that for local registers, the channel number is used as part of the register name.
For example:
DMPREC
DMSFC#
DMMCR#

FIELD Symbolic name for field of register REG Possible values: Field names as listed in the C54x Register Reference Guide. (see Appendix A). Only writable fields are allowed.

fieldval Field values to be assigned to the writable register fields.
Rules to follow:

- ❏ Only writable fields are allowed
- ❏ If fieldval value exceeds the number of bits allowed for field, fieldval is truncated accordingly.

Return Value None

Description Set the DMA register value to regval

Example 1 For Local Registers:

```
DMA_FSET (DMMCR1, CTMOD, 1);
```

Example 2 For global registers:

```
DMA_FSET (DMPREC, INTOSEL, 1);
```

DMA_ADDR — *Get address of given register*

Macro Uint16 DMA_ADDR (REG)

Arguments REG LOCALREG# or GLOBALREG as listed in DMA_RGET() macro

Return Value Address of register LOCALREG and GLOBALREG

Description Get the address of a DMA register. In the case of LOCALREG (sub-addressed registers), the function returns the sub-address. For example: DMA_ADDR (DMSRC1) returns a value of 5.

Example 1 For local registers:

```
myvar = DMA_ADDR (DMMCR1);
```

Example 2 For global registers:

```
myvar = DMA_ADDR (DMPREC);
```

DMA_RGETH — *Get value of DMA register used in handle*

Macro Uint16 DMA_RGETH (DMA_Handle hDma, LOCALREG)

Arguments

hDma	Handle to DMA channel that identifies the specific DMA channel used.
LOCALREG	Same register as in DMA_RGET(), but without channel number (#). Example: DMSRC (instead of DMSRC#)

Return Value Value of register

Description Returns the DMA value for register LOCALREG for the channel associated with handle.

Example

```
DMA_Handle myHandle;
Uint16  myVar;
...
myHandle = DMA_open (DMA_CHA0, DMA_OPEN_RESET);
...
myVar = DMA_RGETH (myHandle, DMMCR);
```

DMA_RSETH *Set value of DMA register*

Macro void DMA_RSETH (DMA_Handle hDma, LOCALREG, Uint16 regval)

Arguments

hDma	Handle to DMA channel that identifies the specific DMA channel used.
LOCALREG	Same register as in DMA_RSET(), but without channel number (#). Example: DMSRC (instead of DMSRC#)
regval	value to write to register LOCALREG for the channel associated with handle.

Return Value None

Description Set the DMA register LOCALREG for the channel associated with handle to the value regval.

Example

```
DMA_Handle myHandle;
...
myHandle = DMA_open (DMA_CHA0, DMA_OPEN_RESET);
...
DMA_RSETH (myHandle, DMMCR, 0x123);
```

DMA_FGETH *Get value of register field*

Macro Uint16 DMA_FGETH (DMA_Handle hDma, LOCALREG, FIELD)

Arguments

hDma	Handle to DMA channel that identifies the specific DMA channel used.
LOCALREG	Same register as in DMA_RSET(), but without channel number (#). Example: DMSRC (instead of DMSRC#) Only registers containing more than one field are supported by this macro.
FIELD	Symbolic name for field of register REG. Possible values: Field names as listed in the C54x Register Reference Guide (see Appendix A). Only readable references are allowed.

Return Value Value of register field given by FIELD, of LOCALREG use by handle.

Description Gets the DMA register field value

Example

```
DMA_Handle myHandle;
...
myHandle = DMA_open (DMA_CHA0, DMA_OPEN_RESET);
...
myVar = DMA_FGETH (myHandle, DMMCR, CTMOD);
```

DMA_FSETH *Set value of register field*

Macro void DMA_FSETH (DMA_Handle hDma, LOCALREG, FIELD, fieldval)

Arguments

hDma	Handle to DMA channel that identifies the specific DMA channel used.
LOCALREG	Same register as in DMA_RSET(), but without channel number (#). Example: DMSRC (instead of DMSRC#) Only registers containing more than one field are supported by this macro.
FIELD	Symbolic name for field of register REG. Possible values: Field names as listed in the C54x Register Reference Guide (see Appendix A). Only readable references are allowed.
fieldval	Field values to be assigned to the writable register fields. Rules to follow: ❏ Only writable fields are allowed ❏ Start from Most-significant field first ❏ Value should be a right-justified contant. If fieldval value exceeds the number of bits allowed for that field, fieldval is truncated accordingly.

Return Value None

Description Set the DMA register field FIELD of the LOCALREG register for the channel associated with handle to the value fieldval.

Example

```
DMA_Handle myHandle;
Uint16  myVar;
...
myHandle = DMA_open (DMA_CHA0, DMA_OPEN_RESET);
...
DMA_FSETH (myHandle, DMMCR, CTMOD, 1);
```

DMA_ADDRH *Get address of given register*

Macro Uint16 DMA_ADDR_H (DMA_Handle hDma, LOCALREG,)

Arguments

hDma	Handle to DMA channel that identifies the specific DMA channel used.
LOCALREG	Same register as in DMA_RSET(), but without channel number (#). Example: DMSRC (instead of DMSRC#)

Return Value Address of register LOCALREG

Description Get the address of a DMA local register (sub-address) for channel used in hDma

Example

```
DMA_Handle myHandle;
Uint16  myVar;
...
myVar = DMA_ADDRH (myHandle, DMMCR);
```

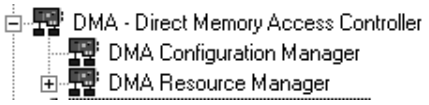
6.5 Configuring the DMA Module Using CSL GUI

The DMA module facilitates configuration of the Direct Memory Access (DMA) controller. The DMA module consists of a configuration manager and a resource 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.

Figure 6–1 illustrates the DMA sections menu on the CSL graphical user interface (GUI).

Figure 6–1. DMA Sections Menu



The DMA includes the following sections:

- **DMA Configuration Manager** allows you to create configuration objects by setting the peripheral 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 six channel handle objects are predefined.

6.5.1 DMA Configuration Manager

The DMA Configuration Manager allows you to create DMA Channel configurations through the Properties page and to generate the configuration objects.

6.5.1.1 Creating/Inserting a Configuration

There is no predefined configuration object available.

Step 1: To configure a DMA channel through the Peripheral Registers, you must insert a new configuration object.

Step 2: To insert a new configuration object, right-click on the DMA Configuration Manager and select insert dmaCfg from the drop-down menu. The configuration objects can be renamed. Their use depends on the on-chip device resources. Because six channels are available, a maximum of six configurations can be used simultaneously.

Note:

A maximum of six configurations may be inserted. This is due to the association that each configuration has with a pre-defined global configuration. The global configuration is dynamically updated with changes made to the associated DMA configuration. One DMA configuration (and its associated global configuration) can be used by more than one DMA channel.

6.5.1.2 Deleting/Renaming an Object

To delete or rename an object, right-click on the configuration object you want to delete or rename. Select Delete to delete a configuration object. Select Rename to rename the object.

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 Figure 2–1, *The CSL Tree,* on page 2-3).

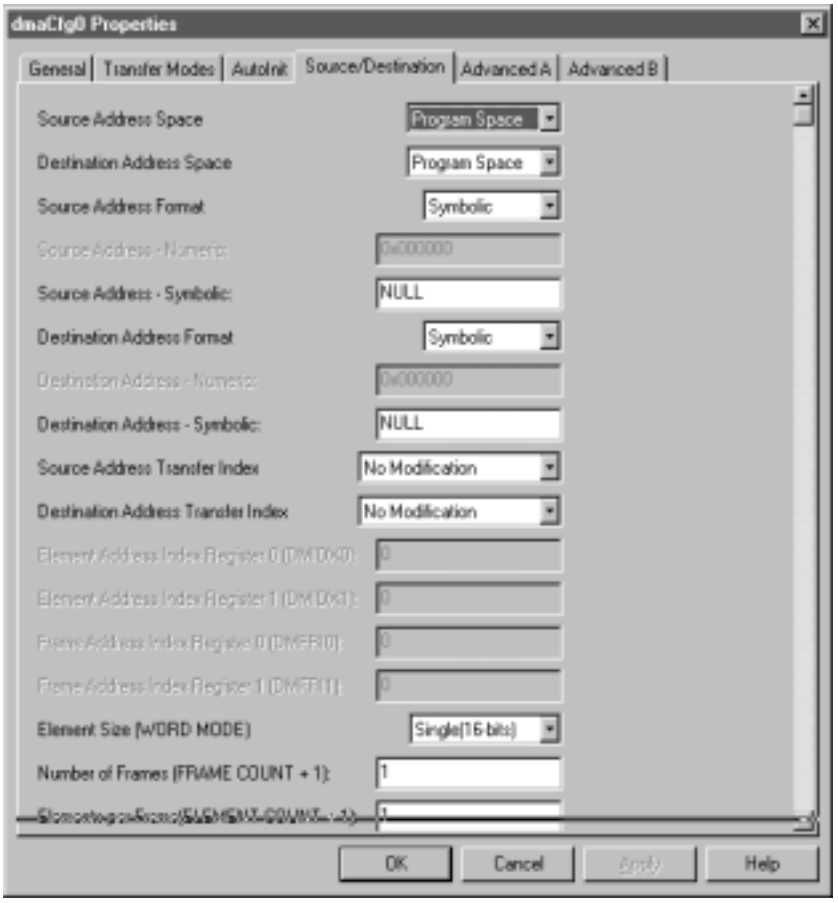
6.5.1.3 Configuring the Object Properties

You can configure object properties through the Properties dialog box. (See Figure 6–2). 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 Peripheral registers related to the DMA. You can set the configuration options through the following Tab pages:

- ❏ General: Breakpoint emulation behavior.
- ❏ Transfer Modes: Allows you to configure the Priority, Sync Events, ABU/Multi-frame
- ❏ Source/Destination: Allows you to configure the Address, Index, Element/Frame Count
- ❏ Autoinit: Allows you to configure the Reload Registers
- ❏ Advanced A and B Pages: These pages contains the full hexadecimal register values and reflects the option setting of the previous pages. Also, the full register values can be entered directly and the new options are mirrored in the related pages automatically.

Figure 6–2. DMA Properties Page



Each page is composed of several options that are set to a default value (at device reset).

6.5.1.4 Specifying Address Formats

The source, destination, and addresses can be specified in either a numeric format (hard coded address) or a symbolic format. Before setting any addresses, ensure that the right format is selected in the Source Address Format and Destination Address Format pull-down menus located on the Source and Destination tabs of the Properties page.

6.5.2 DMA Resource Manager

The DMA Resource Manager allows you to generate the DMA_open() and DMA_config() CSL functions.

Figure 6–3 illustrates the DMA Resource Manager menu on the CSL graphical user interface (GUI).

Figure 6–3. DMA Resource Manager Menu

6.5.2.1 Predefined Objects

The six channel handle objects are predefined and each is associated with a supported on-chip DMA channel as follows:

DMA0 – Default handle name: hDma0

DMA1 – Default handle name: hDma1

DMA2 – Default handle name: hDma2

DMA3 Default handle name: hDma3

DMA4 Default handle name: hDma4

DMA5 Default handle name: hDma5

6.5.2.2 Properties Page

You can generate the DMA_open() and DMA_config() CSL functions through the Properties page.

To access the Properties page, right-click on a predefined DMA channel and select Properties from the drop-down menu (see Figure 6–4).

The first time the Properties page appears, only the Open Handle to DMA check-box can be selected. Select this to open the DMA channel, allowing pre-initialization.

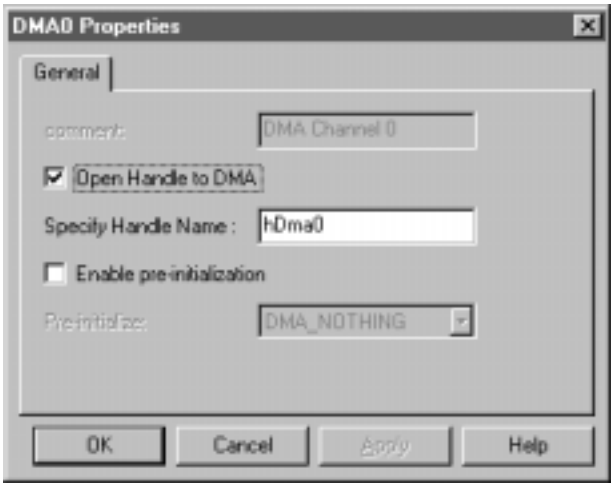
DMA_NOTHING is used to indicate that there is no configuration object selected for this DMA.

To pre-initialize the DMA channel, check the Enable pre-initialization checkbox. You can then select one of the available configuration objects (see section 6.5.1 , *DMA Configuration Manager*) for this channel through the pre-initialize drop-down list.

If DMA_NOTHING is selected, no configuration object is generated for the related DMA handle (see section 6.5.3, *C Code Generation for DMA Module*).

In the example shown in Figure 6–4, the Open DMA Channel option is checked and the handle object hDma1 is now accessible (The handle object can be renamed by typing the new name in the box provided). The DMA_open() function is now generated with hDma1 containing the returned handle address.

Figure 6–4. DMA Properties Page With Handle Object Accessible



6.5.3 C Code Generation for DMA Module

Two C files are generated from the configuration tool:

- Header file
- Source file

6.5.3.1 Header File

The header file includes all the CSL header files of the modules and contains the DMA handles, and configuration objects generated by the configuration tool (see Example 6–1).

Example 6–1. DMA Header File

```
extern DMA_Config dmaCfg0;
extern DMA_GblConfig gDMAConfig0;;
extern DMA_Handle hDma1;
```

6.5.3.2 Source File

The source file includes the declaration of the channel handle objects and the configuration structures (see Example 6–2).

Example 6–2. DMA Source File (Declaration Section)

```
/*  Config Structures */
DMA_Config dmaCfg0 = {
    0x0000,        /*  Channel Priority (0x0000 or 0x0001  */
    0x0000,        /*  Global Reload Register Usage in Autoinit Mode (AUTO
                    IX : 0x0000 or 0x0001)  */
    0x0000,        /*  Transfer Mode Control Register (DMMCR)  */
    0x0000,        /*  Sync Event and Frame Count Register (DMSFC) */
    NULL,         /*  Source Address Register (DMSRC) - Symbolic  */
    NULL,         /*  Destination Address Register (DMDST) - Symbolic  */
    0x0000         /*  Element Count Register (DMCTR) */
};

DMA_GblConfig gDMAConfig0 = {
    0x0,        /*  Breakpoint Emulation Behavior (FREE)  */
    0x0000,        /*  Global Reload Register Usage in Autoinit Mode (AUTO
                    IX : 0x0000 or 0x0001)  */
    NULL,  /*  Source Program Page Address Register (DMSRCP) - Symbolic  */
    NULL,  /*  Destination Program Page Address Register (DMDSTP) - Symbolic
*/
    0x0000,        /*  Element Address Index Register 0 (DMIDX0)  */
    0x0000,        /*  Frame Address Index Register 0 (DMFRI0)  */
    0x0000,        /*  Element Address Index Register 1 (DMIDX1)
    0x0000,        /*  Frame Address Index Register 1 (DMFRI1)
    NULL,  /*  Global Source Address Reload Register (DMGSA) - Symbolic  */
    NULL,  /*  Global Destination Address Reload Register (DMGDA) - Symbolic
*/
    0x0000,         /*  Global Element Count Reload Register (DMGCR)  */
    0x0000         /*  Global Frame Count Reload Register B (DMGFR) */
};

/*  Handles  */
DMA_Handle hDma1;
```

The source file contains the Handle and Configuration Pre-Initialization using the CSL DMA API functions, DMA_open() and DMA_config() (see Example 6–3).

These two functions are encapsulated in a unique function, CSL_cfgInit(), which is called from your main C file. DMA_open() and DMA_config() are generated only if Open Handle to DMA and Enable pre-initialization (with a selected configuration other than DMA_NOTHING) are checked under the DMA Resource Manager Properties page.

Example 6–3. DMA Source File (Body Section)

```
void CSL_cfgInit()
{
    hDma1 = DMA_open(DMA_CHA1, DMA_OPEN_RESET);
    DMA_config(hDma1, &dmaCfg0);
    DMA_globalConfig(0x0FFFF, &gDMAConfig0);
}
```

6.6 Examples

The following CSL DMA initialization examples are provided under the directories:

c:\ti\examples\<target>\csl\manual_config\dma1a, dma1b, dma2, dma3, dma4

Example 1A	DMA channel initialization using DMA_config()
Example 1B	DMA channel initialization using DMA_configArgs()
Example 2	DMA channel auto-initialization with interrupt on transfer completion using DMA_config(). This example also illustrates the usage of globalConfig() to configure DMA global registers.
Example 3	DMA channel data transfer from/to MCBSP.
Example 4	DMA channel data transfer from/to MCBSP in ABU and digital loopback mode.

For illustration purposes, Example 1A is covered in detail below, and is illustrated in Figure 6–5.

Example 1A explains how DMA Channel 0 is initialized to transfer the data table at 0x3000@data space to 0x2000@data space. This example does not use any DMA global registers resources. Basic initialization values are as follows:

- ❏ Source address: 2000h in data space
- ❏ Destination address: 3000h in data space
- ❏ Transfer size: 10h words single words

The following two macros are used to create the initialization values for DMMCR and DMSFC respectively:

DMA_DMMCR_RMK(autoinit, dinm, imod, ctmod, sind, dms, dind, dmd)
0 0 0 0 1 1 1 1

DMA_DMSFC_RMK(dsyn, dblw, framecount)
0 0 0 (single-frame, Nframes-1)

The settings needed for the DMMCR are:

```
DMMCR0 = 0x0145u
#0000000100000101b
```

```
;0~~~~~~~~~~~~~~~ (AUTOINIT)   Autoinitialization disabled

;~0~~~~~~~~~~~~~~ (DINM)       Interrupts masked

;~~0~~~~~~~~~~~~~ (IMOD)       N/A

;~~~0~~~~~~~~~~~~ (CTMOD)      Multi-frame mode

;~~~~0~~~~~~~~~~~              Reserved

;~~~~~001~~~~~~~~ (SIND)       Post increment source
                                address

;~~~~~~~~01~~~~~~ (DMS)        Source in data space

;~~~~~~~~~~0~~~~~              Reserved

;~~~~~~~~~~~001~~ (DIND)       Post increment destination
                                address

;~~~~~~~~~~~~~~01 (DMD)        Destination in data space
```

The settings needed for the DMSFC are:

```
DMSFC0 = 0x0000u

#0000000000000000b

;0000~~~~~~~~~~~~ (DSYN)        No sync event

;~~~~0~~~~~~~~~~~ (DBLW)        Single-word mode

;~~~~~000~~~~~~~~               Reserved

;~~~~~~~~00000000 (Frame Count) FrameCount = 0h
                                (one frame)
```

Figure 6–5. DMA Channel Initialization Using DMA_config()

```
#include <csl_dma.h>

/*-------------------------------------------------------------------------*/
/* create a data from DMA transfer */

#define N       16

/* Place data in separate section to ensure placement */
/* within the DMA memory space defined for the device.*/
/* The address ranges chosen for this example in the  */
/* linker command file place src and dst within the   */
/* DMA memory map for the TMS320C5402. When modifying */
/* this example to run on a different C54x target,    */
/* please check the datasheet for your specific       */
/* device to make sure that the src and dst addresses */
/* for your transfer are assigned to a valid DMA      */
/* memory space.                                      */
```

Figure 6–5. DMA Channel Initialization Using DMA_config() (Continued)

```
#pragma DATA_SECTION(src,”dmaMem”)
Uint16 src[N];

#pragma DATA_SECTION(dst, ”dmaMem”)
Uint16 dst[N];


/* In this example, we will be effecting a DMA   */
/* transfer from DATA to DATA space in internal  */
/* memory. The DMA will operate in multi–frame   */
/* mode and we will poll for DMA operation       */
/* complete.                                     */


/* These are the settings we need for the DMA    */
/* mode control register, DMMCR, in order to     */
/* perform the transfer                          */


/*  DMMCR0 = 0x0145u                                    */
/*  #0000000101000101b                                  */
/*  ;0~~~~~~~~~~~~~~ (AUTOINIT)  No Autoinit            */
/*  ;~0~~~~~~~~~~~~~ (DINM)      No Interrupts          */
/*  ;~~0~~~~~~~~~~~~ (IMOD)      N/A                    */
/*  ;~~~0~~~~~~~~~~~ (CTMOD)     Multi–frame on         */
/*  ;~~~~0~~~~~~~~~~ (SLAXS)     Src not in extended mem */
/*  ;~~~~~001~~~~~~~~ (SIND)     Src addr Post–incr     */
/*  ;~~~~~~~~01~~~~~~ (DMS)      Src in data space      */
/*  ;~~~~~~~~~~0~~~~~ (DLAXS)    Dst not in extended mem */
/*  ;~~~~~~~~~~~001~~ (DIND)     Dst addr Post–incr     */
/*  ;~~~~~~~~~~~~~~01 (DMD)      Dst in data space      */

/* These are the settings required for DMA sync and      */
/* frame count register, DMSFC                           */

/*    DMSFC0 = 0x0000u                                   */
/*    #0000000000000000b                                 */
/*    ;0000~~~~~~~~~~~~ (DSYN)    No sync event          */
/*    ;~~~~0~~~~~~~~~~~ (DBLW)    Single–word mode       */
/*    ;~~~~~000~~~~~~~~           N/A                    */
/*    ;~~~~~~~~00000000 (Frame Count) = 0 (one frame)    */

/* Create a DMA configuration structure for the transfer  */
/* using predefined CSL macros and symbolic constants     */
```

Figure 6–5. DMA Channel Initialization Using DMA_config() (Continued)

```
DMA_Config  myconfig = {
  1,                                  /* Set Priority */
  0,                                  /* Autoix off   */
  DMA_DMMCR_RMK(
    DMA_DMMCR_AUTOINIT_OFF,
    DMA_DMMCR_DINM_OFF,
    DMA_DMMCR_IMOD_FULL_ONLY,
    DMA_DMMCR_CTMOD_MULTIFRAME,
    DMA_DMMCR_SLAXS_OFF,
    DMA_DMMCR_SIND_POSTINC,
    DMA_DMMCR_DMS_DATA,
    DMA_DMMCR_DLAXS_OFF,
    DMA_DMMCR_DIND_POSTINC,
    DMA_DMMCR_DMD_DATA
  ),                                     /* DMMCR */
  DMA_DMSFC_RMK(
    DMA_DMSFC_DSYN_NONE,
    DMA_DMSFC_DBLW_OFF,
    DMA_DMSFC_FRAMECNT_OF(0)
  ),                                     /* DMSFC */
  (DMA_AdrPtr) &src[0],                  /* DMSRC */
  (DMA_AdrPtr) &dst[0],                  /* DMDST */
  (Uint16)(N–1),                         /* DMCTR */
  0,                                     /* DMGSA */
  0,                                     /* DMGDA */
  0,                                     /* DMGCR */
  0,                                     /* DMGFR */
};

/*------------------------------------------------------------------------*/
void main() {
  Uint16 i;

  /* Initialize CSL library, this step is required */
  CSL_init();

  /* Set Src values and Clear destination */
  for (i=0; i<= N–1; i++) {
    src[i] = 0xBEEF;
    dst[i] = 0;
  }
}

/*------------------------------------------------------------------------*/
void taskFunc() {
```

Figure 6–5. DMA Channel Initialization Using DMA_config() (Continued)

```
    DMA_Handle myhDma;
    /* Create a DMA handle pointer */
    Uint16 err = 0;
    Uint16 i;

    LOG_printf(&LogMain,”<DMA1A>”);


  /* Open DMA channel 0, to use for this transfer */

  myhDma = DMA_open(DMA_CHA0, 0);

  /* Call DMA_config function to write your configuration */
  /* values to DMA channel control registers             */
  DMA_config(myhDma, &myconfig);

  /* Call DMA_start to begin the data transfer */
  DMA_start(myhDma);

  /* Poll DMA status too see if its done       */
  while(DMA_getStatus(myhDma));

  /* Check the values to make sure DMA transfer is */
  /* correct.                                      */
  for (i = 0; i <= N–1; i++) {
    if (dst[i] != 0xBEEFu) {
      ++err;
    }
  }

  /* We are done, so close DMA channel */
  DMA_close(myhDma);

  LOG_printf(&LogMain,”%s”,err?”DMA Example 1A FAILED”:”DMA Example 1A PASSED”);
  LOG_printf(&LogMain,”<DONE>”);
}
```

Chapter 7

EBUS Module

This chapter describes the configuration structure, functions, and macros used in the external bus interface (EBUS) module.

Topic **Page**

7.1 Overview

The EBUS module provides a configuration structure, functions, macros, and constants that allow you to control the external bus interface through the CSL.

Table 7–1 summarizes the configuration structure. Table 7–2 lists the EBUS functions.

Use the following guidelines for the EBUS functions:

- You can perform configuration by calling either EBUS_config(), EBUS_configArgs(), or any of the SET register macros.
- Because EBUS_config() and EBUS_configArgs() initialize all three external bus control registers, macros are provided to enable efficient access to individual registers when you need to set only one or two.
- The recommended approach is to initialize the external bus by using EBUS_config() with the EBUS_Config structure.

Table 7–1. EBUS Configuration Structure

Structure	Purpose	See page...
EBUS_Config	EBUS configuration structure used to setup the EBUS interface	7-3

Table 7–2. EBUS Functions

Function	Purpose	See page ...
EBUS_config()	Sets up EBUS using configuration structure (EBUS_Config)	7-4
EBUS_configArgs()	Sets up EBUS using register values passed to the function	7-5

7.2 Configuration Structure

This section describes the configuration structure that you can use to set up the EBUS interface.

EBUS_Config *EBUS configuration structure used to set up EBUS interface*

Structure EBUS_Config

Members

For 544x devices:

Uint16 bscr	Bank-switching control register

For other C54x devices:

Uint16 swwsr	Software wait-state register
Uint16 bscr	Bank-switching control register
Uint16 swcr	Sofware wait-state control register

Description The EBUS configuration structure is used to set up the EBUS Interface. You create and initialize this structure and then pass its address to the EBUS_config() function. You can use literal values or the *EBUS_REG_RMK* macros to create the structure member values.

Example

```
EBUS_Config Config1 = {
   0x7FFF, /* swwsr */
   0xF800, /* bscr */
   0x0000 /* swcr */
};
```

7.3 Functions

This section describes the EBUS API functions.

EBUS_config — *Writes value to set up EBUS using configuration structure*

Function

```
void EBUS_config(
    EBUS_Config *Config
);
```

Arguments

Config — Pointer to an initialized configuration structure

Return Value

None

Description

Writes a value to set up the EBUS using the configuration structure. The values of the structure are written to the port registers (see also EBUS_configArgs() and EBUS_Config).

Example

```
EBUS_Config MyConfig = {
   0x7FFF, /* swwsr */
   0xF800, /* bscr  */
   0x0000  /* swcr  */
};
   ...
   EBUS_config(&MyConfig);
```

EBUS_configArgs *Writes to EBUS using register values passed to the function*

Function

For C544x devices:

```
EBUS_configArgs (Uint16 bscr);
```

For other C54x devices:

```
void EBUS_configArgs(
    Uint16 swwsr,
    Uint16 bscr,
    Uint16 swcr
);
```

Arguments

swwsr	Software wait-state register
bscr	Bank-switching control register
swcr	Software wait-state control register

Return Value

None

Description

Writes to the EBUS using the register values passed to the function. The register values are written to the EBUS registers.

You may use literal values for the arguments; or for readability, you may use the EBUS_REG_RMK macros to create the register values based on field values.

Example

```
EBUS_configArgs (
 0x7FFF, /* swwsr */
 0xF800, /* bscr */
 0x0000 /* swcr */
);
```

7.4 Macros

As covered in section 1.3, CSL offers a collection of macros to gain individual access to the EBUS peripheral registers and fields.

Table 7–3 contains a list of macros available for the EBUS module. To use them, include “csl_ebus.h.”

Table 7–3. EBUS Macros

(a) Macros to read/write EBUS register values

Macro	Syntax
EBUS_RGET()	Uint16 EBUS_RGET(REG)
EBUS_RSET()	void EBUS_RSET(REG, Uint16 regval)

(b) Macros to read/write EBUS register field values (Applicable only to registers with more than one field)

Macro	Syntax
EBUS_FGET()	Uint16 EBUS_FGET(REG, FIELD)
EBUS_FSET()	Void EBUS_FSET(REG,FIELD, Uint16 fieldval)

(c) Macros to read/write EBUS register field values (Applicable only to registers with more than one field)

Macro	Syntax
EBUS_REG_RMK()	Uint16 EBUS_REG_RMK(fieldval_n,...fieldval_0) Note: *Start with field values with most significant field positions: field_n: MSB field field_0: LSB field * only writeable fields allowed
EBUS_FMK()	Uint16 EBUS_FMK(REG, FIELD, fieldval)

(d) Macros to read a register address

Macro	Syntax
EBUS_ADDR()	Uint16 EBUS_ADDR(REG)

Where:

REG: SWWSR (except in C544x), SWCR (except in C544x), BSCR

FIELD: register field name as specified in Appendix A.

- For _FSET and _FMK, field should be a writable field.
- _FGET, this field should, at least, be readable.

regVal: value to write in register REG

fieldVal: value to write in field FIELD of register REG. Rules to follow:

- Only writable fields are allowed.
- Value should be a right-justified constant. If fieldval_n value exceeds the number of bits allowed for that field, fieldval_n is truncated accordingly.

For examples on how to use macros, refer to macro sections 6.4 (DMA) and 11.4 (McBSP).

Chapter 8

GPIO Module

The GPIO module is designed to allow central control of non-multiplexed GPIO pins available in the C54x devices. (C544x devices only)

Topic **Page**

8.1 Overview

The GPIO module is designed to allow central control of the four non-multiplexed GPIO pins (GPIO 0 to GPIO 3) available in each of the C54x core devices (C544x devices only).

Functions that allow you to manipulate the C54x GPIO pins are listed in Table 8–1.

Macros that allows access to registers have been provided on page 8-5 (see Table 8–2).

Table 8–1. GPIO Functions

Function	Description	See page...
GPIO _pinDirection	Sets the GPIO pins as either an input or output pin	8-3
GPIO_pinRead	Reads the GPIO pin value	8-3
GPIO_pinWrite	Writes a value to a GPIO pin	8-4

8.2 Functions

The following are functions available for use with the GPIO module.

GPIO_pinDirection *Sets the GPIO pin as input or output*

Function int GPIO_pinDirection (Uint32 pinId, Uint16 direction);

Arguments

pinId IDs of the GPIO pins to enable. It will have one one of the following values:
GPIO_PIN0
...
GPIO_PIN3

Direction pin direction
GPIO_INPUT //input
GPIO_OUTPUT //output

Return Value None

Description Sets the direction for a general-purpose I/O pin (input or output)

Example

```
/* sets the pin gpio1 as an input */
GPIO_pinDirection (GPIO_PIN1, GPIO_INPUT);
```

GPIO_pinRead *Reads the GPIO pin value*

Function int GPIO_pinRead (Uint32 pinId);

Arguments

pinId IDs of the GPIO pins to enable. It will have one of the following values:
GPIO_PIN0
...
GPIO_PIN3

Return Value Value Value read in GPIO pin (1 or 0)

Description Reads the value in a general purpose input pin.

Example

```
int val;
val = GPIO_pinRead (GPIO_PIN3);
```

GPIO_pinWrite *Writes a value to a GPIO pin*

Function int GPIO_pinWrite (Uint32 pinId, Unit16 val);

Arguments

pinId IDs of the GPIO pins to enable. It will have one of the following values:
GPIO_PIN0
...
GPIO_PIN3

val Value to write to the pinID.

Return Value None

Description Writes a value to a general purpose output pin.

Example

```
GPIO_pinWrite (GPIO_PIN1, 1);  /* sets iopin1 to ”1” */
```

8.3 Macros

As covered in section 1.3, CSL offers a collection of macros to gain individual access to a GPIO specific register (GPIO) in C544x devices.

Table 8–2 contains a list of macros available for the GPIO module. To use them, include “csl_gpio.h.”

Table 8–2. GPIO Macros (C544x devices only)

(a) Macros to read/write GPIO register values

Macro	Syntax
GPIO_RGET()	Uint16 GPIO_RGET(REG)
GPIO_RSET()	void GPIO_RSET(REG, Uint16 regval)

(b) Macros to read/write GPIO register field values (Applicable only to registers with more than one field)

Macro	Syntax
GPIO_FGET()	Uint16 GPIO_FGET(REG, FIELD)
GPIO_FSET()	Void GPIO_FSET(REG,FIELD, Uint16 fieldval)

(c) Macros to read/write GPIO register field values (Applicable only to registers with more than one field)

Macro	Syntax
GPIO_REG_RMK()	Uint16 GPIO_REG_RMK(fieldval_n,...fieldval_0) Note: *Start with field values with most significant field positions: field_n: MSB field field_0: LSB field * only writable fields allowed
GPIO_FMK()	Uint16 GPIO_FMK(REG, FIELD, fieldval)

(d) Macros to read a register address

Macro	Syntax
GPIO_ADDR()	Uint16 GPIO_ADDR(REG)

Where:

REG: GPIO

FIELD: register field name as specified in Appendix A.

- For _FSET and _FMK, field should be a writable field.
- _FGET, this field should, at least, be readable.

regVal: value to write in register REG

fieldVal: value to write in field FIELD of register REG. Rules to follow:

- Only writable fields are allowed.
- Value should be a right-justified constant. If fieldval_n value exceeds the number of bits allowed for that field, fieldval_n is truncated accordingly.

For examples on how to use macros, refer to the macro sections 6.4 (DMA) and 11.4 (McBSP).

8.4 Configuring the GPIO Module Using CSL GUI

The GPIO module facilitates configuration/control of the General-Purpose I/O on the C54x. The module consists of a configuration manager. The configuration manager allows you to configure the directions to either the input or output of the GPIO pins.

Figure 8–1 illustrates the GPIO sections menu on the CSL graphical user interface (GUI)

Figure 8–1. GPIO Sections Menu

The Non-Multiplexed GPIO includes the following section:

Non-Multiplexed GPIO Configuration Manager allows you to configure the GPIO Pin directions.

8.4.1 Non-Multiplexed GPIO Configuration Manager

The Non-Multiplexed GPIO Configuration Manager allows you to configure the GPIO pin directions.

8.4.1.1 Properties Pages of the Non-Multiplexed GPIO Configuration

The Properties pages allow you to set the Peripheral registers related to the GPIO. The configuration options are divided into the following Tab page:

Settings allows you to configure the Input/Output settings of GPIO Pins.

Figure 8–2 depicts the Properties Page dialog box.

Figure 8–2. GPIO Properties Page

The Settings Tab is composed of several options that are set to a default value (at device reset).

The options represent the fields of the GPIO register direction. For further details of the fields and registers, refer to the GPIO section of the *TMS320C54x DSP Enhanced Peripherals Reference Set* (literature number SPRU302).

8.4.2 C Code Generation for GPIO Module

Two C files are generated from the configuration tool:

- ❑ Header file
- ❑ Source file

8.4.2.1 Header File

The header file includes all the csl header files of the modules.

8.4.2.2 Source File

The source file contains the GPIO Register set macro invocation. This macro invocation is encapsulated in a unique function, CSL_cfgInit(), which is called from your main C file.

GPIO_RSET() will be generated only if Configure Non-Multiplexed GPIO is checked under the Non-multiplexed GPIO Configuration Properties page. See Figure 8–2.

Example 8–1. GPIO Source File (Body Section)

```
void CSL_cfgInit()
{

    GPIO_RSET(GPIO, 3840);
}
```

Chapter 9

HPI Module

This chapter contains descriptions for macros available in the HPI module.

Topic **Page**

9.1 Macros

As covered in section 1.3, the CSL offers a collection of macros to gain individual access to the peripheral registers and fields.

Table 9–1 contains a list of macros available for the HPI module. To use them, include “csl_hpi.h.”

Table 9–1. HPI Macros

(a) Macros to read/write HPI register values

Macro	Syntax
HPI_RGET()	Uint16 HPI_RGET(REG)
HPI_RSET()	void HPI_RSET(REG, Uint16 regval)

(b) Macros to read/write HPI register field values (Applicable only to registers with more than one field)

Macro	Syntax
HPI_FGET()	Uint16 HPI_FGET(REG, FIELD)
HPI_FSET()	Void HPI_FSET(REG,FIELD, Uint16 fieldval)

(c) Macros to read/write HPI register field values (Applicable only to registers with more than one field)

Macro	Syntax
HPI_REG_RMK()	Uint16 HPI_REG_RMK(fieldval_n,...fieldval_0) Note: ❑ Start with field values with most significant field positions: field_n: MSB field field_0: LSB field ❑ only writable fields allowed
HPI_FMK()	Uint16 HPI_FMK(REG, FIELD, fieldval)

(d) Macros to read a register address

Macro	Syntax
HPI_ADDR()	Uint16 HPI_ADDR(REG)

Where:

REG: include HPIC, GPIOCR, GPIOSR

FIELD: register field name as specified in Appendix A.

- For _FSET and _FMK, field should be a writable field.
- For _FGET, this field should, at least, be readable.

regVal: value to write in register REG

fieldVal: value to write in field FIELD of register REG. Rules to follow:

- Only writable fields are allowed
- Value should be a right-justified constant. If fieldval_n value exceeds the number of bits allowed for that field, fieldval_n is truncated accordingly.

For examples on how to use macros, refer to macro sections 6.4 (DMA) and 11.4 (McBSP).

Chapter 10

IRQ Module

The IRQ module provides an easy to use interface for enabling/disabling and managing interrupts.

Topic **Page**

10.1 Overview

The IRQ module provides an interface for managing peripheral interrupts to the CPU. This module provides the following functionality:

- Masking an interrupt in the IMR_X register.
- Polling for the interrupt status from the IFR_X register.
- Setting the interrupt vector table address and placing the necessary code in the interrupt vector table to branch to a user-defined interrupt service routine (ISR).
- Enabling/Disabling Global Interrupts in the ST1 (INTM) bit.
- Reading and writing to parameters in the DSP/BIOS dispatch table. (When the DPS BIOS dispatcher option is enabled in DSP BIOS.)

  The DSP BIOS dispatcher is responsible for dynamically handling interrupts and maintains a table of ISRs to be executed for specific interrupts. The IRQ module has a set of APIs that update the dispatch table.

Table 10–1 lists the IRQ configuration structure.

Table 10–2 lists the functions available in the IRQ module.

Table 10–2(a) and (b) list the primary and auxiliary IRQ functions.

Table 10–2(c) lists the API functions that enable DSP/BIOS dispatcher communication.

The IRQ functions in Table 10–2(a) can be used with or without DSP/BIOS; however, if DSP/BIOS is present, do not disable interrupts for long periods of time, as this could disrupt the DSP/BIOS environment.

Table 10–2(b) lists the only API functions that cannot be used when DSP/BIOS dispatcher is present or DSP/BIOS HWI module is used to configure the interrupt vectors. This function, IRQ_plug(), dynamically places code at the interrupt vector location to branch to a user-defined ISR for a specified event. If you call IRQ_plug() when DSP/BIOS dispatcher is present or HWI module has been used to configure interrupt vectors, this could disrupt the DSP/BIOS operating environment.

Table 10–2(c) lists the API functions that enable DSP/BIOS dispatcher communications. These functions should be used only when DSP/BIOS is present **and** the DSP/BIOS dispatcher is enabled.

Table 10–3 lists all IRQ logical interrupt events for this module.

Table 10–1. IRQ Configuration Structure

Structure	Purpose	See page ...
IRQ_Config	IRQ structure that contains all local registers required to set up a specific IRQ channel.	10-8

Table 10–2. IRQ Functions

(a) Primary Functions

Function	Purpose	See page ...
IRQ_clear()	Clears the interrupt flag in the IFR register for the specified event.	10-9
IRQ_disable()	Disables the specified event in the IMR register.	10-10
IRQ_enable()	Enables the specified event in the IMR register flag.	10-11
IRQ_globalDisable()	Globally disables all maskable interrupts. (INTM = 1)	10-12
IRQ_globalEnable()	Globally enables all maskable interrupts. (INTM = 0)	10-13
IRQ_globalRestore()	Restores the status of global interrupt enable/disable (INTM).	10-13
IRQ_restore()	Restores the status of the specified event.	10-15
IRQ_setVecs()	Sets the base address of the interrupt vector table.	10-16
IRQ_test()	Polls the interrupt flag in IFR register the specified event.	10-16

(b) Auxiliary Functions

Function	Purpose	See page ...
IRQ_plug()	Writes the necessary code in the interrupt vector location to branch to the interrupt service routine for the specified event. **Caution:** Do not use this function when DSP/BIOS is present and the dispatcher is enabled.	10-14

Table 10–2. IRQ Functions (Continued)

(c) DSP/BIOS Dispatcher Communication Functions

Function	Purpose	See page ...
IRQ_config()	Updates the DSP/BIOS dispatch table with a new configuration for the specified event.	10-9
IRQ_configArgs()	Updates the DSP/BIOS dispatch table with a new configuration for the specified event.	10-10
IRQ_getArg()	Returns value of the argument to the interrupt service routine that the DSP/BIOS dispatcher passes when the interrupt occurs.	10-11
IRQ_getConfig()	Returns current DSP/BIOS dispatch table entries for the specified event.	10-12
IRQ_map()	Maps a logical event to its physical interrupt.	10-14
IRQ_setArg()	Sets the value of the argument for DSP/BIOS dispatch to pass to the interrupt service routine for the specified event.	10-15

10.1.1 The Event ID Concept

The IRQ module assigns an event ID to each of the possible physical interrupts. Because there are more events possible than can be masked in the IMR register, many of the events share a common physical interrupt. Therefore, it is necessary in some cases to map the logical events to the corresponding physical interrupt. The IRQ module defines a set of constants IRQ_EVT_NNNN that uniquely identify each of the possible logical interrupts. A list of these event IDs is listed in Table 10–3. All of the IRQ APIs operate on logical events.

Table 10–3. IRQ_EVT_NNNN Event List

Constant	Purpose
IRQ_EVT_RS	Reset
IRQ_EVT_SINTR	Software Interrupt
IRQ_EVT_NMI	Non-Maskable Interrupt (NMI)
IRQ_EVT_SINT16	Software Interrupt #16
IRQ_EVT_SINT17	Software Interrupt #17
IRQ_EVT_SINT18	Software Interrupt #18

Table 10–3. IRQ_EVT_NNNN Event List (Continued)

Constant	Purpose
IRQ_EVT_SINT19	Software Interrupt #19
IRQ_EVT_SINT20	Software Interrupt #20
IRQ_EVT_SINT21	Software Interrupt #21
IRQ_EVT_SINT22	Software Interrupt #22
IRQ_EVT_SINT23	Software Interrupt #23
IRQ_EVT_SINT24	Software Interrupt #24
IRQ_EVT_SINT25	Software Interrupt #25
IRQ_EVT_SINT26	Software Interrupt #26
IRQ_EVT_SINT27	Software Interrupt #27
IRQ_EVT_SINT28	Software Interrupt #28
IRQ_EVT_SINT29	Software Interrupt #29
IRQ_EVT_SINT30	Software Interrupt #30
IRQ_EVT_SINT0	Software Interrupt #0
IRQ_EVT_SINT1	Software Interrupt #1
IRQ_EVT_SINT2	Software Interrupt #2
IRQ_EVT_SINT3	Software Interrupt #3
IRQ_EVT_SINT4	Software Interrupt #4
IRQ_EVT_SINT5	Software Interrupt #5
IRQ_EVT_SINT6	Software Interrupt #6
IRQ_EVT_SINT7	Software Interrupt #7
IRQ_EVT_SINT8	Software Interrupt #8
IRQ_EVT_SINT9	Software Interrupt #9
IRQ_EVT_SINT10	Software Interrupt #10
IRQ_EVT_SINT11	Software Interrupt #11
IRQ_EVT_SINT12	Software Interrupt #12
IRQ_EVT_SINT13	Software Interrupt #13

Table 10–3. IRQ_EVT_NNNN Event List (Continued)

Constant	Purpose
IRQ_EVT_INT0	External User Interrupt #0
IRQ_EVT_INT1	External User Interrupt #1
IRQ_EVT_INT2	External User Interrupt #2
IRQ_EVT_INT3	External User Interrupt #3
IRQ_EVT_TINT0	Timer 0 Interrupt
IRQ_EVT_HINT	Host Interrupt (HPI)
IRQ_EVT_DMA0	DMA Channel 0 Interrupt
IRQ_EVT_DMA1	DMA Channel 1 Interrupt
IRQ_EVT_DMA2	DMA Channel 2 Interrupt
IRQ_EVT_DMA3	DMA Channel 3 Interrupt
IRQ_EVT_DMA4	DMA Channel 4 Interrupt
IRQ_EVT_DMA5	DMA Channel 5 Interrupt
IRQ_EVT_RINT0	MCBSP Port #0 Receive Interrupt
IRQ_EVT_XINT0	MCBSP Port #0 Transmit Interrupt
IRQ_EVT_RINT2	MCBSP Port #2 Receive Interrupt
IRQ_EVT_XINT2	MCBSP Port #2 Transmit Interrupt
IRQ_EVT_TINT1	Timer #1 Interrupt
IRQ_EVT_HPINT	Host Interrupt (HPI)
IRQ_EVT_RINT1	MCBSP Port #1 Receive Interrupt
IRQ_EVT_XINT1	MCBSP Port #1 Transmit Interrupt
IRQ_EVT_IPINT	FIFO Full Interrupt
IRQ_EVT_SINT14	Software Interrupt #14
IRQ_EVT_WDTINT	Watchdog Timer Interrupt
IRQ_EVT_UART	UART Interrupt
IRQ_EVT_DAARCV	DAA Receive Interrupt
IRQ_EVT_DAAXMT	DAA Transmit Interrupt

10.2 Using Interrupts with CSL

Interrupts within CSL can be managed in the following methods:

- ❑ Manual setting outside DSPBIOS HWIs
- ❑ Using DSPBIOS HWIs
- ❑ Using DSPBIOS Dispatcher

Example 10–1. Manual Setting Outside DSPBIOS HWIs

```
#define NVECTORS       256

#pragma DATA_SECTION   (myIvtTable, ”myvec”)
int myIvtTable[NVECTORS];

; ...
interrupt void myIsr();

; ...
main (){

; ...
; Option 1: use Event IDs directly
; ...

IRQ_setVecs ((Uint16)myIvtTable);
IRQ_plug (IRQ_EVT_TINT0,&myIsr);
IRQ_enable(IRQ_EVT_TINT0);
IRQ_globalEnable();

; ...
; Option 2: Use the PER_getEventId() function (TIMER as an example)
for a better abstraction
; ...

IRQ_setVecs ((Uint16)myIvtTable);
eventId = TIMER_getEventId (hTimer);
IRQ_plug (eventId,&myIsr);
IRQ_enable (eventId);
IRQ_globalEnable();
; ...

}

interrupt void myISR()

{

//....;

}
```

10.3 Configuration Structure

IRQ_Config *IRQ configuration structure*

Structure IRQ_Config

Members

IRQ_IsrPtr FuncAddr	Function to be called
Uint16 funcArg	Argument to pass to ISR when invoked
Uint32 ierMask	Mask for the interrupts to be disabled when the current interrupt is handled.

Description This is the IRQ configuration structure used to update a DSP/BIOS table entry. You create and initialize this structure then pass its address to the IRQ_config( ) function.

Example

```
IRQ_Config MyConfig = {
  0x0000, /* funcAddr */
  0x0000, /* funcArg  */
  0x0300 /* ierMask  */
};
```

10.4 Functions

This section describes the primary, auxiliary, and DSP/BIOS Dispatcher Communications IRQ functions.

IRQ_clear — *Clears event flag from IFR register*

Function

```
void IRQ_clear(
    Uint16 EventId
);
```

Arguments

EventId — Event ID, see IRQ_EVT_NNNN (Table 10–3) for a complete list of events. Or, use the PER_getEventId() function to get the EventID.

Return Value

None

Description

Clears the event flag from the IFR register

Example

```
IRQ_clear(IRQ_EVT_TINT0);
```

IRQ_config — *Updates Entry in DSPBIOS dispatch table*

Function

```
void IRQ_config(
    Uint16 EventId,
    IRQ_Config *Config
);
```

Arguments

EventID — Event ID, see IRQ_EVT_NNNN for a complete list of events.

Config — Pointer to an initialized configuration structure

Return Value

None

Description

Updates the entry in the DSPBIOS dispatch table for the specified event.

Example

```
IRQ_Config MyConfig = {
  0x0000, /* funcAddr */
  0x0000, /* funcArg  */
  0x0300 /* ierMask  */
};
  ...
IRQ_config(IRQ_EVT_TINT0,&MyConfig);
```

IRQ_configArgs *Updates entry in DSPBIOS dispatch table*

Function

```
void IRQ_configArgs(
    Uint16 EventId,
    IRQ_IsrPtr funcAddr,
    Uint16 funcArg,
    Uint32 ierMask
);
```

Arguments

EventId	Event ID, see IRQ_EVT_NNNN for a complete list of events.
funcAddr	Interrupt service routine address
funcArg	Argument to pass to interrupt service routine when it is invoked by DSPBIOS dispatcher
ierMask	Interrupts to disable while processing the ISR for this event (Mask for IER0, IER1)

Return Value None

Description Updates DSPBIOS dispatch table entry for the specified event.

Example

```
IRQ_configArgs(EventID, funcAddr, funcArg, ierMask);
```

IRQ_disable *Disables specified event*

Function

```
int IRQ_disable(
    Uint16 EventId
);
```

Arguments

EventId	Event ID, see IRQ_EVT_NNNN (Table 10-3) for a complete list of events. Or, use the PER_getEventId() function to get the EventID.

Return Value intm Old value of the specified mask bit in the IMR register.

Description Disables the specified event, by modifying the IMR register.

Example

```
int intm;
intm = IRQ_disable(IRQ_EVT_TINT0);
```

IRQ_enable — *Enables specified event*

Function

```
int IRQ_enable(
    Uint16 EventId
);
```

Arguments

EventId — Event ID, see IRQ_EVT_NNNN (Table 10-3) for a complete list of events. Or, use the PER_getEventId() function to get the EventID.

Return Value

intm — Old value of the specified mask bit in the IMR register.

Description

Enables the specified event.

Example

```
int intm;
intm = IRQ_enable(IRQ_EVT_TINT0);
```

IRQ_getArg — *Gets value for specified event*

Function

```
Uint32 IRQ_getArg(
    Uint16 EventId
);
```

Arguments

EventId — Event ID, see IRQ_EVT_NNNN (Table 10-3) for a complete list of events. Or, use the PER_getEventId() function to get the EventID.

Return Value

Value of argument

Description

Gets the value for specified event.

Example

```
Uint32 arg;
arg = IRQ_getArg(IRQ_EVT_TINT0);
```

IRQ_getConfig *Gets DSP/BIOS dispatch table entry*

Function

```
void IRQ_getConfig(
    Uint16 EventId,
    IRQ_Config *Config
);
```

Arguments

EventId — Event ID, see IRQ_EVT_NNNN (Table 10–3) for a complete list of events. Or, use the PER_getEventId() function to get the EventID.

Config — Pointer to configuration structure

Return Value

None

Description

Reads the current configuration for the IRQ event.

Example

```
IRQ_Config ConfigRead;
...
IRQ_getConfig(IRQ_EVT_TINT0, &ConfigRead);
```

IRQ_globalDisable *Globally Disables Interrupts*

Function

```
int IRQ_globalDisable(
);
```

Arguments

None

Return Value

intm — Returns the old INTM value

Description

This function globally disables interrupts by setting the INTM of the ST1 register. The old value of INTM is returned. This is useful for temporarily disabling global interrupts, then enabling them again.

Example

```
int oldgie;
oldgie = IRQ_globalDisable();
```

IRQ_globalEnable *Globally enables all events*

Function

```
int IRQ_globalEnable(
);
```

Arguments

None

Return Value

intm Returns the old INTM value

Description

This function globally enables all interrupts by setting the INTM of the ST1 register. The old value of INTM is returned. This is useful for temporarily enabling global interrupts, then disabling them again.

Example

```
Uint32 intm;
intm = IRQ_globalEnable();
...
IRQ_globalRestore (intm);
```

IRQ_globalRestore *Restores The Global Interrupt Mask State*

Function

```
void IRQ_globalRestore(
    int intm
);
```

Arguments

intm Value to restore the INTM value to (0 = enable, 1 = disable)

Return Value

gie Previously saved value

Description

This function restores the INTM state to the value passed in by writing to the INTM bit of the ST1 register. This is useful for temporarily disabling/enabling global interrupts, then restoring them back to its previous state.

Example

```
int intm;
intm = IRQ_globalDisable();
...
IRQ_globalRestore (intm);
```

IRQ_map — *Maps Event To Physical Interrupt Number*

Function

```
void IRQ_map(
    Uint16 EventId
);
```

Arguments

EventId — Event ID, see IRQ_EVT_NNNN (Table 10–3) for a complete list of events. Or, use the PER_getEventId() function to get the EventID.

Return Value

None

Description

This function maps a logical event to a physical interrupt number for use by DSPBIOS dispatch.

Example

```
IRQ_map(IRQ_EVT_TINT0);
```

IRQ_plug — *Initializes An Interrupt Vector Table Vector*

Function

```
void IRQ_plug(
    Uint16 EventId,
    IRQ_IsrPtr funcAddr,
);
```

Arguments

EventId — Event ID, see IRQ_EVT_NNNN (Table 10–3) for a complete list of events. Or, use the PER_getEventId() function to get the EventID.

funcAddr — Address of the interrupt service routine to be called when the interrupt happens. This function must be C-callable and if implemented in C, it must be declared using the *interrupt* keyword.

Return Value

None

Description

Initializes an interrupt vector table vector with the necessary code to branch to the specified ISR. **Caution:** Do not use this function when DSP/BIOS is present and the dispatcher is enabled.

Example

```
IRQ_IsrPtr funcAddr;
.
.
.
IRQ_plug (IRQ_EVT_TINT0, funcAddr);
```

IRQ_restore *Restores the status of the specified event*

Function

```
void IRQ_restore(
    Uint16 EventId,
    Uint16 Val
);
```

Arguments

EventId	Event ID, see IRQ_EVT_NNNN (Table 10–3 on page 10-4) for a complete list of events.
Val	Value to restore the specified event to

Return Value

None

Description

Restores the status of the specified event.

Example

```
Uint16 intm = IRQ_disable(IRQ_EVT_TINT0);
....
IRQ_restore(IRQ_EVT_TINT0,intm);
```

IRQ_setArg *Sets value of argument for DSPBIOS dispatch entry*

Function

```
void IRQ_setArg(
    Uint16    EventId,
    Uint32    val
);
```

Arguments

EventId	Event ID, see IRQ_EVT_NNNN (Table 10–3) for a complete list of events. Or, use the PER_getEventId() function to get the EventID.

Return Value

None

Description

Sets the argument that DSP/BIOS dispatcher will pass to the interrupt service routine for the specified event.

Example

```
Uint32 val;
IRQ_setArg(IRQ_EVT_TINT0, val);
```

IRQ_setVecs — *Sets the base address of the interrupt vectors*

Function

```
int IRQ_setVecs(
    Uint32 iptr
);
```

Arguments

iptr IVPD pointer to the DSP interrupt vector table

Return Value

oldVecs Returns the old IVPD Pointer to the DSP interrupt Vector table

Description

Use this function to set the base address of the interrupt vector table in the IVPD register.

Caution: Changing the interrupt vector table base can have adverse effects on your system because you will be effectively eliminating all previous interrupt settings. There is a strong chance that the DSP/BIOS kernel and RTDX will fail if this function is not used with care.

Example

```
IRQ_setVecs (0x8000);
```

IRQ_test — *Tests event to see if its flag is set in IFR register*

Function

```
CSLBool IRQ_test(
    Uint16 EventId
);
```

Arguments

EventId Event ID, see IRQ_EVT_NNNN (Table 10–3) for a complete list of events. Or, use the PER_getEventId() function to get the EventID.

Return Value

Event flag, 0 or 1

Description

Tests an event to see if its flag is set in the IFR register.

Example

```
while (!IRQ_test(IRQ_EVT_TINT0);
```

Chapter 11

McBSP Module

The chapter describes the structure, functions, and macros of the McBSP module.

Topic **Page**

11.1 Overview

THE McBSP is a handle-based module that requires you to call MCBSP_open() to obtain a handle before calling any other functions. Table 11–1 lists the structure for use with the McBSP modules. Table 11–2 lists the functions for use with the McBSP modules.

Table 11–1. McBSP Configuration Structure

Structure	Purpose	See page...
MCBSP_Config	McBSP configuration structure used to setup a McBSP port	11-4

Table 11–2. McBSP Functions

(a) Primary Functions

Function	Purpose	See page ...
MCBSP_close()	Closes a McBSP port	11-10
MCBSP_config()	Sets up the McBSP port using the configuration structure (MCBSP_Config)	11-10
MCBSP_configArgs()	Sets up the McBSP port using the register values passed to the function	11-12
MCBSP_open()	Opens a McBSP port	11-17
MCBSP_start()	Start a transmit and/or receive for a MCBSP port	11-20

Table 11–2. McBSP Functions (Continued)

(b) Auxiliary Functions

Function	Purpose	See page ...
MCBSP_getConfig()	Get MCBSP channel configuration	11-15
MCBSP_getPort()	Get MCBSP Port number used in given handle	11-15
MCBSP_read16()	Performs a direct 16-bit read from the data receive register DRR1	11-17
MCBSP_read32()	Performs two direct 16-bit reads: data receive register 2 DRR2 (MSB) and data receive register 1 DRR1 (LSB)	11-18
MCBSP_reset()	Resets the given serial port	11-18
MCBSP_rfull()	Reads the RFULL bit SPCR1 register	11-18
MCBSP_rrdy()	Reads the RRDY status bit of the SPCR1 register	11-19
MCBSP_write16()	Writes a 16-bit value to the serial port data transmit register, DXR1	11-22
MCBSP_write32()	Writes two 16-bit values to the two serial port data transmit registers, DXR2 (16-bit MSB) and DXR1 (16-bit LSB)	11-22
MCBSP_xempty()	Reads the XEMPTY bit from the SPCR2 register	11-23
MCBSP_xrdy()	Reads the XRDY status bit of the SPCR2 register	11-23

(c) Interrupt Control Functions

Function	Purpose	See page ...
MCBSP_getRcvEventId()	Retrieves the receive event ID for the given port	11-16
MCBSP_getXmtEventId()	Retrieves the transmit event ID for the given port	11-16

(d) Multichannel Control Functions

Function	Purpose	See page ...
MCBSP_channelDisable()	Disables one or several McBSP channels	11-6
MCBSP_channelEnable()	Enables one or several McBSP channels of the selected register	11-7
MCBSP_channelStatus()	Returns the channel status	11-9

11.2 Configuration Structure

This section lists the configuration structure for the McBSP module.

MCBSP_Config *McBSP configuration structure used to setup McBSP port*

Structure MCBSP_Config

Members

Uint16 spcr1	Serial port control register 1 value
Uint16 spcr2	Serial port control register 2 value
Uint16 rcr1	Receive control register 1 value
Uint16 rcr2	Receive control register 2 value
Uint16 xcr1	Transmit control register 1 value
Uint16 xcr2	Transmit control register 2 value
Uint16 srgr1	Sample rate generator register 1 value
Uint16 srgr2	Sample rate generator register 2 value
Uint16 mcr1	Multi-channel control register 1 value
Uint16 mcr2	Multi-channel control register 2 value
Uint16 pcr	Pin control register value

For devices supporting 128 channels:

Uint16 rcera	Receive channel enable register partition A value
Uint16 rcerb	Receive channel enable register partition B value
Uint16 rcerc	Receive channel enable register partition C value
Uint16 rcerd	Receive channel enable register partition D value
Uint16 rcere	Receive channel enable register partition E value
Uint16 rcerf	Receive channel enable register partition F value
Uint16 rcerg	Receive channel enable register partition G value
Uint16 rcerh	Receive channel enable register partition H value
Uint16 xcera	Transmit channel enable register partition A value
Uint16 xcerb	Transmit channel enable register partition B value
Uint16 xcerc	Transmit channel enable register partition C value
Uint16 xcerd	Transmit channel enable register partition D value
Uint16 xcere	Transmit channel enable register partition E value
Uint16 xcerf	Transmit channel enable register partition F value
Uint16 xcerg	Transmit channel enable register partition G value
Uint16 xcerh	Transmit channel enable register partition H value

For devices that do not support 128 channels:

Uint16 rcera
Uint16 rcerb
Uint16 xcera
Uint16 xcerb

Description

McBSP configuration structure used to setup a McBSP port. You create and initialize this structure 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 = {
  0x8001,  /* spcr1 */
  0x0001,  /* spcr2 */
  0x0000,  /* rcr1  */
  0x0000,  /* rcr2  */
  0x0000,  /* xcr1  */
  0x0000,  /* xcr2  */
  0x0001,  /* srgr1 */
  0x2000,  /* srgr2 */
  0x0000,  /* mcr1  */
  0x0000,  /* mcr2  */
  0x0000,  /* pcr   */
  0x0000,  /* rcera */
  0x0000,  /* rcerb */
  0x0000,  /* xcera */
  0x0000,  /* xcerb */

};
...
hMcbsp = MCBSP_open(MCBSP_PORT0, MCBSP_OPEN_RESET)
...
MCBSP_config(hMcbsp,&MyConfig);
```

11.3 Functions

This section lists the primary, auxiliary, interrupt control, and multi-channel functions available for use in the McBSP module.

MCBSP_channelDisable *Disables one or several McBSP channels*

Function

```
void MCBSP_channelDisable(
    MCBSP_Handle hMcbsp,
    Uint16 RegName,
    Uint16 Channels
    );
```

Arguments

hMcbsp — Handle to McBSP port obtained by MCBSP_open()

RegName — Receive and Transmit Channel Enable Registers:
RCERA
RCERB
XCERA
XCERB

For devices supporting 128 channels (see section 1.8) add:
RCERC
RCERD
RCERE
RCERF
RCERG
RCERH
XCERC
XCERD
XCERE
XCERF
XCERG
XCERH

Channels — Available values for the specific RegName are:
MCBSP_CHAN0
MCBSP_CHAN1
MCBSP_CHAN2
MCBSP_CHAN3
MCBSP_CHAN4

MCBSP_CHAN5
MCBSP_CHAN6
MCBSP_CHAN7
MCBSP_CHAN8
MCBSP_CHAN9
MCBSP_CHAN10
MCBSP_CHAN11
MCBSP_CHAN12
MCBSP_CHAN13
MCBSP_CHAN14
MCBSP_CHAN15

Return Value None

Description Disables one or several McBSP channels of the selected register. To disable several channels at the same time,the sign "|" OR has to be added in between.

To see if there is pending data in the receive or transmit buffers before disabling a channel, use MCBSP_rrdy() or MCBSP_xrdy().

Example

```
/* Disables Channel 0 of the partition A */
MCBSP_channelDisable(hMcbsp,RCERA, MCBSP_CHAN0);

/* Disables Channels  1, 2 and 8 of the partition B with "|"*/
MCBSP_channelDisable(hMcbsp,RCERB,(MCBSP_CHAN1 | MCBSP_CHAN2 |
MCBSP_CHAN8));
```

MCBSP_channelEnable *Enables one or several McBSP channels of selected register*

Function

```
void MCBSP_channelEnable(
    MCBSP_Handle hMcbsp,
    Uint16 RegName,
    Uint16 Channels
    );
```

Arguments

hMcbsp Handle to McBSP port obtained by MCBSP_open()

RegName Receive and Transmit Channel Enable Registers:
RCERA
RCERB
XCERA
XCERB

For devices supporting 128 channels (see section 1.8) add:
RCERC
RCERD
RCERE
RCERF
RCERG
RCERH
XCERC
XCERD
XCERE
XCERF
XCERG
XCERH

Channels — Available values for the specificReg Addr are:
MCBSP_CHAN0
MCBSP_CHAN1
MCBSP_CHAN2
MCBSP_CHAN3
MCBSP_CHAN4
MCBSP_CHAN5
MCBSP_CHAN6
MCBSP_CHAN7
MCBSP_CHAN8
MCBSP_CHAN9
MCBSP_CHAN10
MCBSP_CHAN11
MCBSP_CHAN12
MCBSP_CHAN13
MCBSP_CHAN14
MCBSP_CHAN15

Return Value None

Description Enables one or several McBSP channels of the selected register.

To enabling several channels at the same time, the sign "|" OR has to be added in between.

Example

```
/* Enables Channel 0 of the partition A */
MCBSP_channelEnable(hMcbsp,RCERA, MCBSP_CHAN0);
/* Enables Channel 1, 4 and 6 of the partition B  with  "|" */
MCBSP_channelEnable(hMcbsp,RCERB,(MCBSP_CHAN1|  MCBSP_CHAN4  |
MCBSP_CHAN6));
```

MCBSP_channelStatus *Returns channel status*

Function

```
Uint16 MCBSP_channelStatus(
        MCBSP_Handle hMcbsp,
        Uint16 RegName,
        Uint16 Channel
    );
```

Arguments

hMcbsp — Handle to McBSP port obtained by MCBSP_open()

RegName — Receive and Transmit Channel Enable Registers:
RCERA
RCERB
XCERA
XCERB

For devices supporting 128 channels (see section 1.8) add:
RCERC
RCERD
RCERE
RCERF
RCERG
RCERH
XCERC
XCERD
XCERE
XCERF
XCERG
XCERH

Channel — Selectable Channels for the specific RegName are:
MCBSP_CHAN0
MCBSP_CHAN1
MCBSP_CHAN2
MCBSP_CHAN3
MCBSP_CHAN4
MCBSP_CHAN5
MCBSP_CHAN6
MCBSP_CHAN7
MCBSP_CHAN8
MCBSP_CHAN9
MCBSP_CHAN10

MCBSP_CHAN11
CHAN12
CHAN13
CHAN14
CHAN15

Return Value Channel status 0 - Disabled
1 - Enabled

Description Returns the channel status by reading the associated bit into the selected register (RegName). Only one channel can be observed.

Example

```
Uint16 C1, C4;
/* Returns Channel Status of the channel  1 of the partition B
*/
C1=MCBSP_channelStatus(hMcbsp,RCERB,MCBSP_CHAN1);
/* Returns Channel Status of the channel 4 of the partition A
*/
C4=MCBSP_channelStatus(hMcbsp,RCERA,MCBSP_CHAN4);
```

MCBSP_close *Closes McBSP port*

Function

```
void MCBSP_close(
    MCBSP_Handle hMcbsp
);
```

Arguments hMcbsp Handle to McBSP port obtained by MCBSP_open()

Return Value None

Description Closes a McBSP port previously opened via MCBSP_open(). The registers for the McBSP port are set to their power-on defaults and 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 obtained by MCBSP_open()
Config	Pointer to an initialized configuration structure

Return Value None

Description Sets up the McBSP port identified by hMcbsp handle using the configuration structure. The values of the structure are written to the MCBSP port registers.

> **Note:**
>
> If you want to configure all MCBSP registers without starting the MCBSP port, use MCBSP_config() without setting the SPCR2 (XRST, RRST, GRST, and FRST) fields. Then,
>
> ❑ Step 1: Start Transmit and receive
>
> ❑ Step 2: Write the first data valid to the DXR registers
>
> ❑ Step 3: Start Sample rate generator and framesync
>
> This guarantees that the correct value is transmitted/received.

Example

```
MCBSP_Config MyConfig = {
   0x8001,  /* spcr1 */
   0x0001,  /* spcr2 */
   0x0000,  /* rcr1  */
   0x0000,  /* rcr2  */
   0x0000,  /* xcr1  */
   0x0000,  /* xcr2  */
   0x0001,  /* srgr1 */
   0x2000,  /* srgr2 */
   0x0000,  /* mcr1  */
   0x0000,  /* mcr2  */
   0x0000,  /* pcr   */
   0x0000,  /* rcera */
   0x0000,  /* rcerb */
   0x0000,  /* xcera */
   0x0000   /* xcerb */
};
…
MCBSP_config(hMcbsp,&MyConfig);
```

For complete examples, refer to section 11.4.

MCBSP_configArgs *Sets up McBSP port using register values passed in*

Function

```
void MCBSP_configArgs(
    MCBSP_Handle hMcbsp,
    Uint16 spcr1,
    Uint16 spcr2,
    Uint16 rcr1,
    Uint16 rcr2,
    Uint16 xcr1,
    Uint16 xcr2,
    Uint16 srgr1,
    Uint16 srgr2,
    Uint16 mcr1,
    Uint16 mcr2,
    Uint16 pcr,
```

For devices that support 128 channels:

```
    Uint16 rcera,
    Uint16 rcerb,
    Uint16 rcerc,
    Uint16 rcerd,
    Uint16 rcere,
    Uint16 rcerf,
    Uint16 rcerg,
    Uint16 rcerh,
    Uint16 xcera,
    Uint16 xcerb,
    Uint16 xcerc,
    Uint16 xcerd,
    Uint16 xcere,
    Uint16 xcerf,
    Uint16 xcerg,
    Uint16 xcerh,
```

For devices that do not support 128 channels:

```
    Uint16 rcera,
    Uint16 rcerb,
    Uint16 xcera,
    Uint16 xcerb
    );
```

Arguments	hMcbsp	Handle to McBSP port obtained by MCBSP_open()
	spcr1	Serial port control register 1 value
	spcr2	Serial port control register 2 value
	rcr1	Receive control register 1 value
	rcr2	Receive control register 2 value
	xcr1	Transmit control register 1 value
	xcr2	Transmit control register 2 value
	srgr1	Sample rate generator register 1 value
	srgr2	Sample rate generator register 2 value
	mcr1	Multi-channel control register 1 value
	mcr2	Multi-channel control register 2 value
	pcr	Pin control register value
	rcerx	Receive channel enable register partition x value
	xcerx	Transmit channel enable register partition x value
	Where x= A, B, C, D, E, F, G, H	
Return Value	None	

Description

Sets up the McBSP port using the register values that are passed. The register values are written to the port registers.

Note:

If you want to configure all MCBSP registers without starting the MCBSP port, use MCBSP_config() without setting the SPCR2 (XRST, RRST, GRST, and FRST) fields. Then,

- ❏ Step 1: Start Transmit and receive
- ❏ Step 2: Write the first data valid to the DXR registers
- ❏ Step 3: Start Sample rate generator and framesync

This guarantees that the correct value is transmitted/received.

You may use literal values for the arguments or for readability, you may use the MCBSP_RMK macros to create the register values based on field values.

Example

```
MCBSP_configArgs(hMcbsp,
   0x8001, /* spcr1 */
   0x0001, /* spcr2 */
   0x0000, /* rcr1 */
   0x0000, /* rcr2  */
   0x0000, /* xcr1  */
   0x0000, /* xcr2  */
   0x0001, /* srgr1 */
   0x2000, /* srgr2 */
   0x0000, /* mcr1 */
   0x0000, /* mcr2 */
   0x0000  /* pcr  */
   0x0000, /* rcera*/
   0x0000, /* rcerb*/
   0x0000, /* xcera*/
   0x0000 /* xcerb*/

);
```

MCBSP_getConfig *Get MCBSP channel configuration*

Function

```
void MCBSP_getConfig  (
        MCBSP_Handle  hMcbsp,
        MCBSP_Config   *Config
         );
```

Arguments

hMcbsp	Handle to McBSP port; (see MCBSP_open())
Config	Pointer to an initialized configuration structure (see MCBSP_Config)

Return Value None

Description Get the current configuration for the McBSP port used by handle. This is accomplished by reading the actual McBSP port registers and fields and storing them back in the Config structure.

Example

```
MCBSP_Config  ConfigRead;
  ...
  myHandle = MCBSP_open (MCBSP_PORT0, 0);
 MCBSP_getConfig (myHandle, &ConfigRead);
```

MCBSP_getPort *Get McBSP port number used in given handle*

Function Uint16 MCBSP_getPort (MCBSP_Handle hMcbsp);

Arguments

hMcbsp	Handle to McBSP port given by MCBSP_open()

Return Value Port number

Description Get Port number used by specific handle

Example

```
Uint16  PortNum;
  ...
  PortNum = MCBSP_getPort (hMcbsp);
```

MCBSP_getRcvEventId *Retrieves receive event ID for given port*

Function

```
Uint16 MCBSP_getRcvEventId(
        MCBSP_Handle hMcbsp
        );
```

Arguments

hMcbsp Handle to McBSP port obtained by MCBSP_open()

Return Value

Receiver event ID

Description

Retrieves the IRQ receive event ID for the given port. Use this ID to manage the event using the IRQ module.

Example

```
Uint16 RecvEventId;
...
RecvEventId = MCBSP_getRcvEventId(hMcbsp);
IRQ_enable(RecvEventId);
```

MCBSP_getXmt EventID *Retrieves transmit event ID for given port*

Function

```
Uint16 MCBSP_getXmtEventId(
        MCBSP_Handle hMcbsp
      );
```

Arguments

hMcbsp Handle to McBSP port obtained by MCBSP_open()

Return Value

Transmitter event ID

Description

Simple replace receive for transmit. Use this ID to manage the event using the IRQ module.

Example

```
Uint16 XmtEventId;
...
XmtEventId = MCBSP_getXmtEventId(hMcbsp);
IRQ_enable(XmtEventId);
```

MCBSP_open — *Opens McBSP port*

Function

```
MCBSP_Handle MCBSP_open(
        int devNum,
       Uint32 flags
        );
```

Arguments

devNum — McBSP device (port) number:
- ❑ MCBSP_PORT0
- ❑ MCBSP_PORT1
- ❑ MCBSP_PORT2 (if available in the device)
- ❑ MCBSP_PORT_ANY

flags — Open flags, may be logical OR of any of the following:
- ❑ MCBSP_OPEN_RESET

Return Value

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 (-1) 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_PORT0,MCBSP_OPEN_RESET);
```

MCBSP_read16 — *Performs16-bit data read*

Function

```
Uint16 MCBSP_read16(
    MCBSP_Handle hMcbsp
    );
```

Arguments

hMcbsp — Handle to McBSP port obtained by MCBSP_open()

Return Value

Data read for MCBSP receive port.

Description

Directly reads a 16-bit value from the McBSP data receive register DRR1. **Depending on the receive word data length you have selected in the RCR1/RCR2 registers, the actual data could be 8, 12, or 16 bits long. This function does not verify if new valid data as been received. Use MCBSP_rrdy() prior to calling MCBSP_read32() for this purpose.**

Example

```
Uint16 Data;
...
Data = MCBSP_read16(hMcbsp);
```

MCBSP_read32 *Performs 32-bit data read*

Function

```
Uint32 MCBSP_read32(
    MCBSP_Handle hMcbsp
    );
```

Arguments hMcbsp Handle to McBSP port obtained by MCBSP_open()

Return Value Data (MSW-LSW ordering)

Description A 32-bit read. First, the 16-bit MSW (Most significant word) is read from register DRR2. Then, the 16-bit LSW (least significant word) is read from register DRR1. **Depending on the receive word data length you have selected in the RCR1/RCR2 register, the actual data could be 20, 24, or 32 bits. This function does not verify that new valid data has been received. Use MCBSP_rrdy() prior to calling MCBSP_read32() for this purpose.**

Example

```
Uint32 Data;
...
Data = MCBSP_read32(hMcbsp);
```

MCBSP_reset *Resets given serial port*

Function

```
void MCBSP_reset(
    MCBSP_Handle hMcbsp
    );
```

Arguments hMcbsp Handle to McBSP port obtained by MCBSP_open()

Return Value None

Description Resets the given serial port. If you use INV (-1) for hMcbsp, all serial ports are reset. Actions Taken:

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

Example

```
MCBSP_reset(hMcbsp);
MCBSP_reset(INV);
```

MCBSP_rfull

Reads RFULL bit of serial port control register 1

Function

```
CSLBool MCBSP_rfull(
    MCBSP_Handle hMcbsp
    );
```

Arguments

hMcbsp	Handle to McBSP port obtained by MCBSP_open()

Return Value

RFULL	Returns RFULL status bit of SPCR1 register, 0 (receive buffer empty) or 1(receive buffer full)

Description

Reads the RFULL bit of the serial port control register 1. (Both RBR and RSR are full. A receive overrun error could have occured.)

Example

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

MCBSP_rrdy

Reads RRDY status bit of SPCR1 register

Function

```
CSLBool MCBSP_rrdy(
    MCBSP_Handle hMcbsp
    );
```

Arguments

hMcbsp	Handle to McBSP port obtained by MCBSP_open()

Return Value

RRDY	Returns RRDY status bit of SPCR1, 0 (no new data to be received) or 1 (new data has been received)

Description

Reads the RRDY status bit of the SPCR1 register. A 1 indicates the receiver is ready with data to be read.

Example

```
if (MCBSP_rrdy(hMcbsp)) {
val = MCBSP_read16(hMcbsp);
}
```

MCBSP_start

Starts transmit and/or receive operation for a McBSP port

Function

```
void MCBSP_start(
    MCBSP_Handle hMcbsp,
    Uint16 startMask,
    Uint16 SampleRateGenDelay
    );
```

Arguments

hMcbsp	Handle to McBSP port obtained by MCBSP_open()
startMask	Start mask. It could be any of the following values (or their logical OR): MCBSP_XMIT_START: start transmit (XRST field) MCBSP_RCV_START: start receive (RRST field) MCBSP_SRGR_START: start sample rate generator (GRST field) MCBSP_SRGR_FRAMESYNC: start framesync generation (FRST field)
SampleRateGenDelay	Sample rate generates delay. MCBSP logic requires two sample_rate generator clock_periods after enabling the sample rate generator for its logic to stabilize. Use this parameter to provide the appropriate delay before starting the MCBSP. A conservative value should be equal to:

$$SampleRateGenDelay = \frac{2 \; x \; Sample\_Rate\_Generator\_Clock\_period}{4 \; x \; C54x\_Instruction\_Cycle}$$

A default value of:

MCBSP_SAMPLE_RATE_DELAY_DEFAULT (0xFFFF value)

can be used (maximum value).

Return Value

None

Description

Starts a transmit and/or a receive operation for a MCBSP port.

Note:

If you want to configure all MCBSP registers without starting the MCBSP port, use MCBSP_config() without setting the SPCR2 (XRST, RRST, GRST, and FRST) fields. Then,

- ❏ Step 1: Start Transmit and receive
- ❏ Step 2: Write the first data valid to the DXR registers
- ❏ Step 3: Start Sample rate generator and framesync

This guarantees that the correct value is transmitted/received (see example 2).

Example 1

```
MCBSP_start(hMcbsp,
      MCBSP_SRGR_START|MCBSP_RCV_START,
      0x200
      );
```

Example 2

```
/* Step 1 */
MCBSP_config(hMcbsp, &MyConfig);
MCBSP_start(hMcbsp, MCBSP_RCV_START|MCBSP_XMIT_START, 0x200);

/* Step 2 */
MCBSP_write16(hMcbsp, 0x1234);

/* Step 3 */
MCBSP_start(hMcbsp, MCBSP_SRGR_START|MCBSP_SRGR_FRAMESYNC,
                                                0x200);
```

MCBSP_write16 *Writes a 16-bit data value*

Function

```
void MCBSP_write16(
    MCBSP_Handle hMcbsp,
    Uint16 Val
    );
```

Arguments

hMcbsp	Handle to McBSP port obtained by MCBSP_open()
Val	16-bit data value to be written to MCBSP transmit register.

Return Value None

Description Directly writes a 16-bit value to the serial port data transmit register; DXR1. Depending on the receive word data length you have selected in the RCR1/RCR2 registers, the actual data could be 8, 12, or 16 bits long. **This function does not check if the transmitter is ready.** Use MCBSP_xrdy() prior to calling MCBSP_write16() for this purpose.

Example

```
MCBSP_write16(hMcbsp,0x1234);
```

MCBSP_write32 *Writes a 32-bit data value*

Function

```
Void MCBSP_write32(
    MCBSP_Handle hMcbsp,
    Uint32 Val
    );
```

Arguments

hMcbsp	Handle to McBSP port obtained by MCBSP_open()
Val	32-bit data value

Return Value None

Description Depending on the transmit word data length you have selected in the XCR1|XCR2 registers, the actual data could be 20, 24, or 32 bits long. **This function does not verify that all valid data has been transmitted. Use MCBSP_xrdy(), prior to calling MCBSP_write32(), for this purpose.**

Example

```
MCBSP_write32(hMcbsp,0x12345678);
```

MCBSP_xempty *Reads XEMPTY bit from SPCR2 register*

Function

```
CSLBool MCBSP_xempty(
    MCBSP_Handle hMcbsp
    );
```

Arguments

hMcbsp	Handle to McBSP port obtained by MCBSP_open()

Return Value

XEMPTY	Returns XEMPTY bit of SPCR2 register, 0(transmit buffer empty) or 1(transmit buffer full)

Description

Reads the XEMPTY bit from the SPCR2 register. A 0 indicates the transmit shift (XSR) is empty.

Example

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

MCBSP_xrdy *Reads XRDY status bit of SPCR2 register*

Function

```
CSLBool MCBSP_xrdy(
    MCBSP_Handle hMcbsp
    );
```

Arguments

hMcbsp	Handle to McBSP port obtained by MCBSP_open()

Return Value

XRDY	Returns XRDY status bit of SPCR2, 0 (not ready to transmit) and 1 (ready to transmit).

Description

Reads the XRDY status bit of the SPCR2 register. A “1” indicates that the transmitter is ready to transmit a new word. A ”0” indicates that the transmitter is not ready to transmit a new word.

Example

```
if (MCBSP_xrdy(hMcbsp)) {
  ...
  MCBSP_write16 (hMcbsp, 0x1234);
  ...
}
```

11.4 Macros

As covered in section 1.5, the CSL offers a collection of macros to get individual access to the peripheral registers and fields.

The following are the list of macros available for the MCBSP. To use these macros, include "`csl_mcbsp.h`".

Because the MCBSP has several channels, macros identify the channel by either the channel number or the handle used.

Table 11–3 lists the macros available for a MCBSP channel using the channel number as part of the register name.

Table 11–4 lists the macros available for a MCBSP channel using its corresponding handle.

Table 11–3. MCBSP CSL Macros (using port number)

(a) Macros to read/write MCBSP register values

Macro
MCBSP_RGET()
MCBSP_RSET()

(b) Macros to read/write MCBSP register field values (Applicable only to registers with more than one field)

Macro
MCBSP_FGET()
MCBSP_FSET()

(c) Macros to read/write MCBSP register field values (Applicable only to registers with more than one field)

Macro
MCBSP_REG_RMK()
MCBSP_FMK()

(d) Macros to read a register address

Macro
MCBSP_ADDR()

Table 11–4. MCBSP CSL Macros (using handle)

(a) Macros to read/write MCBSP register values

Macro	See page...
MCBSP_RGETH()	11-34
MCBSP_RSETH()	11-36

(b) Macros to read/write register field values (Applicable only to registers with more than one field)

Macro	See page...
MCBSP_FGETH()	11-37
MCBSP_FSETH()	11-38

(c) Macros to read a register address

Macro	See page...
MCBSP_ADDRH()	11-40

MCBSP_RGET *Get the value of a MCBSP register*

Macro Uint16 MCBSP_RGET (REG#)

Arguments

REG#	Register name with channel number (#) where # = 0,1, (2: depending on the device) DRR1# DRR2# DXR1# DXR2# SPCR1# SPCR2# RCR1# RCR2# XCR1# XCR2# SRGR1# SRGR2# MCR1# MCR2# PCR# RCERA# RCERB# XCERA# XCERB# For devices supporting 128-channels, add: RCERC# XCERC# RCERD# XCERD# RCERE# XCERE# RCERF# XCERF# RCERG# XCERG# RCERH# XCERH#

Return Value value of register

Description Returns the MCBSP register value

Example

```
Uint16     myVar;
...
myVar = MCBSP_RGET(RCR10); /*get register RCR1 of channel 0 */
```

MCBSP_RSET *Set the value of a MCBSP register*

Macro Void MCBSP_REG_SET (MCBSP_Handle hMcbsp, Uint16 RegVal)

Arguments

REG# Register name with channel number (#) where
= 0,1, (2: depending on the device)
DRR1#
DRR2#
DXR1#
DXR2#
SPCR1#
SPCR2#
RCR1#
RCR2#
XCR1#
XCR2#
SRGR1#
SRGR2#
MCR1#
MCR2#
PCR#
RCERA#
RCERB#
XCERA#
XCERB#

For devices supporting 128-channels, add:
RCERC#
XCERC#
RCERD#
XCERD#
RCERE#
XCERE#
RCERF#
XCERF#
RCERG#

XCERG#
RCERH#
XCERH#

regval — Register value needed to write to register REG

Return Value None

Description Set the MCBSP register REG value to regval

Example

```
MCBSP_RSET(RCR10, 0x4); /* RCR1C for channel 0 = 0x4 */
```

MCBSP_REG_RMK *Creates a register value based on individual field values*

Macro Uint16 MCBSP_REG_RMK (fieldval_n,...,fieldval_0)

Arguments

REG — **Only writable registers containing more than one field are supported by this macro. Please note that the channel number is not used as part of the register name.**

SPCR1
SPCR2
RCR1
RCR2
XCR1
XCR2
SRGR1
SRGR2
MCR1
MCR2
PCR
RCERA
RCERB
XCERA
XCERB

For devices supporting 128-channels, add:
RCERC
XCERC
RCERD
XCERD
RCERE
XCERE
RCERF
XCERF

RCERG
XCERG
RCERH
XCERH

fieldval_n field values to be assigned to the register fields rules to follow:
- ❏ Only writable fields are allowed
- ❏ Start from Most-significat field first
- ❏ Value should be a right-justified constant. If fieldval_n value exceeds the number of bits allowed for that field, then fieldval_n is truncated accordingly.

Return Value value of register that corresponds to the concatenation of values passed for the fields. (writable fields only)

Description Returns the MCBSP register value given to specific field values. You can use constants or the CSL symbolic constants covered in section 1.6.

Example

```
/* frame length, word length */
myVal = MCBSP_RCR1_RMK Uint16 myVal (4,3);

or you can use the PER_REG_FIELD_SYMVAL symbolic constants
provided in CSL (See section 1.6)
```

MCBSP_REG_RMK macros are typically used to initialize a MCBSP configuration structure used for the MCBSP_config() function. For more examples see section 11.6.

MCBSP_FMK *Creates a register value based on individual field values*

Macro Uint16 MCBSP_FMK (REG, FIELD, fieldval)

Arguments REG **Only writable registers containing more than one field are supported by this macro. Please note that the channel number is not used as part of the register name.**
SPCR1
SPCR2
RCR1
RCR2
XCR1
XCR2
SRGR1

SRGR2
MCR1
MCR2
PCR
RCERA
RCERB
XCERA
XCERB

For devices supporting 128-channels, add:
RCERC
XCERC
RCERD
XCERD
RCERE
XCERE
RCERF
XCERF
RCERG
XCERG
RCERH
XCERH

FIELD — Symbolic name for field of register REG. Possible values are the field names as listed in the C54x Register Reference Guide. **Only writable fields are allowed**.

fieldval — field values to be assigned to the register fields rules to follow:

- ❏ Only writable fields are allowed
- ❏ Start from Most-significat field first
- ❏ Value should be a right-justified constant. If fieldval_n value exceeds the number of bits allowed for that field, then fieldval_n is truncated accordingly.

Return Value

Shifted version of fieldval. fieldval is shifted to the bit numbering appropriate for FIELD.

Description

Returns the shifted version of fieldval. fieldval is shifted to the bit numbering appropriate for FIELD within register REG. This macro allows the user to initialize few fields in REG as an alternative to the MCBSP_REG_RMK() macro that requires ALL the fields in the register to be initialized. The returned value could be ORed with the result of other _FMK macros, as shown in the example below.

Example

```
Uint16 myregval;
Myregval = MCBSP_FMK (RCR1, RFRLEN1, 1) | MCBSP_FMK (RCR1,
RWDLEN1,2);
```

MCBSP_FGET *Gets the value of a register field*

Macro

Uint16 MCBSP_FGET (REG#, FIELD)

Arguments

REG# Register name with channel number (#) where # = 0,1, (2: depending on the device)
DRR1#
DRR2#
DXR1#
DXR2#
SPCR1#
SPCR2#
RCR1#
RCR2#
XCR1#
XCR2#
SRGR1#
SRGR2#
MCR1#
MCR2#
PCR#
RCERA#
RCERB#
XCERA#
XCERB#

For devices supporting 128-channels, add:
RCERC#
XCERC#
RCERD#
XCERD#
RCERE#
XCERE#
RCERF#
XCERF#
RCERG#
XCERG#
RCERH#
XCERH#

FIELD symbolic name for field of register REG. Possible values are the field names listed in the C54x Register Reference Guide (Appendix x) **Only readable fields are allowed**.

Return Value Value of register field

Description Gets the MCBSP register FIELD value

Example

```
Uint16 myVar;
...
myVar = MCBSP_FGET(RCR20,RPHASE);
```

MCBSP_FSET *Sets the value of a register field*

Macro Void MCBSP_FSET (REG#, FIELD, fieldval)

Arguments

REG# Register name with channel number (#) where
= 0,1, (2: depending on the device)
DRR1#
DRR2#
DXR1#
DXR2#
SPCR1#
SPCR2#
RCR1#
RCR2#
XCR1#
XCR2#
SRGR1#
SRGR2#
MCR1#
MCR2#
PCR#
RCERA#
RCERB#
XCERA#
XCERB#

For devices supporting 128-channels, add:
RCERC#
XCERC#
RCERD#
XCERD#

RCERE#
XCERE#
RCERF#
XCERF#
RCERG#
XCERG#
RCERH#
XCERH#

FIELD — Symbolic name for field of register REG. Possible values: Field names as listed in the C54x Register Reference Guide. **Only writable fields are allowed**.

fieldval — field values to be assigned to the register fields rules to follow:

- ❏ Only writable fields are allowed
- ❏ Start from Most-significat field first
- ❏ Value should be a right-justified constant. If fieldval_n value exceeds the number of bits allowed for that field, then fieldval_n is truncated accordingly.

Return Value None

Description Set the MCBSP register value to regval

Example For Registers:

```
MCBSP_FSET(RCR20,RPHASE,2);
```

MCBSP_ADDR *Get the address of a given register*

Macro Uint16 MCBSP_ADDR (REG#)

Arguments REG# — Register name with channel number (#) where # = 0,1, (2: depending on the device)

DRR1#
DRR2#
DXR1#
DXR2#
SPCR1#
SPCR2#
RCR1#
RCR2#
XCR1#

XCR2#
SRGR1#
SRGR2#
MCR1#
MCR2#
PCR#
RCERA#
RCERB#
XCERA#
XCERB#

For devices supporting 128-channels, add:
RCERC#
XCERC#
RCERD#
XCERD#
RCERE#
XCERE#
RCERF#
XCERF#
RCERG#
XCERG#
RCERH#
XCERH#

Return Value Address of register REG

Description Get the address of a given MCBSP register.

Example For Registers:

```
myVar = MCBSP_ADDR(RCR10); /*get register RCR1 of channel 0 */
```

MCBSP_RGETH *Get the value of a MCBSP register used in a handle*

Macro Uint16 MCBSP_RGETH (MCBSP_Handle hMcbsp, REG)

Arguments

hMcbsp	Handle to MCBSP channel that identifies the MCBSP channel used.
REG	Similar to register in MCBSP_RGET(), but without channel number (#). DRR1

DRR2
DXR1
DXR2
SPCR1
SPCR2
RCR1
RCR2
XCR1
XCR2
SRGR1
SRGR2
MCR1
MCR2
PCR
RCERA
RCERB
XCERA
XCERB

For devices supporting 128-channels, add:
RCERC
XCERC
RCERD
XCERD
RCERE
XCERE
RCERF
XCERF
RCERG
XCERG
RCERH
XCERH

Return Value value of register

Description Returns the MCBSP register value for register REG for the channel associated with handle.

Example

```
MCBSP_Handle myHandle;
Uint16     myVar;
...
myHandle = MCBSP_open (MCBSP_PORT0, MCBSP_OPEN_RESET);
...
myVar = MCBSP_RGETH(myHandle, RCR1);
```

MCBSP_RSETH *Set the value of a MCBSP register*

Macro Void MCBSP_RSETH (MCBSP_Handle hMcbsp, REG, Uint16 RegVal)

Arguments

hMcbsp — Handle to McBSP port that identifies specific McBSP port being used.

REG# — Similar to register in MCBSP_RGET(), but without channel number (#).

DRR1
DRR2
DXR1
DXR2
SPCR1
SPCR2
RCR1
RCR2
XCR1
XCR2
SRGR1
SRGR2
MCR1
MCR2
PCR
RCERA
RCERB
XCERA
XCERB

For devices supporting 128-channels, add:

RCERC
XCERC
RCERD
XCERD
RCERE
XCERE
RCERF
XCERF
RCERG
XCERG
RCERH
XCERH

regval — value to write to register REG for the channel associated with handle.

Return Value None

Description Set the MCBSP register REG for the channel associated with handle to the value regval.

Example

```
MCBSP_Handle myHandle;
...
myHandle = MCBSP_open (MCBSP_PORT0, MCBSP_OPEN_RESET);
...
MCBSP_RSETH(myHandle, RCR1, 0x4);
```

MCBSP_FGETH *Get the value of a register field*

Macro Uint16 MCBSP_FGETH (MCBSP_Handle Hmcbsp, REG, FIELD)

Arguments

hMcbsp — Handle to McBSP port that identifies specific McBSP port being used.

REG — Similar to register in MCBSP_RGET(), but without channel number (#).
DRR1
DRR2
DXR1
DXR2
SPCR1
SPCR2
RCR1
RCR2
XCR1
XCR2
SRGR1
SRGR2
MCR1
MCR2
PCR
RCERA
RCERB
XCERA

XCERB

For devices supporting 128-channels, add:
RCERC
XCERC
RCERD
XCERD
RCERE
XCERE
RCERF
XCERF
RCERG
XCERG
RCERH
XCERH

FIELD — symbolic name for field of register REG Possible values: Field names listed in the C54x Register Reference Guide **Only readable fields are allowed**.

Return Value — Value of register field given by FIELD and of REG used by handle.

Description — Gets the MCBSP register FIELD value

Example

```
MCBSP_Handle myHandle;
Uint16 myVar;
...
myHandle = MCBSP_open (MCBSP_PORT0, MCBSP_OPEN_RESET);
...
myVar = MCBSP_FGETH(myHandle, RCR2, RPHASE);
```

MCBSP_FSETH *Set the value of a register field*

Macro — Void MCBSP_FSETH (MCBSP_Handle hMcbsp, REG, FIELD, fieldval)

Arguments

hMcbsp — Handle to McBSP port that identifies specific McBSP port being used.

REG# — Similar to register in MCBSP_RGET(), but without channel number (#).
DRR1
DRR2
DXR1

DXR2
SPCR1
SPCR2
RCR1
RCR2
XCR1
XCR2
SRGR1
SRGR2
MCR1
MCR2
PCR
RCERA
RCERB
XCERA
XCERB

For devices supporting 128-channels, add:
RCERC
XCERC
RCERD
XCERD
RCERE
XCERE
RCERF
XCERF
RCERG
XCERG
RCERH
XCERH

FIELD — Symbolic name for field of register REG. Possible values are the field names as listed the C54x Register Reference Guide. **Only writable fields are allowed**.

fieldval — field values to be assigned to the register fields
rules to follow:

- ❏ Only writable fields are allowed
- ❏ Value should be a right-justified constant. If fieldval_n value exceeds the number of bits allowed for that field, then fieldval is truncated accordingly.

Return Value None

Description Set the MCBSP register field FIELD of the REG register for the channel associated with handle to the value fieldval.

Example

```
MCBSP_Handle myHandle;
...
myHandle = MCBSP_open (MCBSP_PORT0, MCBSP_OPEN_RESET);
...
MCBSP_FSETH(myHandle, RCR2, RPHASE,1);
```

MCBSP_ADDRH *Get the address of a given register*

Macro Uint16 MCBSP_ADDR (REG#)

Arguments

hMcbsp — Handle to MCBSP channel that identifies the MCBSP channel used. Use only for MCBSP channel registers. Registers are listed as part of the MCBSP_RGETH macro description.

REG — Similar to register in MCBSP_RGET(), but without channel number (#).
DRR1
DRR2
DXR1
DXR2
SPCR1
SPCR2
RCR1
RCR2
XCR1
XCR2
SRGR1
SRGR2
MCR1
MCR2
PCR
RCERA
RCERB
XCERA
XCERB

For devices supporting 128-channels, add:
RCERC

XCERC
RCERD
XCERD
RCERE
XCERE
RCERF
XCERF
RCERG
XCERG
RCERH
XCERH

Return Value Address of register REG

Description Gets the address of the MCBSP register associated with handle hMCBSP

Example 1

```
MCBSP_Handle myHandle;
Uint16 myVar;
...
myVar = MCBSP_ADDRH(myHandle, RCR1);
```

11.5 Configuring the McBSP Module Using CSL GUI

11.5.1 Overview

The McBSP module facilitates configuration/control of the Multichannel Buffered Serial Port (McBSP). The module consists of a configuration manager and a resource manager. The configuration manager allows creation of one or more configuration objects. The configuration objects contain all of the data necessary to set the McBSP Control Registers. The resource manager associates a configuration object with a specified port.

Figure 11–1 illustrates the GPIO sections menu on the CSL graphical user interface (GUI).

Figure 11–1. 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.

11.5.2 McBSP Configuration Manager

The McBSP Configuration Manager allows you to create device configurations through the Properties page and to generate the configuration objects.

11.5.2.1 Creating/Inserting a Configuration Object

There is no predefined configuration object available.

To configure a McBSP port through the peripheral registers, you must insert a new configuration object.

To insert a new configuration object, right-click on the McBSP Configuration Manager and select insert mcbspCfg from the drop-down menu. The configuration objects can be renamed. Their use depends upon the on-chip device resources.

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 McBSP Resource Manager. The goal is to provide more flexibility and to reduce the time required to modify register values.

11.5.2.2 Deleting/Renaming an Object

To delete or to rename an object, right-click on the configuration object you want to delete or rename. Select Delete to delete a configuration object. Select Rename to rename the object.

If a configuration object is used by one of the predefined handle objects of the McBSP 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 Figure 2–1, *The CSL Tree,* on page 2-3).

11.5.2.3 Configuring the Object Properties

The Properties pages allow you to set the Peripheral registers related to the McBSP Port (see Figure 11–2). 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.

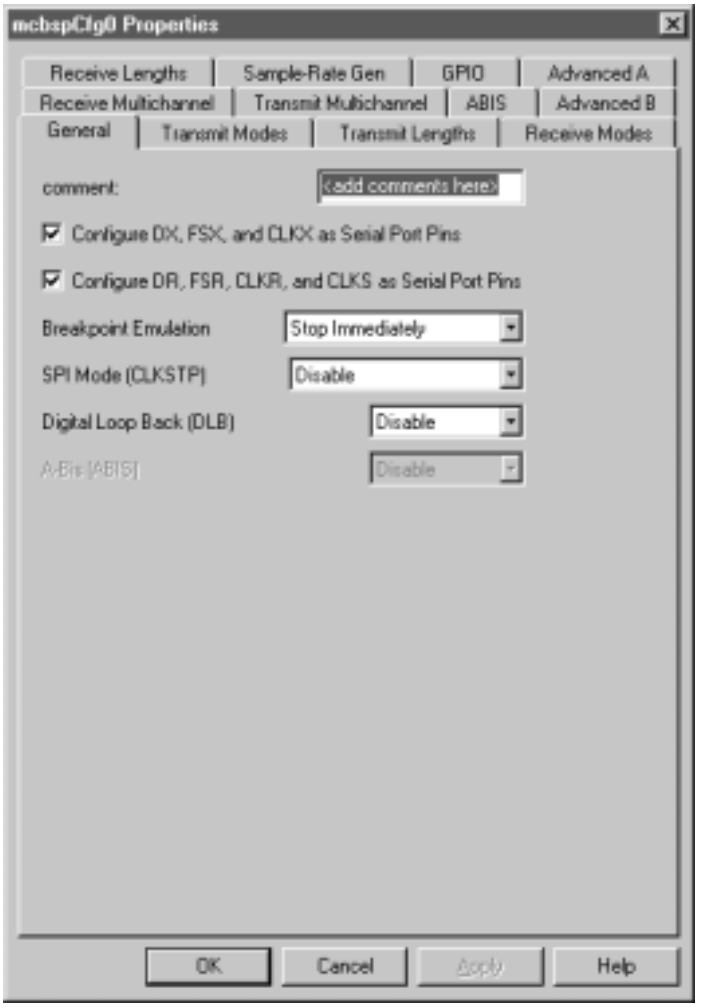
The Properties pages allow you to set the Peripheral registers related to the McBSP. You can set the configuration options through the following pages:

- ❏ General: Allows you to configure the Digital Loopback, ABIS Mode, Breakpoint Emulation.
- ❏ Transmit Modes: Allows you to configure the Transmit Interrupt mode, Frame Sync, Clock control.
- ❏ Transmit Lengths: Allows you to configure the Transmit Phase, elements-per-word, elements per frame.
- ❏ Receive Modes: Allows you to configure the Receive Interrupt mode, Frame Sync, Clock control.
- ❏ Receive Lengths: Allows you to configure the Receive Phase, elements-per-word, elements per frame.
- ❏ Sample-Rate Generator: Allows you to configure the Sample-Rate Generator (Frame Setup).

- Receive Multichannel: Allows you to configure the Receive Element and Block partitioning.
- Transmit Multichannel: Allows you to configure the Transmit Element and Block partitioning.
- GPIO: General Purpose I/O pin configuration.
- ABIS: Allows you to configure the McBSP ABIS mode.
- Some fields are activated according to the setup of the Transmitter, Receiver, and Sample-rate generator options.
- Advanced A and B: Summary of the previous pages. These pages contain 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 11–2 depicts the Properties Page.

Figure 11–2. McBSP Properties Page



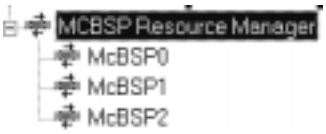
Each Tab page is composed of several options that are set to a default value (at device reset).

11.5.3 McBSP Resource Manager

The McBSP Resource Manager allows you to generate the MCBSP_open() and the MCBSP_config() CSL functions.

Figure 11–3 illustrates the McBSP Resource Manager menu on the CSL graphical user interface (GUI).

Figure 11–3. McBSP Resource Manager Menu



11.5.3.1 Predefined Objects

Three handle objects are predefined and each of them is associated with a supported on-chip McBSP port.

- **McBSP0** – Default handle name: hMcbsp0
- **McBSP1** – Default handle name: hMcbsp1
- **McBSP2** – 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 11.5.2, *McBSP Configuration Manager*.

11.5.3.2 Properties Page

You can generate the MCBSP_open() and MCBSP_config() CSL functions through the Properties page.

To access the Properties page, right-click on a predefined McBSP channel and select Properties from the drop-down menu (see Figure 11–4).

The first time the Properties page appears, only the Open Handle to McBSP check-box can be selected. Select this to open the McBSP channel, allowing pre-initialization.

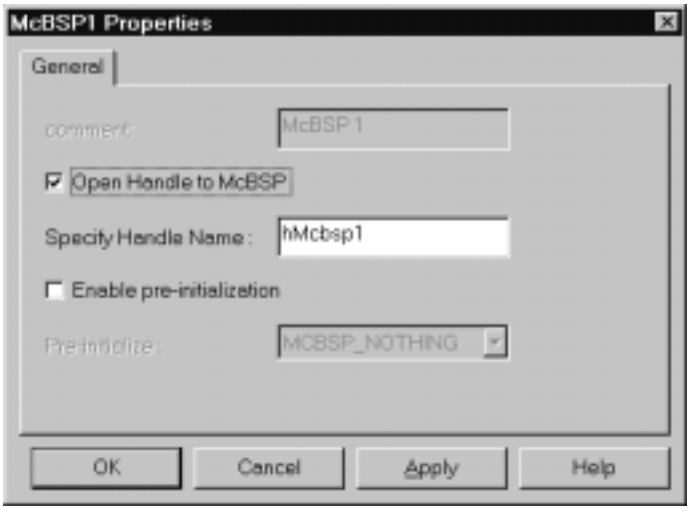
MCBSP_NOTHING is used to indicate that there is no configuration object selected for this serial port.

To pre-initialize a McBSP port, check the Enable Pre-Initialization box. You can then select one of the available configuration objects (see section 11.5.2, *McBSP Configuration Manager*) for this channel through the pre-initialize drop-down list.

If MCBSP_NOTHING is selected, no configuration object is generated for the related McBSP handle. (see section 11.5.4, *C Code Generation for McBSP Module*).

In the example shown in Figure 11–4, the Open Handle to McBSP option is checked and the handle object hMcbsp1 is now accessible (The handle object can be renamed by typing the new name in the box provided). The MCBSP_open() function is now generated with hMcbsp0 containing the returned handle address.

Figure 11–4. McBSP Properties Page With Handle Object Accessible



11.5.4 C Code Generation for McBSP Module

Two C files are generated from the configuration tool:

- Header file
- Source file.

11.5.4.1 Header File

The header file includes all the csl header files of the modules and contains the McBSP handle and configuration objects defined from the configuration tool (see Example 11–1).

Example 11–1. McBSP Header File

```
extern MCBSP_Config mcbsCfg0;
extern MCBSP_Handle hMcbsp1;
```

11.5.4.2 Source File

The source file includes the declaration of the handle object and the configuration structures (see Example 11–2).

Example 11–2. McBSP Source File (Declaration Section)

```
/*  Config Structures */
MCBSP_Config mcbspCfg0 = {
   0x0000,   /* Serial Port Control Register 1              */
   0x0000,   /* Serial Port Control Register 2              */
   0x0000,   /* Receive Control Register 1                  */
   0x0000,   /* Receive Control Register 2                  */
   0x0000,   /* Transmit Control Register 1                 */
   0x0000,   /* Transmit Control Register 2                 */
   0x0000,   /* Sample Rate Generator Register 1            */
   0x0000,   /* Sample Rate Generator Register 2            */
   0x0000,   /* Multi–channel Control Register 1            */
   0x0000,   /* Multi–channel Control Register 2            */
   0x0000,   /* Pin Control Register                        */
   0x0000,   /* Receive Channel Enable Register Partition A  */
   0x0000,   /* Receive Channel Enable Register Partition B  */
   0x0000,   /* Transmit Channel Enable Register Partition A */
   0x0000    /* Transmit Channel Enable Register Partition B */
};

/*  Handles  */
MCBSP_Handle hMcbsp1;
```

The source file contains the Handle and Configuration Pre-Initialization using the CSL McBSP functions, MCBSP_open() and MCBSP_config() (see Example 11–3).

These two functions are encapsulated in a unique function, CSL_cfgInit(), which is called from your main C file. MCBSP_open() and MCBSP_config() are generated only if Open Handle to McBSP and Enable pre-initialization (with a selected configuration other than MCBSP_NOTHING) are, respectively, checked under the McBSP Resource Manager Properties page.

You must use MCBSP_start() in your code to start the McBSP. The MCBSP_Config structure generated by the CSL GUI does not start the McBSP port transmit or receive operations (unless you force this behavior through the McBSP GUI Advance Tab settings). In the case of a write, this is typically done after you write your first data to the DXR register(s) to guarantee a valid data transfer (see note in MCBSP_start() description on page 11-20).

Example 11–3. McBSP Source File (Body Section)

```
void CSL_cfgInit()
{


    hMcbsp1 = MCBSP_open(MCBSP_PORT1, MCBSP_OPEN_RESET);
    MCBSP_config(hMcbsp1, &mcbspCfg0);
}
```

11.6 Examples

Examples for the McBSP module are found in the CCS examples\<target>\csl directory.

Example 11–4 illustrates the McBSP port initialization using MCBSP_config(). The example also explains how to set the McBSP into digital loopback mode and perform 32-bit reads/writes from/to the serial port.

Example 11–4. McBSP Port Initialization using MCBSP_config

```
#include <csl.h>
#include <csl_mcbsp.h>

/* Step 0: This is your MCBSP register configuration */

static MCBSP_Config ConfigLoopBack32= {
   ....
};


void main(void) {

  MCBSP_Handle mhMcbsp;
  Uint32 xmt[n], rcv[n];

  ....

/* Step 1: Initialize CSL */

   CSL_init();

/* Step 2: Open and configure the MCBSP port */

   mhMcbsp = MCBSP_open(MCBSP_PORT0, MCBSP_OPEN_RESET);

   MCBSP_config(mhMcbsp, &ConfigLoopBack32);

/* Step 3: Write the first data value and start */
/* the sample rate genterator in the MCBSP     */

   MCBSP_write32(mhMcbsp, xmt[0];
   MCBSP_start(mhMcbsp,MCBSP_SRGR_START|MCBSP_SRGR_FRAMESYNC,0x300u);

......

   while (!MCBSP_rrdy((mhMcbsp));
   rcv[0] = MCBSP_read32(mhMcbsp);
```

Example 11–4. McBSP Port Initialization using MCBSP_config (Continued)

```
/* Begin the data transfer loop of the remaining (N-1) values. */
for (i=1; i<N-1;i++)
   {

/* Wait for XRDY signal before writing data to DXR */
   while (!MCBSP_xrdy(mhMcbsp));

/* Write 32 bit data value to DXR */
   MCBSP_write32(mhMcbsp,xmt[i]);

/* Wait for RRDY signal to read data from DRR */
   while (!MCBSP_rrdy(mhMcbsp));

/* Read 32 bit value from DRR */
   rcv[i] = MCBSP_read32(mhMcbsp);
}
   MCBSP_close(mhMcbsp);

} /* main */
```

Chapter 12

PLL Module

This chapter describes the structure, functions, and macros of the PLL module.

Topic **Page**

12.1 Overview

The CSL PLL module offers functions and macros to control the Phase Locked Loop fo the C54xx clock.

The PLL module is not handle-based.

Table 12–1 lists the configuration structure to use with the PLL functions.

Table 12–2 lists the functions available as part of the PLL module.

Section 12.4 includes a description of available PLL macros.

Table 12–1. PLL Configuration Structure

Structure	Purpose	See page...
PLL_Config	PLL structure that contains the register required to setup the PLL.	12-3

Table 12–2. PLL Functions

Function	Purpose	See page ...
PLL_config()	Configure the PLL with the values provided in a configuration structure.	12-4
PLL_configArgs()	Configures the PLL with values provided as function arguments.	12-4
PLL_setFreq()	Initializes the PLL to produce the desired CPU output frequency (clkout)	12-5

12.2 Configuration Structure

This section describes the structure in the PLL module.

PLL_Config *PLL configuration structure used to set up PLL interface*

Structure PLL_Config

Members

Uint16 pllmode	available values PLL_MODE_DIV = 1 (divide mode) PLL_MODE_MUL = 2 (pll multiplier mode) This value combines the effect of the PLLNDIV and PLLDIV fields.
Uint16 pllcount	internal lockup counter (number of PLL clock input cycles that the PLL logic should wait before “locking” in the new frequency.)
Uint16 pllmul	PLL multiplier register field value.

For CHIP_5410 and CHIP_5416 add:

Uint16 divfct	divide down factor

Description PLL configuration structure used to set up the PLL Interface. You create and initialize this structure and then pass its address to the PLL_config() function.

Example

```
/* clock_out freq = clock_in freq * final multiplier */
PLL_Config  myconfig = {
   PLL_MODE_DIV,
   20,
   1       /* final multiplier = 0.5 */
}
```

12.3 Functions

This section describes the functions in the PLL module.

PLL_config — *Writes a value to set up PLL using configuration structure*

Function void PLL_config (PLL_Config *config);

Arguments Config Pointer to an initialized configuration structure

Return Value none

Description Writes a value to up the PLL using the configuration structure. The values of the structure are written to the port registers. See also PLL_configArgs() and PLL_Config.

Example

```
PLL_Config MyConfig {
           PLL_MODE_DIV,
           20,
           1
           };
PLL_config (&MyConfig);
```

PLL_configArgs — *Writes to PLL using register values passed to function*

Function void PLL_configArgs (Uint16 pllmode, Uint16 pllmul, Uint16 pllcount);
For 5410 and 5416 add: Uint16 divfct);

Arguments

Uint16 pllmode — available values:
PLL_MODE_DIV = 1 (divide mode)
PLL_MODE_MUL = 2 (pll multiplier mode)
This value combines the effect of the PLLNDIV and PLLDIV fields.

Uint16 pllcount — Internal lockup counter (number of PLL clock input cycles that the PLL logic should wait before “locking” in the new frequency.)

Uint16 pllmul — PLL multiplier register field value.

Uint16 divfct — Divide down factor

Return Value none

Description Writes to the PLL using the register values passed to the function. The register values are written to the PLL registers. You may use literal values for the arguments.

Clock out frequency is determined as follows:

Example

```
PLL_configArgs (PLL_MODE_DIV, 1, 20);
```

PLL_setFreq

Initializes the PLL to produce the desired CPU output frequency

Function void PLL_setFreq (Uint16 mul, Uint16 div);

Arguments

mul integer multiplier
div integer divisor

Where mul should not be further divisible by div. The table below shows valid ranges for mul and div:

	Range
mul	[1,15]
div	{1,2,4}

This function does not verify that arguments passed are within valid ranges.

Return Value None

Description Initializes the PLL to produce the desired CPU output frequency (clkout)

Example

```
PLL_setFreq (1, 2);    // set clkout = 1/2 clkin
```

12.4 Macros

As covered in section 1.5, CSL offers a collection of macros to get individual access to the peripheral registers (CLKMD) and fields.

The following is a list of macros available for the PLL module. To use them, include “csl_pll.h”.

Table 12–3. PLL CSL Macros

(a) Macros to read/write PLL register values

Macro	Syntax
PLL_RGET()	Uint16 PLL_RGET(*REG*)
PLL_RSET()	Void PLL_RSET(*REG*, Uint16 *regval*)

(b) Macros to read/write PLL register field values (Applicable only to registers with more than one field)

Macro	Syntax
PLL_FGET()	Uint16 PLL_FGET(*REG*, *FIELD*)
PLL_FSET()	Void PLL_FSET(*REG*, *FIELD*, Uint16 *fieldval*)

(c) Macros to create value to PLL registers and fields (Applies only to registers with more than one field)

Macro	Syntax
PLL_*REG*_RMK()	Uint16 PLL_*REG*_RMK(*fieldval_n*,…*fieldval_0*) Note: *Start with field values with most significant field positions: field_n: MSB field field_0: LSB field *only writable fields allowed
PLL_FMK()	Uint16 PLL_FMK(*REG*, *FIELD*, *fieldval*)

(d) Macros to read a register address

Macro	Syntax
PLL_ADDR()	Uint16 PLL_ADDR(*REG*)

Where:

REG: CLKMD

FIELD: register field name as specified in Appendix A.

- For *REG*_FSET and *REG*_FMK, *FIELD* should be a writable field.
- *REG*_FGET, the field must be a readable field.

regVal: value to write in register (REG)

fieldVal: value to write in field (FIELD)

For examples on how to use macros, refer macro sections 6.4 (DMA) and 11.4 (McBSP).

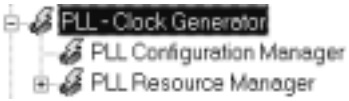
12.5 Configuring the PLL Module Using CSL GUI

12.5.1 Overview

The PLL module facilitates programming of the Phase Locked Loop controlling C54xx clock. The PLL module consists of a configuration manager and a resource manager. The configuration manager allows creation of one or more configuration objects. A configuration object consists of the necessary register settings to control the PLL. The resource manager associates a selected configuration with the PLL.

Figure 12–1 illustrates the PLL sections menu on the CSL graphical user interface (GUI).

Figure 12–1. PLL Sections Menu



The PLL includes the following two sections:

- **PLL Configuration Manager** allows you to create configuration objects by setting the Peripheral registers related to the PLL.
- **PLL Resource Manager** allows you to associate a configuration object to the PLL.

12.5.2 PLL Configuration Manager

The PLL Configuration Manager allows you to create PLL configurations through the Properties page and to generate the configuration objects.

12.5.2.1 Creating/Inserting a configuration

There is no predefined configuration object.

To configure a PLL setting through the Peripheral Registers, you must insert a new configuration object.

To insert a new configuration object, right-click on the PLL Configuration Manager and select Insert pllCfg. The configuration objects can be renamed.

Note:

The number of configuration objects is unlimited. Several configurations can be created. You user can select one for the PLL and can change the configuration later just by selecting another configuration under the PLL Resource Manager. This feature allows you more flexibility and reduces the time required to modify register values.

12.5.2.2 Deleting/Renaming and Object

To delete or rename an object, right-click on the configuration object you want to delete or rename. Select Delete to delete a configuration object. Select Rename to rename the object.

If a configuration object is used by the predefined object of the PLL 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 Figure 2–1, *The CSL Tree,* on page 2-3).

12.5.2.3 Configuring the Object Properties

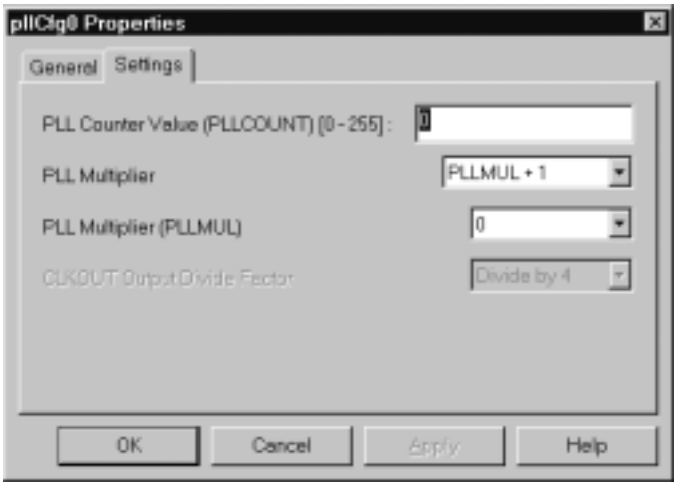
You can configure object properties through the Properties dialog box (see Figure 12–2). To access the Properties dialog box, right-click on a configuration object and select Properties. By default, the General page of the Properties dialog box is displayed.

The Properties pages allow you to set the Peripheral registers related to the PLL. You can set the configuration options through the following tab page:

- Settings: Allows you to configure the Counter Value, Multiplier, Divide Factor

Figure 12–2 depicts the Properties Page dialog box.

Figure 12–2. PLL 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 PLL registers; the associated field name is shown in parenthesis. For further details concerning the fields and registers for 5416, refer to the *Expansion Bus* chapter of the *TMS320C54x DSP CPU and Peripherals References Set* (SPRU131F).

12.5.3 PLL Resource Manager

The PLL Resource Manager allows you to generate the PLL_config() CSL function.

Because only one PLL is supported, only one resource is available and used as the default.

Figure 12–3 illustrates the PLL Resource Manager menu on the CSL graphical user interface (GUI).

Figure 12–3. PLL Resource Manager Menu

12.5.3.1 Properties Page

You can generate the PLL_config() CSL function through the Properties page.

To access the Properties page, right-click on a predefined PLL channel and select Properties from the drop-down menu (see Figure 12–4).

The first time the properties page appears, only the Enable Configuration of PLL check box can be selected. Select this to enable the PLL configuration.

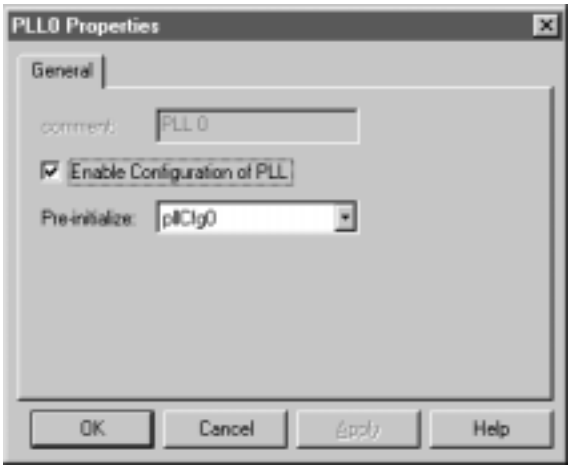
PLL_NOTHING is used to indicate that there is no configuration object selected for this peripheral.

To pre-initialize the PLL channel, check the Enable Configuration of PLL box. One of the available configuration objects (see section 6.5.1, *PLL Configuration Manager*) can then be selected for this channel through the Pre-Initialize drop-down list.

If PLL_NOTHING remains selected, The PLL_config() function will not be generated for the PLL.

In Figure 12–4, the Enable Configuration of PLL option is checked and the PLL_config function will be generated (See section 12.5.4, *C Code Generation for PLL Module*).

Figure 12–4. PLL Properties Page



12.5.4 C Code Generation for PLL Module

Two C files are generated from the configuration tool:

- Header file
- Source file.

12.5.4.1 Header File

The header file includes all the csl header files of the modules and contains the PLL configuration objects defined from the configuration tool (see Example 12–1).

Example 12–1. PLL Header File

```
extern PLL_Config pllCfg0;
```

12.5.4.2 Source File

The source file includes the declaration of the configuration structures (values of the peripheral registers) (see Example 12–2).

Example 12–2. PLL Source File (Declaration Section)

```
/*  Config Structures */
PLL_Config pllCfg0 = {
    0x2,  /* PLL Multiplier/Divider Mode */
    0x0,  /* PLL Counter Value (PLLCOUNT)  */
    0x0,  /* PLL Multiplier Value (PLLMUL)  */
};
```

The source file contains the Pre-Initialization PLL API function, PLL_config(). This function is encapsulated into a unique function, CSL_cfgInit(), which is called from your main C file (see Example 12–3).

PLL_config() is generated only if Enable Configuration of PLL is checked under the PLL Resource Manager Properties page (with a selected configuration other than PLL_NOTHING) (see Figure 12–4).

Example 12–3. PLL Source File (Body Section)

```
void CSL_cfgInit()
{

   PLL_config(&pllCfg0);
}
```

Chapter 13

PWR Module

The CSL PWR module offers functions to control the power consumption of different sections in the C54x device.

Topic **Page**

13.1 Overview

The CSL PWR module offers functions to control the power consumption of different sections in the C54x device. The PWR module is not handle-based.

Currently, there are no macros available for the power-down module.

Table 13–1 lists the functions for use with the PWR modules that order specific parts of the C54x to power down.

Table 13–1. PWR Functions

Function	Purpose	See page ...
PWR_powerDown	Forces the DSP to enter a power-down(IDLE) state	13-3

13.2 Functions

This section lists the functions in the PWR module.

PWR_powerDown *Forces the DSP to enter a power-down state*

Function void PWR_powerDown (Uint16 pwrMode, Uint16 wakeMode);

Arguments

mode pwrMode:

- ❏ PWR_CPU_DOWN: CPU goes idle, but peripherals keep running. This corresponds to the IDLE #1 instruction.
- ❏ PWR_CPU_PER_DOWN: Both CPU and peripherals power-down. This corresponds to the IDLE #2 instruction.
- ❏ PWR_CPU_PER_PLL_DOWN: CPU, peripherals, and PLL power-down. This corresponds to the IDLE #3 instruction.

wakeMode (Valid for all pwrdModes above)

- ❏ PWR_WAKEUP_MI: Wakes up with an unmasked interrupt and jumps to execute the ISRs executed.
- ❏ PWR_WAKEUP_NMI: Wakes up with an unmasked interrupt and executes the next instruction (interrupt is not taken).

Return Value None

Description Power-down the device in different power-down and wake-up modes. In the C54x, power-down is achieved by executing an IDLE K instruction.

Example

```
PWR_powerDown (PWR_CPU_DOWN, PWR_WAKEUP_MI);
```

Chapter 14

TIMER Module

This chapter describes the structure and functions for the TIMER Module.

Topic **Page**

14.1 Overview

Table 14–1 lists the configuration structure for the TIMER module.

Table 14–2 lists, in the order in which they are typically called, the functions available for use with the TIMER modules.

Section 14.4 includes descriptions for available TIMER macros.

Table 14–1. TIMER Configuration Structure

Structure	Purpose	See page...
TIMER_Config	TIMER configuration structure used to setup a timer device	14-3

Table 14–2. TIMER Functions

Function	Purpose	See page ...
TIMER_close()	Closes a previously opened TIMER device	14-4
TIMER_config()	Sets up the TIMER register using the configuration structure	14-4
TIMER_configArgs()	Sets up the TIMER using the register values passed in	14-5
TIMER_getConfig()	Gets the TIMER configuration	14-5
TIMER_getEventId()	Obtains IRQ event ID for the timer device	14-6
TIMER_open()	Opens a TIMER device	14-6
TIMER_reload()	Reloads the TIMER	14-7
TIMER_reset()	Resets the TIMER device	14-7
TIMER_start()	Starts the TIMER device running	14-7
TIMER_stop()	Stops the TIMER device running	14-8

14.2 Configuration Structure

This section lists the structure in the TIMER module.

TIMER_Config *TIMER configuration structure used to setup Timer device*

Structure TIMER_Config

Members

Uint16 tcr	Control register value
Uint16 prd	Period register value

For C5440, C5441, and C5471 devices only:

Uint16 tscr	Timer scaler register

Description The TIMER configuration structure is used to setup a timer device. You create and initialize this structure then pass its address to the TIMER_config() function. You can use literal values or the TIMER_RMK macros to create the structure member values.

Example

```
TIMER_Config MyConfig = {
  0x0000, /* tcr */
  0x1000 /* prd */
  };
…
TIMER_config(hTimer,&MyConfig);
```

14.3 Functions

This section lists the functions in the TIMER module.

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

Closes a previously opened timer device (see TIMER_open()).

The following tasks are performed:

- ❏ The timer IRQ event is disabled and cleared
- ❏ The timer registers are set to their default values

Example

```
TIMER_close(hTimer);
```

TIMER_config — *Sets up TIMER register using configuration structure*

Function

```
void TIMER_config(
    TIMER_Handle hTimer,
    TIMER_Config *Config
);
```

Arguments

hTimer — Device handle, (see TIMER_open()).

config — Pointer to an initialized configuration structure

Return Value

None

Description

Sets up the TIMER register using the configuration structure. The values of the structure are written to the registers TCR, PRD, TIM, (see also TIMER_configArgs() and TIMER_Config.)

Example

```
TIMER_Config MyConfig = {
...
};
…
TIMER_config(hTimer,&MyConfig);
```

TIMER_configArgs *Sets up TIMER using register values passed in*

Function

```
void TIMER_configArgs(
    TIMER_Handle hTimer,
    Uint16 tcr,
For 5440, 5441, and 5471:
    Uint16 tscr,
For all devices:
    Uint16 prd);
```

Arguments

hTimer	Device handle (see TIMER_open()).
tcr	Control register value
tscr	Secondary control register
prd	Period register value

Return Value

None

Description

Sets up the timer using the register values passed in. The register values are written to the timer registers. The timer control register (tcr) is written last (see also TIMER_config()).

You may use literal values for the arguments or for readability, you may use the TIMER_RMK macros to create the register values based on field values.

Example

```
TIMER_configArgs (hTimer,
   0x0010, /* tcr */
   0x1000 /* prd */
);
```

TIMER_getConfig *Gets the TIMER configuration structure for the specified device*

Function

```
void TIMER_getConfig(
    TIMER_Handle hTimer,
    TIMER_Config *Config
);
```

Arguments

hTimer	Device handle, see TIMER_open()
Config	Pointer to a TIMER configuration structure

Return Value

None

Description

Gets the Timer configuration structure for the specified device.

Example

```
TIMER_Config MyConfig;
TIMER_getConfig(hTimer, &MyConfig);
```

TIMER_getEventId *Obtains IRQ event ID for TIMER device*

Function

```
Uint16 TIMER_getEventId(
    TIMER_Handle hTimer
);
```

Arguments

hTimer Device handle (see TIMER_open()).

Return Value

Event ID IRQ Event ID for the timer device

Description

Obtains the IRQ event ID for the timer device (see IRQ Module, Chapter 9).

Example

```
Uint16 TimerEventId;
TimerEventId = TIMER_getEventId(hTimer);
IRQ_enable(TimerEventId);
```

TIMER_open *Opens TIMER device*

Function

```
TIMER_Handle TIMER_open(
    int DevNum,
    Uint16 Flags
);
```

Arguments

DevNum Device Number:
TIMER_DEV_ANY
TIMER_DEV0
TIMER_DEV1

Flags Open flags, logical OR of any of the following:
TIMER_OPEN_RESET

Return Value

Device Handle Device handle

Description

Before a TIMER device can be used, it must first be opened by this function. Once opened, it cannot be opened again until closed (see TIMER_close()).The return value is a unique device handle that is used in subsequent TIMER API calls. If the open fails, INV (–1) 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_reload — *Reloads TIMER*

Function

```
void TIMER_reload(
    TIMER_Handle hTimer
);
```

Arguments hTimer Device handle (see TIMER_open()).

Return Value None

Description Reloads the timer, TIM loaded with PRD and PSC loaded with TDDR value.

Example

```
TIMER_reload(hTimer);
```

TIMER_reset — *Resets TIMER device*

Function

```
void TIMER_reset(
    TIMER_Handle hTimer
);
```

Arguments hTimer Device handle (see TIMER_open()).

Return Value None

Description Resets the timer device. Disables and clears the interrupt event and sets the timer registers to default values. If INV (–1) is specified, all timer devices are reset.

Example

```
TIMER_reset(hTimer);
TIMER_reset(INV);
```

TIMER_start — *Starts TIMER device running*

Function

```
void TIMER_start(
    TIMER_Handle hTimer
);
```

Arguments hTimer Device handle (see TIMER_open()).

Return Value None

Description Starts the timer device running. TSS field =0.

Example

```
TIMER_start(hTimer);
```

TIMER_stop

Stops TIMER device running

Function

```
void TIMER_stop(
    TIMER_Handle hTimer
);
```

Arguments

hTimer Device handle (see TIMER_open()).

Return Value

None

Description

Stops the timer device running. TSS field =1.

Example

```
TIMER_stop(hTimer);
```

14.4 Macros

CSL offers a collection of macros to access CPU control registers and fields. For additional details, see section 1.5.

Because the TIMER peripheral typically has two independent timers in some, but not all C54x devices, the macros identify the correct timer through either the device number or the handle.

- Table 14–3 lists the TIMER macros available that use the device number as part of the register name.
- Table 14–4 lists the TIMER macros available that use a handle.

Both Table 14–3 and Table 14–4 use the following conventions:

To use the TIMER macros, include csl_timer.h and follow these restrictions:

- Only writable fields are allowed
- Values should be a right-justified constants.
- If *fieldval_n* value exceeds the number of bits allowed for that field, *fieldval_n* is truncated accordingly

For examples that are similar to the TIMER macros, see section 6.4 in the DMA chapter or section 11.4 in the McBSP chapter.

Table 14–3. TIMER CSL Macros Using Timer Port Number

(a) Macros to read/write TIMER register values

Macro	Syntax
TIMER_RGET()	Uint16 TIMER_RGET(*REG#*)
TIMER_RSET()	void TIMER_RSET(*REG#*, Uint16 *regval*)

(b) Macros to read/write TIMER register field values (Applicable only to registers with more than one field)

Macro	Syntax
TIMER_FGET()	Uint16 TIMER_FGET(*REG#*, *FIELD*)
TIMER_FSET()	Void TIMER_FSET(*REG#*, *FIELD*, Uint16 *fieldval*)

(c) Macros to create value to write to TIMER registers and fields (Applies only to registers with more than one field)

Macro	Syntax
TIMER_*REG*_RMK()	Uint16 TIMER_*REG*_RMK(*fieldval_n*,...*fieldval_0*) Note: *Start with field values with most significant field positions: field_n: MSB field field_0: LSB field * only writable fields allowed
TIMER_FMK()	Uint16 TIMER_FMK(*REG*, *FIELD*, *fieldva*l)

(d) Macros to read a register address

Macro	Syntax
TIMER_ADDR()	Uint16 TIMER_ADDR(*REG#*)

Notes: 1) *REG* indicates the register, TCR, PRD, TSCR (C5440, C5441, C5471 only), or TIM**.**

2) REG# indicates, if applicable, a register name with the channel number (example: TCR0)

3) *FIELD* indicates the register field name as specified in Appendix A.

- ❏ For *REG*_FSET and *REG*_FMK, *FIELD* must be a writable field.
- ❏ For *REG*_FGET, the field must be a readable field.

4) *regval* indicates the value to write in the register (*REG*)

5) *fieldval* indicates the value to write in the field *(FIELD*)

Table 14–4. TIMER CSL Macros Using Handle

(a) Macros to read/write TIMER register values

Macro	Syntax
TIMER_RGETH()	Uint16 TIMER_RGETH(TIMER_Handle hTimer, *REG*)
TIMER_RSETH()	void TIMER_RSETH(TIMER_Handle hTimer, *REG*, Uint16 *regval*)

(b) Macros to read/write TIMER register field values (Applicable only to registers with more than one field)

Macro	Syntax
TIMER_FGETH()	Uint16 TIMER_FGETH(TIMER_Handle hTimer, *REG*, *FIELD*)
TIMER_FSETH()	Void TIMER_FSETH(TIMER_Handle hTimer, *REG*, *FIELD*, *fieldval*)

(c) Macros to read a register address

Macro	Syntax
TIMER_ADDRH()	Uint16 TIMER_ADDRH(TIMER_Handle hTimer, *REG*)

Notes: 1) *REG* indicates the register, TCR, PRD, TSCR (C5440, C5441, C5471 only), or TIM**.**

2) *FIELD* indicates the register field name as specified in Appendix A.

- For *REG*_FSET and *REG*_FMK, *FIELD* must be a writable field.
- For *REG*_FGET, the field must be a readable field.

3) *regVal* indicates the value to write in the register (*REG*)

4) *fieldVal* indicates the value to write in the field *(FIELD*)

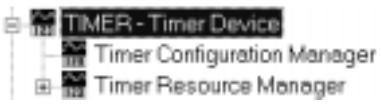
14.5 Configuring the TIMER Module Using CSL GUI

14.5.1 Overview

The Timer module facilitates configuration/control of the on-chip Timer. The timer module consists of a configuration manager and a resource manager. The configuration manager allows the creation of one or more configuration objects. The configuration object consists of the necessary data to set the Timer control registers. The resource manager associates a selected configuration with a timer.

Figure 14–1 illustrates the Timer sections menu on the CSL graphical user interface (GUI).

Figure 14–1. Timer Sections Menu



The TIMER includes the following two sections:

- **TIMER Configuration Manager** allows you to create configuration objects. There are 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. Two handle objects are predefined for some devices and just one for other devices.

14.5.2 TIMER Configuration Manager

The TIMER Configuration Manager allows you to create device configurations through the Properties page and generate the configuration objects.

14.5.2.1 Creating/Inserting a configuration

There are no predefined configuration objects available.

To configure a TIMER device through the peripheral, you must insert a new configuration object.

To insert a new configuration object, right-click on the TIMER Configuration Manager and select Insert timerCfg from the drop-down menu. The configuration objects can be renamed. Their use depends on the on-chip device resources.

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.

14.5.2.2 Deleting/Renaming an Object

To delete or to rename an object, right-click on the configuration object you want to delete or rename. Select Delete to delete a configuration object. Select Rename to rename the object.

If a configuration object is used by one of the predefined handle objects of the TIMER Resource Manager (see section 14.5.3, *Timer 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 Figure 2–1, *The CSL Tree,* on page 2-3).

14.5.2.3 Configuring the Object Properties

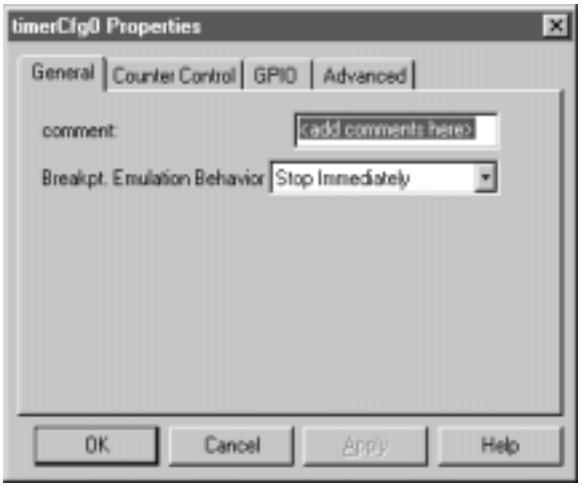
You can configure object properties through the Properties dialog box (see Figure 14–2). To access the Properties dialog box, right-click on a configuration object and select Properties. By default, the General page of the Properties dialog box is displayed.

The Properties pages allow you to set the Peripheral registers related to the TIMER. You can set the configuration options through the following tab pages:

- ❑ General: Allows you to configure the Breakpoint Emulation
- ❑ Counter Control: Allows you to configure the Counter configuration
- ❑ GPIO: General Purpose I/O pin configuration
- ❑ Advanced Page: Allows you to configure the Summary of the previous pages
- ❑ This page contains the full hexadecimal register values and reflects the setting of the previous pages
- ❑ The full register values can be entered directly and the new options will be mirrored on the previous three pages automatically

Figure 14–2 depicts the Properties Page dialog box.

Figure 14–2. TIMER Properties Page



Each Tab page is composed of several options that are set to a default value (at device reset).

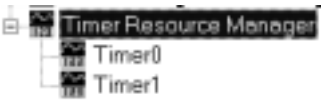
The options represent the fields of the TIMER registers; the associated field name is shown in parenthesis.

14.5.3 TIMER Resource Manager

The TIMER Resource Manager allows you to generate the TIMER_open() and the TIMER_config() CSL functions.

Figure 14–3 illustrates the DMA Resource Manager menu.

Figure 14–3. Timer Resource Manager Menu



14.5.3.1 Predefined Objects

Two handle objects are predefined and each of them is associated with a supported on-chip TIMER device.

- **TIMER0** – Default handle name: hTimer0
- **TIMER1** – Default handle name: hTimer1

Note:

The above objects can neither be deleted nor renamed.

A configuration is enabled if at least one configuration object is defined previously in section 14.5.2.

14.5.3.2 Properties Page

You can generate the TIMER_config and TIMER_open CSL functions through the Properties page.

To access the Properties page, right-click on a predefined TIMER handle object and select Properties from the drop-down menu (see Figure 14–4).

The first time the properties page appears, only the Open Handle to Timer check-box can be selected. Select this to open the TIMER configuration, allowing pre-initialization.

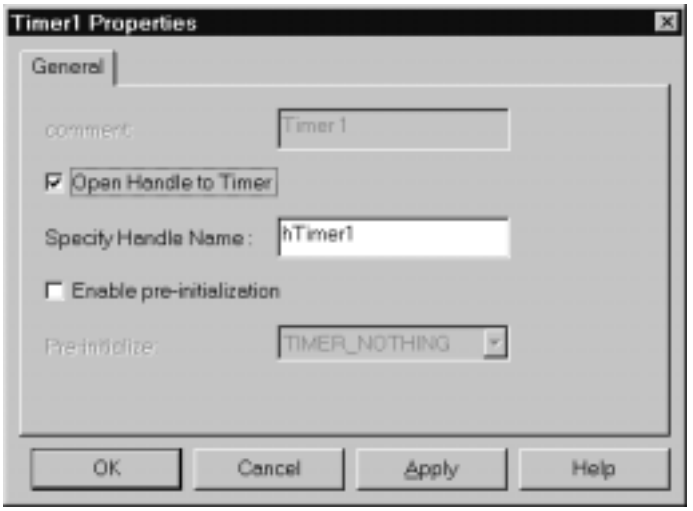
TIMER_NOTHING is used to indicate that there is no configuration object selected for this device.

To pre-initialize the TIMER channel, check the Enable Pre-Initialization box. One of the available configuration objects (see section 14.5.2 , *TIMER Configuration Manager*) can then be selected for this channel through the Pre-initialize drop-down list.

If TIMER_NOTHING is selected, no configuration object will be generated for the related TIMER handle (see section 14.5.4, *C Code Generation for TIMER*.)

In Figure 14–4, the Open Handle to TIMER option is checked and the handle object hTimer1 is now accessible (renaming allowed). The TIMER_open() function will be generated with hTimer1 containing the return handle address.

Figure 14–4. Timer Properties Page With Handle Object Accessible



14.5.4 C Code Generation for TIMER

Two C files are generated from the configuration tool:

- Header file
- Source file.

14.5.4.1 Header File

The header file includes all the csl header files of the modules and contains the TIMER handle and configuration objects defined from the configuration tool (see Example 14–1).

Example 14–1. Timer Header File

```
extern TIMER_Config timerCfg0;
extern TIMER_Handle hTimer1;
```

14.5.4.2 Source File

The source file includes the declaration of the handle object and the configuration structures (see Example 14–2).

Example 14–2. Timer Source File (Declaration Section)

```
/*  Config Structures */
TIMER_Config timerCfg1 = {
    0x0000,        /*  Timer Control Register   */
    0x0000         /*  Timer Period Register   */
};

/*  Handles  */
TIMER_Handle hTimer1;
```

The source file contains the Handle and Configuration Pre-Initialization using CSL TIMER API functions TIMER_open() and TIMER_config() (see Example 14–3). These two functions are encapsulated into a unique function, CSL_cfgInit(), which is called from your main C file.

TIMER_open() and TIMER_config() will be generated only if Open Handle to TIMER and Enable-Pre-Initialization (with timerCfg0) are, respectively, checked on the TIMER Resource Manager Properties page.

Note:

You must use TIMER_start() in your code to start the TIMER. The TIMER_Config structure generated by the CSL GUI does not start the TIMER port.

Example 14–3. Timer Source File (Body Section)

```
void CSL_cfgInit()
{

   hTimer1 = TIMER_open(TIMER_DEV1, TIMER_OPEN_RESET);
   TIMER_config(hTimer1, &timerCfg1);

}
```

Chapter 15

UART Module

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

Topic **Page**

15.1 Overview

The Universal Asynchronous Receiver/Transmitter (UART) controller is the key component of the serial communications subsystem of a computer. Asynchronous transmission allows data to be transmitted without a clock signal to the receiver. Instead, the sender and receiver must agree on timing parameters in advance. Special bits are added to each word that is used to synchronize the sending and receiving units.

The configuration of UART can be performed by using one of the following methods:

1) Register-based configuration

   A register-based configuration can be performed by calling either UART_config(), UART_configArgs(), or any of the SET register field macros.

2) Parameter-based configuration (Recommended)

   A parameter-based configuration can be performed by calling UART_setup(). Compared to the register-based approach, this method provides a higher level of abstraction.

Table 15–1 lists the configuration structures and functions used with the UART module.

Table 15–1. UART APIs

Structure	Type	Purpose	See page ...
UART_Config	S	UART configuration structure used to setup the UART	15-5
UART_config	F	Sets up the UART using the configuration structure	15-8
UART_configArgs	F	Sets up the UART using regsiter values	15-8
UART_eventDisable	F	Disable UART interrupts	15-9
UART_eventEnable	F	Enable UART interrupts	15-9
UART_fgetc	F	Read a character from UART by polling	15-11
UART_fgets	F	This routine reads a string from the uart	15-12
UART_fputc	F	Write a character from UART by polling	15-12
UART_fputs	F	This routine writes a string from the uart	15-12
UART_getConfig	F	Reads the UART configuration	15-12

Note: F = Function; S = Structure

Table 15–1. UART APIs (Continued)

Structure	Type	Purpose	See page ...
UART_read	F	Read a buffer of data from UART by polling	15-13
UART_setCallback	F	Plugs UART interrupt routines into UART dispatcher table	15-13
UART_Setup	S	UART configuration structure used to setup the UART	15-5
UART_setup	F	Sets up the UART using the register values passed into the code	15-14
UART_write	F	Write a buffer of data to UART by polling	15-14

Note: F = Function; S = Structure

15.2 Configuration Structures

UART_Config *Configuration Structure for UART*

Members

Uint16	dll	Divisor Latch Register (low 8 bits)
Uint16	dlm	Divisor Latch Register (high 8 bits)
Uint16	lcr	Line Control Register
Uint16	fcr	FIFO Control Register
Uint16	mcr	Modem Control Register

Description UART configuration structure. This structure is created and initialized, and then passed to the UART_Config() function.

UART_Setup *Structure used to initialize the UART*

Members

Uint16 clkInput — UART input clock frequency. Valid symbolic values are:

- UART_CLK_INPUT_58 // Input clock = 58.9824MHz
- UART_CLK_INPUT_117 // Input clock = 117.9684MHz

Uint16 baud — Baud Rate (Range: 150 – 115200). Valid symbolic values are:

- UART_BAUD_150
- UART_BAUD_300
- UART_BAUD_600
- UART_BAUD_1200
- UART_BAUD_1800
- UART_BAUD_2000
- UART_BAUD_2400
- UART_BAUD_3600
- UART_BAUD_4800
- UART_BAUD_7200

UART_BAUD_9600
UART_BAUD_19200
UART_BAUD_38400
UART_BAUD_57600
UART_BAUD_115200

Uint16 wordLength bits per word (Range: 5,6,7,8).
Valid symbolic values are:
UART_WORD5 5 bits per word
UART_WORD6 6 bits per word
UART_WORD7 7 bits per word
UART_WORD8 8 bits per word

Uint16 stopBits stop bits in a word (1, 1.5, and 2)
Valid symbolic values are:
UART_STOP1 1 stop bit
UART_STOP1_PLUS_HALF
1 and 1/2 stop bits
UART_STOP2 2 stop bits

Uint16 parity parity setups
Valid symbolic values are:
UART_DISABLE_PARITY
UART_ODD_PARITY odd parity
UART_EVEN_PARITY even parity
UART_MARK_PARITY mark parity
(the parity bit is always '1')
UART_SPACE_PARITY space parity
(the parity bit is always '0')

Uint16 fifoControl FIFO Control
Valid symbolic values are:
UART_FIFO_DISABLE
UART_FIFO_DMA0_TRIG01
UART_FIFO_DMA0_TRIG04
UART_FIFO_DMA0_TRIG08
UART_FIFO_DMA0_TRIG14

DMA mode0: always be 0

		RCVR FIFO trigger level: There are four trigger levels for the RCVR FIFO interrupt. TRIG01 – 1 byte TRIG04 – 4 bytes TRIG08 – 8 byte TRIG14 – 14 bytes
	Uint16 loopbackEnable	loopback Enable Valid Symbolic values are: UART_NO_LOOPBACK UART_LOOPBACK

Description Structure used to init the UART. After created and initialized, it is passed to the UART_init() function.

15.3 Functions

15.3.1 CSL Primary Functions

UART_config — *Initializes the UART using the configuration structure*

Function void UART_config (UART_Config *Config);

Arguments Configure pointer to an initialized configuration structure (containing values for all registers that are visible to the user)

Description Writes a value to initialize the UART using the configuration structure.

Example

```
UART_Config Config = {
   0x00,  /* DLL */
   0x06,  /* DLM – baud rate 150 */
   0x18,  /* LCR – even parity, 1 stop bit, 5
             bits word length */
   0x00,  /* Disable FIFO */
   0x00   /* No Loop Back */
};
UART_config(&Config);
```

UART_configArgs — *Setups up the UART using register values*

Function void UART_configArgs(Uint16 dll, Uint16 dlm, Uint16 lcr, Uint16 Fcr, Uint16 mcr);

Arguments

dll	value to setup DLL register
dlm	value to setup DLM register
lcr	value to setup LCR register
fcr	value to setup FCR register
mcr	value to setup MCR register

Description Sets up the UART using the register values passed.

Example

```
Uint16 tDll = 0x00    /* DLL */
Uint16 tDlm = 0x06    /* DLM */
Uint16 tLcrl = 0x18   /* LCR */
Uint16 tFcr = 0x00    /* FCR */
Uint16 tMcr = 0x00    /* MCR */

UART_configArgs(tDLL, tDlm, tLcr, tFcr, tMcr);
```

UART_eventDisable *Disables UART interrupts*

Function void UART_eventDisable(Uint16 ierMask);

Arguments ierMask can be one or a combination of the following:

```
UART_RINT    0x01     // Enable rx data available
                         interrupt
UART_TINT    0x02     // Enable tx hold register
                         empty interrupt
UART_LSINT   0x04     // Enable rx line status
                         interrupt
UART_MSINT   0x08     // Enable modem status
                         interrupt
UART_ALLINT  0x0f     // Enable all interrupts
```

Description It disables the interrupt specified by the ierMask.

Example

```
UART_eventDisable(UART_TINT);
```

UART_eventEnable *Enables a UART interrupt*

Function void UART_eventEnable (Uint16 isrMask);

Arguments isrMask can be one or a combination of the following:

```
UART_RINT    0x01  // Enable rx data available interrupt
UART_TINT    0x02  // Enable tx hold register
                      empty interrupt
UART_LSINT   0x04  // Enable rx line status interrupt
UART_MSINT   0x08  // Enable modem status interrupt
UART_ALLINT  0x0f  // Enable all interrupts
```

UART_eventEnable

Description It enables the UART interrupt specified by the isrMask.

Example

```
UART_eventEnable(UART_RINT|UART_TINT);
```

UART_fgetc

Reads UART characters

Function

CSLBool UART_fgetc(int *c, Uint32 timeout);

Arguments

c	Character read from UART
timeout	Time out for data ready. If it is setup as 0, means there will be no time out count. The function will block forever until DR bit is set.

Description

Read a character from UART by polling.

Example

```
Int retChar;
CSLBool returnFlag

returnFlag = UART_fgetc(&retChar,0);
```

UART_fgets

Reads UART strings

Function

CSLBool UART_fgets(char* pBuf, int bufSize, Uint32 timeout);

Arguments

pBuf	Pointer to a buffer
bufSize	Length of the buffer
timeout	Time out for data ready. If it is setup as 0, means there will be no time out count. The function will block forever until DR bit is set.

Description

This routine reads a string from the uart. The string will be read upto a newline or until the buffer is filled. The string is always NULL terminated and does not have any newline character removed.

Example

```
char readBuf[10];
CSLBool returnFlag

returnFlag = UART_fgets(&readBuf[0], 10, 0);
```

UART_fputc

Writes characters to the UART

Function

CSLBool UART_fputc(const int c, Uint32 timeout);

Arguments

c	The character, as an int, to be sent to the uart.
timeout	Time out for data ready. If it is setup as 0, means there will be no time out count. The function will block forever if THRE bit is not set.

Description

This routine writes a character out through UART.

Example

Example const int putchar = 'A';
CSLBool returnFlag;

```
ReturnFlag = UART_fputc(putchar, 0);
```

UART_fputs

Writes strings to the UART

Function

CSLBool UART_fputs(const char* pBuf, Uint32 timeout);

Arguments

pBuf	Pointer to a buffer
timeout	Time out for data ready. If it is setup as 0, means there will be no time out count. The function will block forever if THRE bit is not set.

Description

This routine writes a string to the uart. The NULL terminator is not written and a newline is not added to the output.

Example

```
UART_fputs("\n\rthis is a test!\n\r");
```

UART_getConfig

Reads the UART Configuration Structure

Function

void UART_getConfig (UART_Config *Config);

Arguments

Config Pointer to an initialized configuration structure (including all registers that are visible to the user)

Description

Reads the UART configuration structure.

Example

```
UART_Config Config;

UART_getConfig(&Config);
```

UART_read

Reads received data

Function

CSLBool UART_read(char *pBuf, Uint16 length, Uint32 timeout);

Arguments

pbuf	Pointer to a buffer
length	Length of data to be received
timeout	Time out for data ready. If it is setup as 0, means there will be no time out count. The function will block forever until DR bit is set.

Description

Receive and put the received data to the buffer pointed by pbuf.

Example

```
Uint16 length = 10;
char pbuf[length];
CSLBool returnFlag;

ReturnFlag = UART_read(&pbuf[0],length, 0);
```

UART_setCallback

Associates a function to the UART dispatch table

Function

void UART_setCallback(UART_IsrAddr *isrAddr);

Arguments

isrAddr is a structure containing pointers to the 5 functions that will be executed when the corresponding events is enabled.

Description

It associates each function specified in the isrAddr structure to the UART dispatch table.

Example

```
UART_IsrAddr MyIsrAddr= {
   NULL,           // Receiver line status
   UartRxIsr,      // received data available
   UartTxIsr,      // transmiter holding register empty
   NULL            // character time-out indication
};
UART_setCallback(&MyIsrAddr);
```

UART_setup

Sets the UART based on the UART_Setup configuration structure

Function void UART_setup (UART_Setup *Params);

Arguments Params Pointer to an initialized configuration structure that contains values for UART setup.

Description Sets UART based on UART_Setup structure.

Example

```
UART_Setup Params = {
   UART_CLK_INPUT_58,    /* input clock freq        */
   UART_BAUD_115200,     /* baud rate               */
   UART_WORD8,           /* word length              */
   UART_STOP1,           /* stop bits                */
   UART_DISABLE_PARITY,  /* parity                   */
   UART_FIFO_DISABLE,    /* FIFO enable/disable      */
   UART_FIFO_DMA_MODE0,  /* DMA mode                 */
   UART_FIFO_TRIG01,     /* FIFO trigger level       */
   UART_NO_LOOPBACK,     /* Loop Back enable/disable */
};
UART_setup(&Params);
```

UART_write

Transmits buffers of data by polling

Function CSLBool UART_write(char *pBuf, Uint16 length, Uint32 timeout);

Arguments

pbuf	Pointer to a data buffer
Length	Length of the data buffer
timeout	Time out for data ready. If it is setup as 0, means there will be no time out count. The function will block forever if THRE bit is not set.

Description Transmit a buffer of data by polling.

Example

Uint16 length = 4;
char pbuf[4] = {0x74, 0x65, 0x73, 0x74};
CSLBool returnFlag;

```
ReturnFlag = UART_write(&pbuf[0],length,0);
```

15.4 Macros

15.4.1 General Macros

Table 15–2. UART CSL Macros

Macro	Syntax
(a) Macros to read/write UART register values	
UART_RGET()	Uint16 UART_RGET(*REG*)
UART_RSET()	void UART_RSET(*REG*, Uint16 *regval*)
(b) Macros to read/write UART register field values (Applicable only to registers with more than one field)	
UART_FGET()	Uint16 UART_FGET(*REG*, *FIELD*)
UART_FSET()	void UART_FSET(*REG*, *FIELD*, Uint16 *fieldval*)
(c) Macros to create value to write to UART registers and fields (Applicable only to registers with more than one field)	
UART_*REG*_RMK()	Uint16 UART_*REG*_RMK(*fieldval_n*,...*fieldval_0*) Note: *Start with field values with most significant field positions: field_n: MSB field field_0: LSB field * only writable fields allowed
UART_FMK()	Uint16 UART_FMK(*REG*, *FIELD*, *fieldva*l)

Notes: 1) *REG* indicates the registers: URIER, URIIR, URBRB, URTHR, URFCR, URLCR, URMCR, URLSR, URMSR, URDLL or URDLM.

2) *FIELD* indicates the register field name.

3) – or *REG*_FSET and *REG*__FMK, *FIELD* must be a writable field.

4) – For *REG*_FGET, the field must be a readable field.

5) *regval* indicates the value to write in the register (*REG*)

6) *fieldval* indicates the value to write in the field *(FIELD*)

Table 15–2. UART CSL Macros (Continued)

Macro	Syntax
(d) Macros to read a register address	
UART_ADDR()	Uint16 UART_ADDR(*REG*)

Notes: 1) *REG* indicates the registers: URIER, URIIR, URBRB, URTHR, URFCR, URLCR, URMCR, URLSR, URMSR, URDLL or URDLM.

2) *FIELD* indicates the register field name.

3) – or *REG*_FSET and *REG*__FMK, *FIELD* must be a writable field.

4) – For *REG*_FGET, the field must be a readable field.

5) *regval* indicates the value to write in the register (*REG*)

6) *fieldval* indicates the value to write in the field *(FIELD*)

15.4.2 UART Control Signal Macros

All the UART control signals are mapped through HPIGPIO pins. They are configurable through GPIOCR and GPIOSR registers. Since C54x DSP are commonly used as DCE (Data Communication Equipment), these signals are configured as following:

HD0 – DTR – Input

HD1 – RTS – Input

HD2 – CTS – Output

HD3 – DSR – Output

HD4 – DCD – Output

HD5 – RI – Output

UART_ctsOff

Sets a CTS signal to OFF

Macro UART_ctsOff

Arguments None

Description Set CTS signal off.

Example

```
UART_ctsOff;
```

UART_ctsOn

Sets a CTS signal to ON

Macro UART_ctsOn

Arguments None

Description Set CTS signal on.

Example

```
UART_ctsOn;
```

UART_flowCtrlInit

Initializes the HPIGPIO registers for flow control

Macro UART_flowCtrlInit

Arguments None

Description Initialize HPIGPIO registers for flow control.

Example

```
UART_flowCtrlInit;
```

UART_isRts

Verifies that RTS is ON

Macro UART_isRts

Arguments None

Description Check if RTS is on. Return RTS value.

Example CSLBool rtsSignal;

```
rtsSignal = UART_isRts;
```

UART_dtcOff

Sets a DTC signal to OFF

Macro UART_dtcOff

Arguments None

Description Set DTC signal off.

Example

```
UART_dtcOff;
```

UART_dtcOn

Sets a DTC signal to ON

Macro UART_dtcOn

Arguments None

Description Set DTC signal on.

Example

```
UART_dtcOn;
```

UART_riOff

Sets an RI signal to OFF

Macro UART_riOff

Arguments None

Description Set RI signal off.

Example

```
UART_riOff;
```

UART_riOn

Sets an RI signal to ON

Macro UART_riOn

Arguments None

Description Set RI signal on.

Example

```
UART_riOn;
```

UART_dsrOff

Sets a DSR signal to OFF

Macro UART_dsrOff

Arguments None

Description Set DSR signal off.

Example

```
UART_dsrOff;
```

UART_dsrOn

Sets a DSR signal to ON

Macro UART_dsrOn

Arguments None

Description Set DSR signal on.

Example

```
UART_dsrOn;
```

UART_isDtr

Verifies that DTR is ON

Macro UART_isDtr

Arguments Nobe

Description Check if DTR is on. Return DTR value.

Example CSLBool dtrSignal;

```
dtrSignal = UART_isDtr;
```

15.5 Configuring the UART Module Using CSL GUI

The Universal Asynchronous Receiver/Transmitter (UART) controller is the key component of the serial communications subsystem of a computer. Asynchronous transmission allows data to be transmitted without a clock signal to the receiver.

The configuration manager allows creation of one or more configuration objects. The configuration objects contain all of the data necessary to set the UART Control Registers.

The resource manager allows the user to choose a configuration object to to use in configuring the UART.

Figure 15–1 illustrates the UART sections menu on the CSL graphical user interface (GUI)

Figure 15–1. UART Sections Menu

The UART includes the following two sections:

- **UART Configuration Manager** allows you to create configuration objects. No predefined configuration objects.
- **UART Resource Manager** allows you to select a configuration and enable that configuration within the object.

15.5.1 UART Configuration Manager

The UART Configuration Manager allows you to create UART configurations through the Properties page and to generate the configuration objects.

15.5.1.1 Inserting a Configuration Object

There is no predefined configuration object available.

To configure the UART registers, you must insert a new configuration object.

To insert a new configuration object, right-click on the UART Configuration Manager and select insert uartCfg from the drop-down menu. The configuration objects can be renamed. Their use depends upon the on-chip device resources.

Note:

The number of configuration objects is unlimited. Several configurations can be created and the user can select the right one for this application and can change the configuration later just by selecting a new one under the UART Resource Manager. The goal is to provide more flexibility and to reduce the time required to modify register values.

15.5.1.2 Deleting/Renaming a Configuration Object

To delete or to rename an object, right-click on the configuration object you want to delete or rename. Select Delete to delete a configuration object. Select Rename to rename the object.

If the Delete and Rename options are grayed out and non-usable, use the UART Resource Manager to configure the UART. The Show Dependency option is accessible and shows which device is using the configuration object (see Figure 2–1, *The CSL Tree,* on page 2-3).

15.5.1.3 Configuring the Object Properties

The Properties pages allow you to set the UART registers (see Figure 15–2). 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.

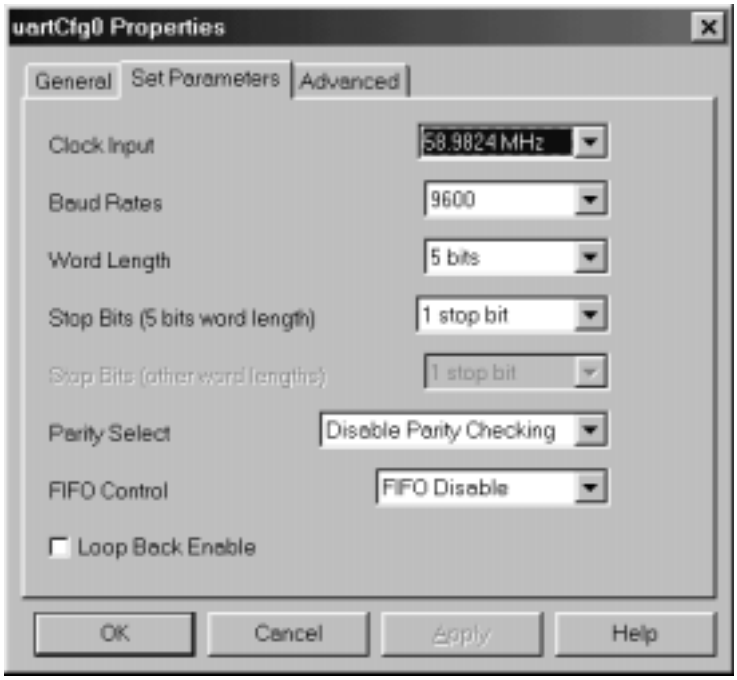
The Properties pages allow you to set the time, date, and alarm registers of the UART. You can set the configuration options through the following pages:

- ❏ Setup: Allows you to configure the UART based on a set of parameters.
- ❏ Advanced: Allows you to configure the values of the previous pages by directly setting the UART registers.

  This page contains the full 8-bit register values; the full register values can be entered directly and the new options will be mirrored on the previous page automatically.

Figure 15–2 depicts the properties page.

Figure 15–2. UART Properties Page



Each Tab page is composed of several options that are set to a default value (at device reset).

15.5.2 UART Resource Manager

The UART Resource Manager allows you to generate the UART_config() CSL function.

Figure 15–3 illustrates the UART Resource Manager menu on the CSL graphical user interface (GUI).

Figure 15–3. UART Resource Manager Menu

15.5.2.1 Properties Page

You can generate the UART_Config() CSL function through the Properties page.

To access the Properties page, right-click on the predefined UART peripheral and select Properties from the drop-down menu (see Figure 15–4).

The first time the Properties page appears, only the Enable UART Configuration check-box can be selected. Select this to enable the UART configuration.

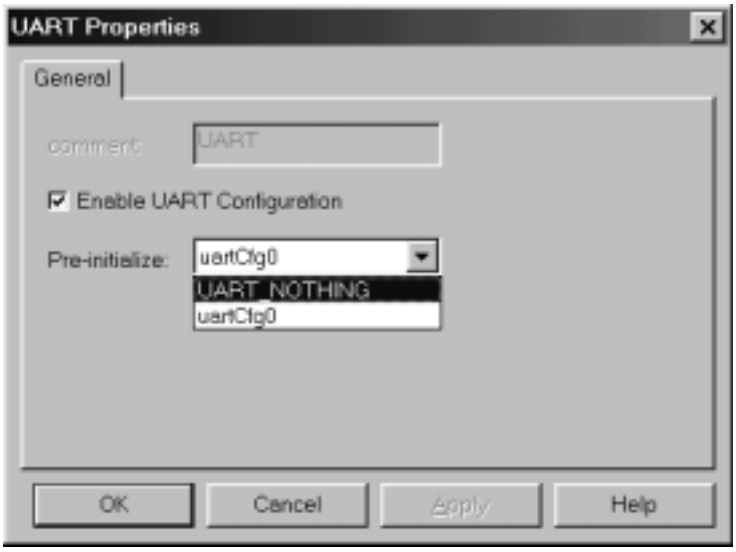
UART_NOTHING is used to indicate that there is no configuration object selected for this serial port.

To pre-initialize the UART peripheral, check the Enable UART Configuration box. You can then select one of the available configuration objects (see section 15.5.1) for this peripheral through the pre-initialize drop-down list.

If UART_NOTHING remains selected, the UART_config() function will not be generated for the UART. (see section 15.5.3).

In the example shown in Figure 15–4, the uartCfg0 is selected, UART and the UART_config() function will be generated, thereby setting the UART to UARTfg0 configuration settings.

Figure 15–4. UART Resource Manager Properties Page



15.5.3 C Code Generation for UART Module

The two C files generated from the configuration tool are a header file and a source file.

15.5.3.1 Header File

The header file includes all the csl header files of the modules and contains the configuration objects defined from the configuration tool (see Example 15–1).

Example 15–1. UART Header File

```
extern UART_Config uartCfg0;
```

15.5.3.2 Source File

The source file includes the declaration of the handle object and the configuration structures (see Example 15–2).

Example 15–2. UART Source File (Declaration Section)

```
/*  Config Structures */

UART_Config uartCfg0 = {

   0x80,     /* Divisor Latch (URDLL – LSB)    */
   0x01,     /* Divisor Latch (URDLL – MSB)    */
   0x00,     /* Line Control Register (URLCR)  */
   0x00,     /* FIFO Control Register (URFCR)  */
   0x00      /* Modem Control Register (URMCR) */

};
```

The source file contains the Configuration Pre-Initialization using the CSL UART API function UART_config() (see Example 15–3).

This function is encapsulated into a unique function, CSL_cfgInit(), which is called from your main C file. UART_config() is generated only if Enable UART configuration (with a selected configuration other than UART_NOTHING) is checked under the UART Resource Manager Properties page.

Example 15–3. UART Source File (Body Section)

```
void CSL_cfgInit()

{

   UART_config(&uartCfg0);

}
```

Chapter 16

WDTIM Module

This chapter lists the configuration structure, functions, and macros available for use with the WDTIM modules.

Topic **Page**

16.1 Overview

Table 16–1 and Table 16–2 list the configuration structures and functions used with the WDTIM module.

Table 16–1. WDTIM Configuration Structure

Structure	Purpose	See page...
WDTIM_Config	WDTIM configuration structure used to setup a watchdog timer device	16-3

Table 16–2. WDTIM Functions

Function	Purpose	See page ...
WDTIM_config	Sets up the WDTIM register using the configuration structure	16-4
WDTIM_configArgs	Sets up the WDTIM using the register values passed in	16-4
WDTIM_getConfig	Reads the current register values of the watchdog timer and stores the result in the configuration structure	16-5
WDTIM_service	Writes to the watchdog key of the timer	16-5
WDTIM_start	Starts the WDTIM device running	16-5

16.2 Configuration Structure

This section lists the structure in the WDTIM module.

WDTIM_Config *WDTIM configuration structure used to setup timer device*

Structure WDTIM_Config

Members

Uint16 wdtcr	control
Uint16 wdtscr	secondary control
Uint16 wdprd	period

Description The WDTIM configuration structure is used to setup a watchdog timer device. You create and initialize this structure then pass its address to the WDTIM_config() function. You can use literal values or the WDTIM_RMK macros to create the structure member values.

Example

```
WDTIM_Config MyConfig = {
  0x0000, /* control          */
  0x1000, /* secondary control */
  0x1000  /* period           */
  };
…
WDTIM_config(&MyConfig);
```

16.3 Functions

This section lists the functions in the WDTIM module.

WDTIM_config — *Sets up WDTIM register using configuration structure*

Function

```
void WDTIM_config(
    WDTIM_Config *Config
);
```

Arguments

config	Pointer to an initialized configuration structure

Return Value

None

Description

Sets up the WDTIM register using the configuration structure. The values of the structure are written to the registers TCR, PRD, and TIM (see also WDTIM_configArgs() and WDTIM_Config).

Example

```
WDTIM_Config MyConfig = {
};
…
WDTIM_config(&MyConfig);
```

WDTIM_configArgs — *Sets up WDTIM using register values passed in*

Function

```
void WDTIM_configArgs(
    Uint16 wdtcr,
    Uint16 wdtscr,
    Uint16 wdprd
);
```

Arguments

wdtcr	Control register value
wdtscr	Secondary Control register value
wdprd	Period register value

Return Value

None

Description

Sets up the timer using the register values passed in. The register values are written to the timer registers. The timer control register (wdtcr) is written last (see also WDTIM_config()).

You may use literal values for the arguments or for readability, you may use the WDTIM_RMK macros to create the register values based on field values.

Example

```
WDTIM_configArgs (
   0x0010, /* wdtcr  */
   0x0000, /* wdtscr */
   0x1000 /* wdprd  */
);
```

WDTIM_getConfig

Gets the WDTIM configuration structure for a specified device

Function

```
void WDTIM_getConfig(
    WDTIM_Config *Config
);
```

Arguments

Config Pointer to a WDTIM configuration structure

Return Value

None

Description

Gets the WDTIM configuration structure for a specified device.

Example

```
WDTIM_Config MyConfig;
WDTIM_getConfig(&MyCOnfig);
```

WDTIM_service

Writes to the watchdog key of the timer

Function

```
void WDTIM_service(
);
```

Arguments

None

Return Value

None

Description

Services the watchdog timer by writing a sequence of A5C6h, followed by an A7Eh to the WDKEY field of the WDTSCR register, before the watchdog timer times out. **This function must be called periodically to prevent a watchdog timeout.**

Example

```
WDTIM_service();
```

WDTIM_start

Starts WDTIM device running

Function

```
void WDTIM_start(
);
```

Arguments

None

Return Value

None

Description

Starts the timer device running. TSS field =0.

Example

```
WDTIM_start();
```

16.4 Macros

CSL offers a collection of macros to access CPU control registers and fields. For additional details (see section 1.5).

Table 16–3 lists the available WDTIM macros.

Table 16–3 uses the following conventions:

To use the WDTIM macros, include csl_wdtim.h and follow these restrictions:

- Only writable fields are allowed
- Values should be a right-justified constants.
- If *fieldval_n* value exceeds the number of bits allowed for that field, *fieldval_n* is truncated accordingly

For examples that are similar to the WDTIM macros, see section 6.4 in the DMA chapter or section 11.4 in the McBSP chapter.

Table 16–3. WDTIM CSL Macros Using Timer Port Number

(a) Macros to read/write WDTIM register values

Macro	Syntax
WDTIM_RGET()	Uint16 WDTIM_RGET(*REG*)
WDTIM_RSET()	void WDTIM_RSET(*REG*, Uint16 *regval*)

(b) Macros to read/write WDTIM register field values (Applicable only to registers with more than one field)

Macro	Syntax
WDTIM_FGET()	Uint16 WDTIM_FGET(*REG*, *FIELD*)
WDTIM_FSET()	void WDTIM_FSET(*REG*, *FIELD*, Uint16 *fieldval*)

(c) Macros to create value to write to WDTIM registers and fields (Applicable only to registers with more than one field)

Macro	Syntax
WDTIM_*REG*_RMK()	Uint16 WDTIM_*REG*_RMK(*fieldval_n*,...*fieldval_0*) Note: *Start with field values with most significant field positions: field_n: MSB field field_0: LSB field * only writable fields allowed
WDTIM_FMK()	Uint16 WDTIM_FMK(*REG*, *FIELD*, *fieldva*l)

(d) Macros to read a register address

Macro	Syntax
WDTIM_ADDR()	Uint16 WDTIM_ADDR(*REG*)

Notes: 1) *REG* indicates the register, WDTCR, WDPRD, WDTSCR, or WDTIM**.**

2) *FIELD* indicates the register field name as specified in Appendix A.
   - For *REG*_FSET and *REG*_FMK, *FIELD* must be a writable field.
   - For *REG*_FGET, the field must be a readable field.

3) *regval* indicates the value to write in the register (*REG*)

4) *fieldval* indicates the value to write in the field *(FIELD*)

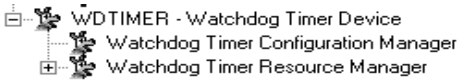
16.5 Configuring the WATCHDOG TIMER Module Using CSL GUI

16.5.1 Overview

The WATCHDOG TIMER module facilitates configuration/control of the on-chip WATCHDOG TIMER. The WATCHDOG TIMER module consists of a configuration manager and a resource manager. The configuration manager allows the creation of one or more configuration objects. The configuration object consists of the necessary data to set the WATCHDOG TIMER control registers. The resource manager associates a selected configuration with a timer.

Figure 16–1 illustrates the WATCHDOG TIMER sections menu on the CSL graphical user interface (GUI).

Figure 16–1. WATCHDOG TIMER Sections Menu



The WATCHDOG TIMER includes the following two sections:

- **WATCHDOG TIMER Configuration Manager**: Allows you to create configuration objects. (There are no predefined configuration objects.)
- **WATCHDOG TIMER Resource Manager**: Allows you to associate a configuration object to the Watchdog Timer. The WATCHDOG TIMER is only available in the TMS320C5440 and TMS320C5441 devices.

16.5.2 WATCHDOG TIMER Configuration Manager

The WATCHDOG TIMER Configuration Manager allows you to create device configurations through the Properties page and generate the configuration objects.

16.5.2.1 Creating/Inserting a configuration

There are no predefined configuration objects available.

To configure a WATCHDOG TIMER device through the peripheral, you must insert a new configuration object.

To insert a new configuration object, right-click on the WATCHDOG TIMER Configuration Manager and select Insert wdtimCfg from the drop-down menu. The configuration objects can be renamed. Their use depends on the on-chip device resources.

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 WATCHDOG TIMER Resource Manager. This feature provides you with more flexibility and reduces the time required to modify register values.

16.5.2.2 Deleting/Renaming an Object

To delete or to rename an object, right-click on the configuration object you want to delete or rename. Select Delete to delete a configuration object. Select Rename to rename the object.

If a configuration object is used by one of the predefined handle objects of the WATCHDOG TIMER 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 Figure 2–1, *The CSL Tree,* on page 2-3).

16.5.2.3 Configuring the Object Properties

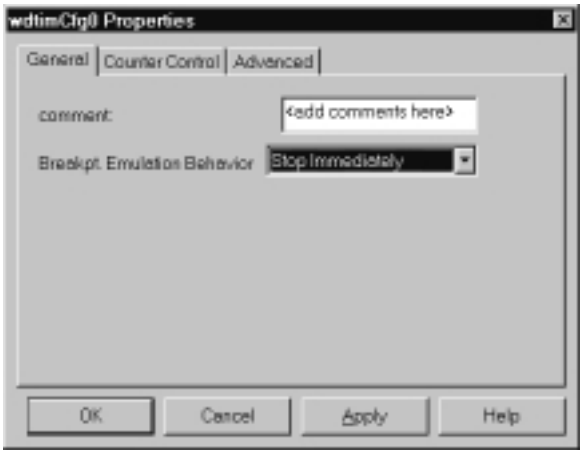
You can configure object properties through the Properties dialog box (see Figure 16–2). To access the Properties dialog box, right-click on a configuration object and select Properties. By default, the General page of the Properties dialog box is displayed.

The Properties pages allow you to set the Peripheral registers related to the WATCHDOG TIMER. You can set the configuration options through the following tab pages:

- ❏ General: Allows you to configure the Breakpoint Emulation
- ❏ Counter Control: Allows you to configure the Breakpoint Emulation Counter configuration
- ❏ Advanced Page: Allows you to configure the Summary of the previous two pages
- ❏ This page contains the full hexadecimal register values and reflects the setting of the two pages
- ❏ The full register values can be entered directly and the new options will be mirrored on the previous two pages automatically

Figure 16–2 depicts the Properties Page dialog box.

Figure 16–2. WATCHDOG TIMER Properties Page



Each Tab page is composed of several options that are set to a default value (at device reset).

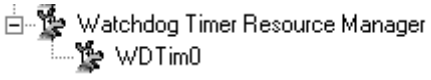
The options represent the fields of the WATCHDOG TIMER registers; the associated field name is shown in parenthesis.

16.5.3 WATCHDOG TIMER Resource Manager

The WATCHDOG TIMER Resource Manager allows you to generate the WDTIM_config() CSL function.

Figure 16–3 illustrates the WATCHDOG TIMER Resource Manager Menu.

Figure 16–3. WATCHDOG TIMER Resource Manager Menu



16.5.3.1 Properties Page

You can generate the WDTIM_config() csl function through the Properties page.

To access the Properties page, right-click on a predefined TIMER handle object and select Properties from the drop-down menu (see Figure 16–4).

The first time the properties page appears, only the Enable Configuration of WATCHDOG TIMER check-box can be selected. Select this to open the WATCHDOG TIMER configuration, allowing pre-initialization.

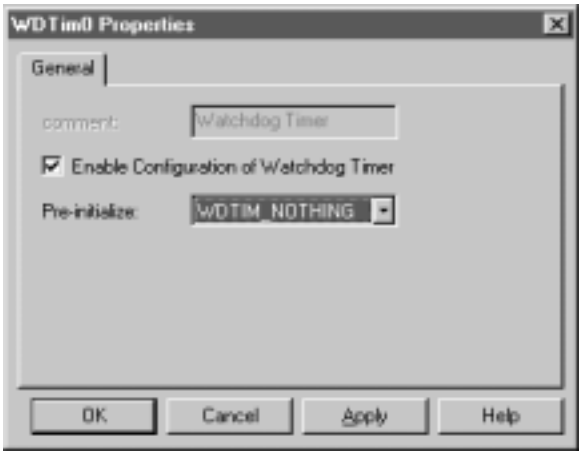
WDTIM_NOTHING is used to indicate that there is no configuration object selected for this device.

To pre-initialize the Watchdog Timer, check the Enable Configuration of WATCHDOG TIMER box. One of the available configuration objects (see section 16.5.2) can then be selected for this channel through the Pre-Initialize drop-down list.

If WDTIM_NOTHING remains selected, no WDTIM_config() function call will be generated for the WATCHDOG TIMER. (See section 16.5.4.)

In Figure 16–4, the Enable Configuration of Watchdog Timer option is checked and wdtimCfg0 is now accessible. The WDTIM_config() function will be generated.

Figure 16–4. WATCHDOG TIMER Properties Page



16.5.4 C Code Generation for WATCHDOG TIMER

Two C files are generated from the configuration tool:

- ❑ Header file
- ❑ Source file.

16.5.4.1 Header File

The header file includes all the csl header files of the modules and contains the WATCHDOG TIMER configuration objects defined from the configuration tool (see Example 16–1).

Example 16–1. WATCHDOG TIMER Header File

```
extern WDTIM_Config wdtimCfg0;
```

16.5.4.2 Source File

The source file includes the declaration of the configuration structures (see Example 16–2).

Example 16–2. WATCHDOG TIMER Source File (Declaration Section)

```
/*  Config Structures */
WDTIM_Config wdtimCfg0 = {
    0x0000,         /*  Timer Control Register   */
    0x0000,         /*  Timer Period Register   */
    0x0000        /*  Timer Scale Register (TSCR)  */
};
```

The source file contains the Configuration Pre-Initialization using the CSL WATCHDOG TIMER API WDTIM_config() (see Example 16–3). This function is encapsulated into a unique function, CSL_cfgInit(), which is called from your main C file.

WDTIM_config() will be generated only if Enable Configuration of WATCHDOG TIMER is checked and a configuration other than WDTIM_NOTHING is selected on the Watchdog Timer Resource Manager Properties page.

Note:

You must use WDTIM_start() in your code to start the WATCHDOG TIMER. The WDTIM_Config structure generated by the CSL GUI does not start the WATCHDOG TIMER port.

Example 16–3. WATCHDOG TIMER Source File (Body Section)

```
    void CSL_cfgInit()
    {

       WDTIM_config(&wdtimCfg0);
    }
```

Appendix A

Peripheral Registers

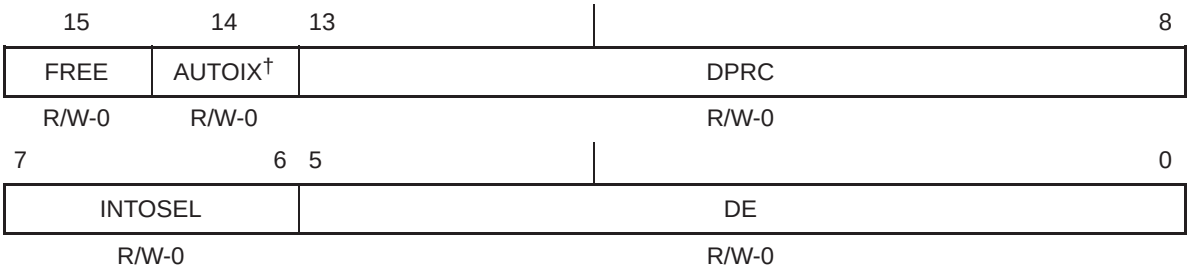
This appendix provides symbolic constants for the peripheral registers.

Topic **Page**

A.1 DMA Registers

A.1.1 DMA Channel Priority and Enable Control Register (DMPREC)

Figure A–1. DMA Channel Priority and Enable Control Register (DMPREC)



† Only available on specific devices.

Legend: R/W-x = Read/Write-Reset value

Table A–1. DMA Channel Priority and Enable Control Register (DMPREC) Field Values (DMA_DMPREC_field_symval)

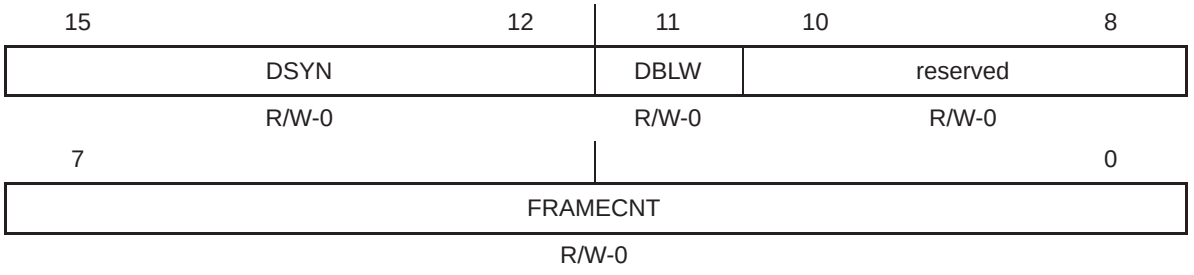
Bit	*field*	*symval*	Value	Description
15	FREE			Controls the behavior of the DMA controller during emulation.
		OFF	0	DMA transfers are suspended when the emulator stops
		ON	1	DMA transfers continue even during emulation stop
14	AUTOIX			**For C5409A, C54010A, C5416, C5440, and C5441:** Selects which DMA global reload registers are used to reload the DMA channels.
		USE_DMA0	0	All DMA channels use DMGSA0, DMGDA0, DMGCR0, and DMGFR0 as their reload registers.
		USE_CHAN	1	Each DMA channel uses its local set of reload registers during autoinitialization mode.
13-8	DPRC	OF(*value*)	0-63	DMA channel priority control bit. Each bit specifies the priority of a DMA channel. When the bit is cleared to 0, the channel is a low priority; when the bit is set to 1, the channel is a high priority.
7-6	INTOSEL			Interrupt multiplex control bits. The INTOSEL bits control how the DMA interrupts are assigned in the interrupt vector table and IMR/IMF registers. The effects of this field on the operation are device-specific.
				For C5401, C5402, C5409, C5409A, C5420, 5421, and 5471
		NONE	00	Interrupts available: Timer 1, McBSP 1 RINT/XINT
		CH2_CH3	01	Interrupts available: Timer 1, DMA channel 2, DMA channel 3
		CH0_TO_CH3	10	Interrupts available: DMA channel 0, DMA channel 1, DMA channel 2, DMA channel 3

Table A–1. DMA Channel Priority and Enable Control Register (DMPREC) Field Values (DMA_DMPREC_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
			11	Reserved
	INTOSEL			**For C5410, C5410A, C5416, and C5420 54CST, 5404, 5407:**
		CH4_CH5	00	Interrupts available: McBSP 0 RINT/XINT, McBSP 1 RINT/XINT, McBSP 2 RINT/XINT, DMA channel 4, DMA channel 5
		CH2_TO_CH5	01	Interrupts available: McBSP 0 RINT/XINT, McBSP 2 RINT/XINT, DMA channel 2, DMA channel 3, DMA channel 4, DMA channel 5
		CH0_TO_CH5	10	Interrupts available: McBSP 0 RINT/XINT, DMA channel 0, DMA channel 1, DMA channel 2, DMA channel 3, DMA channel 4, DMA channel 5
			11	Reserved
5-0	DE	OF(*value*)	0-63	DMA channel enable bit. Each bit enables a DMA channel. When the bit is cleared to 0, the channel is disabled; when the bit is set to 1, the channel is enabled.

A.1.2 DMA Channel n Sync Select and Frame Count Register (DMSFCn)

Figure A–2. DMA Channel n Sync Select and Frame Count Register (DMSFCn)



Legend: R/W-x = Read/Write-Reset value

Table A–2. DMA Channel n Sync Select and Frame Count Register (DMSFCn) Field Values (DMA_DMSFC_field_symval)

Bit	*field*	*symval*	Value	Description
15-12	DSYN			DMA sync event. Specifies which sync event is used to initiate DMA transfers for the corresponding DMA channel. The effects of this field on the operation are device-specific.
				For C5401 and C5402:
		NONE	0000	No sync event (nonsynchronization operation)
		REVT0	0001	McBSP 0 receive event (REVT0)

Table A–2. DMA Channel n Sync Select and Frame Count Register (DMSFCn) Field Values (DMA_DMSFC_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
		XEVT0	0010	McBSP 0 transmit event (XEVT0)
			0011	Reserved
			0100	Reserved
	DSYN	REVT1	0101	McBSP 1 receive event (REVT1)
		XEVT1	0110	McBSP 1 transmit event (XEVT1)
			0111-1100	Reserved
		TINT0	1101	Timer 0 interrupt event
		INT3	1110	External interrupt 3 event
		TINT1	1111	Timer 1 interrupt event
				For C5409, C5409A, and C5471:
		NONE	0000	No sync event (nonsynchronization operation)
		REVT0	0001	McBSP 0 receive event (REVT0)
		XEVT0	0010	McBSP 0 transmit event (XEVT0)
		REVT2	0011	McBSP 2 receive event (REVT2)
		XEVT2	0100	McBSP 2 transmit event (XEVT2)
		REVT1	0101	McBSP 1 receive event (REVT1)
		XEVT1	0110	McBSP 1 transmit event (XEVT1)
			0111-1100	Reserved
		TINT0	1101	Timer interrupt event
		INT3	1110	External interrupt 3 event
				For C5410, C5410A, and C5416:
		NONE	0000	No sync event (nonsynchronization operation)
		REVT0	0001	McBSP 0 receive event (REVT0)
		XEVT0	0010	McBSP 0 transmit event (XEVT0)
		REVT2	0011	McBSP 2 receive event (REVT2)
		XEVT2	0100	McBSP 2 transmit event (XEVT2)
		REVT1	0101	McBSP 1 receive event (REVT1)
		XEVT1	0110	McBSP 1 transmit event (XEVT1)
		REVTA0	0111	McBSP 0 receive event — ABIS mode (REVTA0)
		XEVTA0	1000	McBSP 0 transmit event — ABIS mode (XEVTA0)

Table A–2. DMA Channel n Sync Select and Frame Count Register (DMSFCn) Field Values (DMA_DMSFC_field_symval) (Continued)

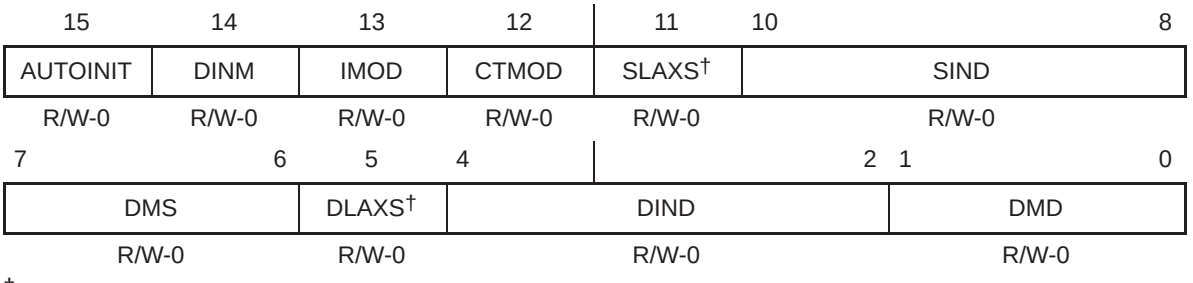
Bit	*field*	*symval*	Value	Description
		REVTA2	1001	McBSP 2 receive event — ABIS mode (REVTA2)
		XEVTA2	1010	McBSP 2 transmit event — ABIS mode (XEVTA2)
	DSYN	REVTA1	1011	McBSP 1 receive event — ABIS mode (REVTA1)
		XEVTA1	1100	McBSP 1 transmit event — ABIS mode (XEVTA1)
		TINT0	1101	Timer interrupt event
		INT3	1110	External interrupt 3 event
			1111	Reserved
				For C5420 and C5421:
		NONE	0000	No sync event (nonsynchronization operation)
		REVT0	0001	McBSP 0 receive event (REVT0)
		XEVT0	0010	McBSP 0 transmit event (XEVT0)
		REVT2	0011	McBSP 2 receive event (REVT2)
		XEVT2	0100	McBSP 2 transmit event (XEVT2)
		REVT1	0101	McBSP 1 receive event (REVT1)
		XEVT1	0110	McBSP 1 transmit event (XEVT1)
		FIFO_REVT	0111	FIFO receive buffer not empty event
		FIFO_XEVT	1000	FIFO transmit buffer not full event
			1001-1111	Reserved
				For C5440 and C5441:
		NONE	0000	No sync event (nonsynchronization operation)
		REVT0	0001	McBSP 0 receive event (REVT0)
		XEVT0	0010	McBSP 0 transmit event (XEVT0)
		REVT2	0011	McBSP 2 receive event (REVT2)
		XEVT2	0100	McBSP 2 transmit event (XEVT2)
		REVT1	0101	McBSP 1 receive event (REVT1)
		XEVT1	0110	McBSP 1 transmit event (XEVT1)
			0111-1100	Reserved
				For C54CST, 5404, and 5407:
		NONE	0000	No sync event (nonsynchronization operation)
		REVT0	0001	McBSP 0 receive event (REVT0)

Table A–2. DMA Channel n Sync Select and Frame Count Register (DMSFCn) Field Values (DMA_DMSFC_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
		XEVT0	0010	McBSP 0 transmit event (XEVT0)
		REVT2	0011	McBSP 2 receive event (REVT2)
		XEVT2	0100	McBSP 2 transmit event (XEVT2)
		REVT1	0101	McBSP 1 receive event (REVT1)
		XEVT1	0110	McBSP 1 transmit event (XEVT1)
			0611-1100	Reserved
			1000-1101	Reserved
		UART	0111	UART interrupt event
		TINT0	1101	Timer interrupt event
		TINT1	111	Timer1 interrupt event
		TINT3	1110	External interrupt 3 event
11	DBLW			Double-word mode enable bit.
		OFF	0	Single-word mode. DMA transfers 16-bit words.
		ON	1	Double-word mode. Allows the DMA to transfer 32-bit words in any index mode. Two consecutive 16-bit transfers are initiated and the source and destination addresses are automatically updated following each transfer.
10-8	reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
7-0	FRAMECNT	OF(*value*)	0-255	Frame count. Specifies the number of frames to be included in a block transfer. The frame count is initialized to 1 less than the desired number of frames.

A.1.3 DMA Channel n Transfer Mode Control Register (DMMCRn)

Figure A–3. DMA Channel n Transfer Mode Control Register (DMMCRn)



† Only available on specific devices with DMA extended data memory.

Legend: R/W-x = Read/Write-Reset value

Table A–3. DMA Channel n Transfer Mode Control Register (DMMCRn) Field Values (DMA_DMMCR_field_symval)

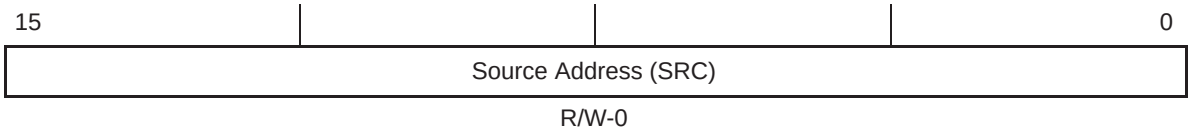
Bit	*field*	*symval*	Value	Description
15	AUTOINIT			DMA autoinitialization mode enable bit.
		OFF	0	Autoinitialization is disabled.
		ON	1	Autoinitialization is enabled.
14	DINM			DMA interrupt generation mask bit.
		OFF	0	No interrupt is generated
		ON	1	Interrupt is generated based on IMOD bit
13	IMOD			DMA interrupt generation mode bit operates in conjunction with CTMOD bit.
				In ABU mode (CTMOD = 1):
		FULL_ONLY	0	Interrupt at buffer full only.
		HALF_AND_FULL	1	Interrupt at half full buffer and buffer full.
				In multiframe mode (CTMOD = 0):
		BLOCK_ONLY	0	Interrupt at completion of block transfer.
		FRAME_AND_BLOCK	1	Interrupt at end of frame and end of block.
12	CTMOD			DMA transfer counter mode control bit.
		MULTIFRAME	0	Multiframe mode
		ABU	1	ABU mode
11	SLAXS			**For devices with DMA extended data memory:** DMA source space select bit.
		OFF	0	No external access
		ON	1	External access
10-8	SIND			DMA source address transfer index mode bit.
		NOMOD	000	No modification
		POSTINC	001	Postincrement
		POSTDEC	010	Postdecrement
		DMIDX0	011	Postincrement with index offset (DMIDX0)
		DMIDX1	100	Postincrement with index offset (DMIDX1)
		DMFRI0	101	Postincrement with index offset (DMIDX0 and DMFRI0)
		DMFRI1	110	Postincrement with index offset (DMIDX1 and DMFRI1)
			111	Reserved

Table A–3. DMA Channel n Transfer Mode Control Register (DMMCRn) Field Values (DMA_DMMCR_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
7-6	DMS			DMA source address space select bit.
		PROGRAM	00	Program space
		DATA	01	Data space
		IO	10	I/O space
			11	Reserved
5	DLAXS			**For devices with DMA extended data memory:** DMA destination space select bit.
		OFF	0	No external access
		ON	1	External access
4-2	DIND			DMA destination address transfer index mode bit.
		NOMOD	000	No modification
		POSTINC	001	Postincrement
		POSTDEC	010	Postdecrement
		DMIDX0	011	Postincrement with index offset (DMIDX0)
		DMIDX1	100	Postincrement with index offset (DMIDX1)
		DMFRI0	101	Postincrement with index offset (DMIDX0 and DMFRI0)
		DMFRI1	110	Postincrement with index offset (DMIDX1 and DMFRI1)
			111	Reserved
1-0	DMD			DMA destination address space select bit.
		PROGRAM	00	Program space
		DATA	01	Data space
		IO	10	I/O space
			11	Reserved

A.1.4 DMA Channel n Source Address Register (DMSRCn)

Figure A–4. DMA Channel n Source Address Register (DMSRCn)



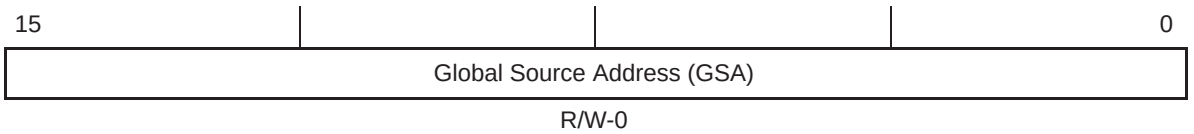
Legend: R/W-x = Read/Write-Reset value

Table A–4. DMA Channel n Source Address Register (DMSRCn) Field Values (DMA_DMSRC_field_symval)

Bit	*field*	*symval*	Value	Description
15-0	SRC	OF(*value*)	0-FFFFh	Specifies the 16 least-significant bits of the extended address for the source location. The source address register is initialized prior to starting the DMA transfer in software, and updated automatically during transfers by the DMA controller.

A.1.5 DMA Global Source Address Reload Register (DMGSA)

Figure A–5. DMA Global Source Address Reload Register (DMGSA)



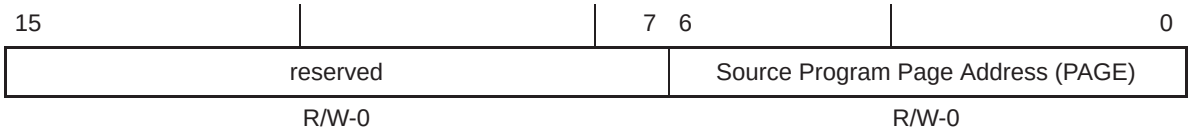
Legend: R/W-x = Read/Write-Reset value

Table A–5. DMA Global Source Address Reload Register (DMGSA) Field Values (DMA_DMGSA_field_symval)

Bit	*field*	*symval*	Value	Description
15-0	GSA	OF(*value*)	0-FFFFh	A 16-bit source address used to reload DMSRCn.

A.1.6 DMA Source Program Page Address Register (DMSRCP)

Figure A–6. DMA Source Program Page Address Register (DMSRCP)



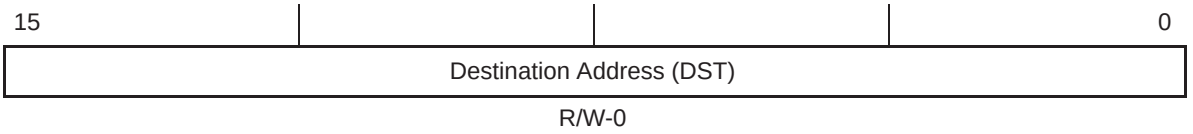
Legend: R/W-x = Read/Write-Reset value

Table A–6. DMA Source Program Page Address Register (DMSRCP) Field Values (DMA_DMSRCP_field_symval)

Bit	*field*	*symval*	Value	Description
15-7	reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
6-0	PAGE	OF(*value*)	0-127	Specifies the 7 most-significant bits of the extended program page address for the source location.

A.1.7 DMA Channel n Destination Address Register (DMDSTn)

Figure A–7. DMA Channel n Destination Address Register (DMDSTn)



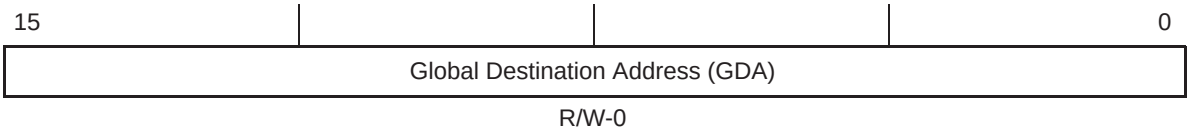
Legend: R/W-x = Read/Write-Reset value

Table A–7. DMA Channel n Destination Address Register (DMDSTn) Field Values (DMA_DMDST_field_symval)

Bit	*field*	*symval*	Value	Description
15-0	DST	OF(*value*)	0-FFFFh	Specifies the 16 least-significant bits of the extended address for the destination location. The destination address register is initialized prior to starting the DMA transfer in software, and updated automatically during transfers by the DMA controller.

A.1.8 DMA Global Destination Address Reload Register (DMGDA)

Figure A–8. DMA Global Destination Address Reload Register (DMGDA)



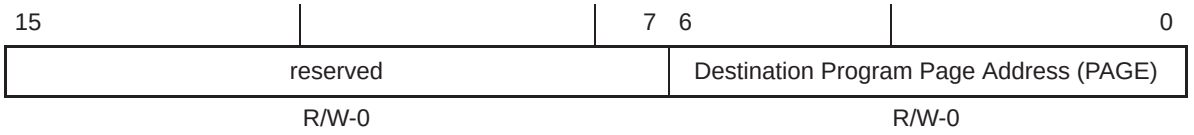
Legend: R/W-x = Read/Write-Reset value

Table A–8. DMA Global Destination Address Reload Register (DMGDA) Field Values (DMA_DMGDA_field_symval)

Bit	*field*	*symval*	Value	Description
15-0	GDA	OF(*value*)	0-FFFFh	A 16-bit destination address used to reload DMDSTn.

A.1.9 DMA Destination Program Page Address Register (DMDSTP)

Figure A–9. DMA Destination Program Page Address Register (DMDSTP)



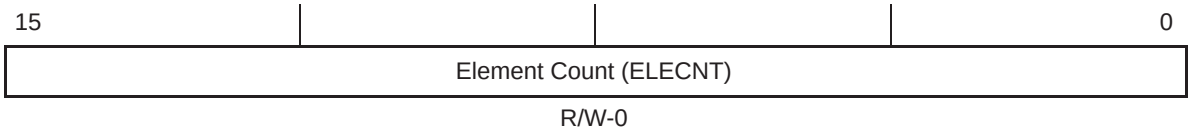
Legend: R/W-x = Read/Write-Reset value

Table A–9. DMA Destination Program Page Address Register (DMDSTP) Field Values (DMA_DMDSTP_field_symval)

Bit	*field*	*symval*	Value	Description
15-7	reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
6-0	PAGE	OF(*value*)	0-127	Specifies the 7 most-significant bits of the extended program page address for the destination location.

A.1.10 DMA Channel n Element Count Register (DMCTRn)

Figure A–10. DMA Channel n Element Count Register (DMCTRn)



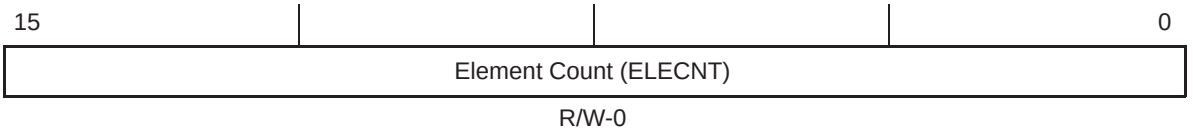
Legend: R/W-x = Read/Write-Reset value

Table A–10. DMA Channel n Element Count Register (DMCTRn) Field Values (DMA_DMCTR_field_symval)

Bit	*field*	*symval*	Value	Description
15-0	ELECNT	OF(*value*)	0-FFFFh	A 16-bit element counter that keeps track of the number of DMA transfers to be performed. The element count register should be initialized to 1 less than the desired number of element transfers.

A.1.11 DMA Global Element Count Reload Register (DMGCR)

Figure A–11. DMA Global Element Count Reload Register (DMGCR)



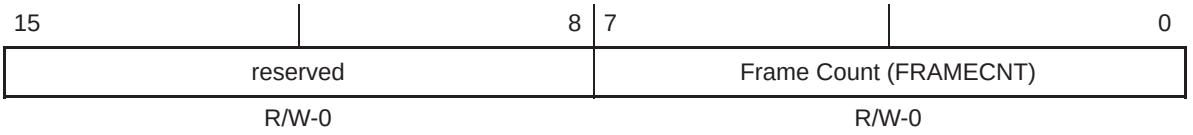
Legend: R/W-x = Read/Write-Reset value

Table A–11. DMA Global Element Count Reload Register (DMGCR) Field Values (DMA_DMGCR_field_symval)

Bit	*field*	*symval*	Value	Description
15-0	ELECNT	OF(*value*)	0-FFFFh	A 16-bit unsigned element count value used to reload DMCTR.

A.1.12 DMA Global Frame Count Reload Register (DMGFR)

Figure A–12. DMA Global Frame Count Reload Register (DMGFR)



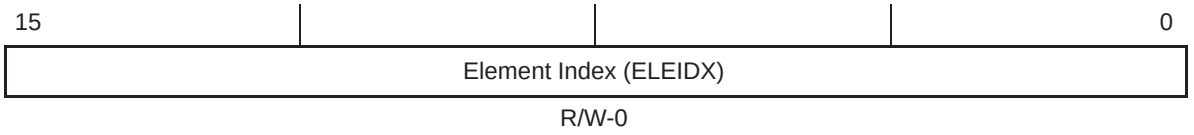
Legend: R/W-x = Read/Write-Reset value

Table A–12. DMA Global Frame Count Reload Register (DMGFR) Field Values (DMA_DMGFR_field_symval)

Bit	*field*	*symval*	Value	Description
15-8	reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
7-0	FRAMECNT	OF(*value*)	0-FFFFh	An 8-bit unsigned frame count value used to reload the Frame Count field of DMSFCn.

A.1.13 DMA Element Address Index Register 0 (DMIDX0)

Figure A–13. DMA Element Address Index Register 0 (DMIDX0)



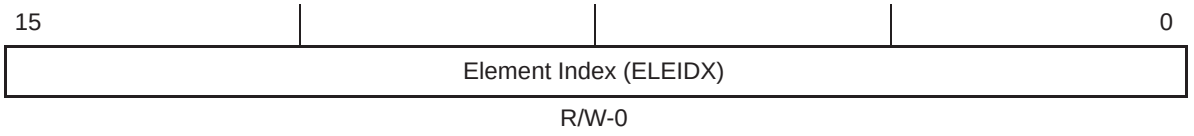
Legend: R/W-x = Read/Write-Reset value

Table A–13. DMA Element Address Index Register 0 (DMIDX0) Field Values (DMA_DMIDX0_field_symval)

Bit	*field*	*symval*	Value	Description
15-0	ELEIDX	OF(*value*)	0-FFFFh	A 16-bit unsigned index value used to modify the source or destination address following the transfer of each element.

A.1.14 DMA Element Address Index Register 1 (DMIDX1)

Figure A–14. DMA Element Address Index Register 1 (DMIDX1)



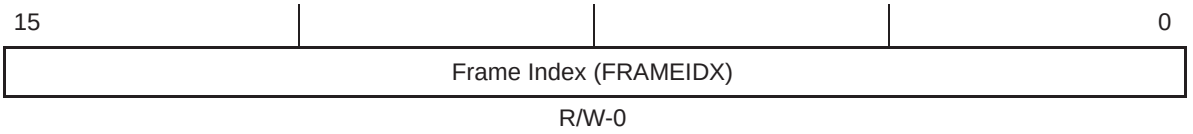
Legend: R/W-x = Read/Write-Reset value

Table A–14. DMA Element Address Index Register 1 (DMIDX1) Field Values (DMA_DMIDX1_field_symval)

Bit	*field*	*symval*	Value	Description
15-0	ELEIDX	OF(*value*)	0-FFFFh	A 16-bit unsigned index value used to modify the source or destination address following the transfer of each element.

A.1.15 DMA Frame Address Index Register 0 (DMFRI0)

Figure A–15. DMA Frame Address Index Register 0 (DMFRI0)



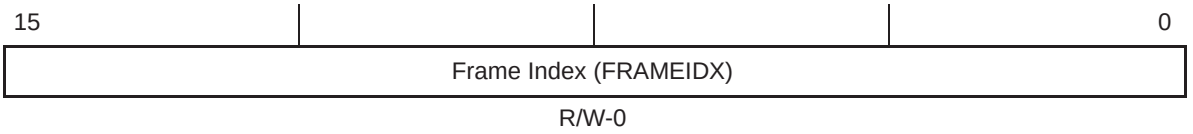
Legend: R/W-x = Read/Write-Reset value

Table A–15. DMA Frame Address Index Register 0 (DMFRI0) Field Values (DMA_DMFRI0_field_symval)

Bit	*field*	*symval*	Value	Description
15-0	FRAMEIDX	OF(*value*)	0-FFFFh	A 16-bit unsigned index value used to modify the source or destination address following the completion of blocks (or frames) of element transfers. When both element and frame indexes are used, the address is modified by the element index after each transfer and then modified by the frame index at the end of each frame.

A.1.16 DMA Frame Address Index Register 1 (DMFRI1)

Figure A–16. DMA Frame Address Index Register 1 (DMFRI1)



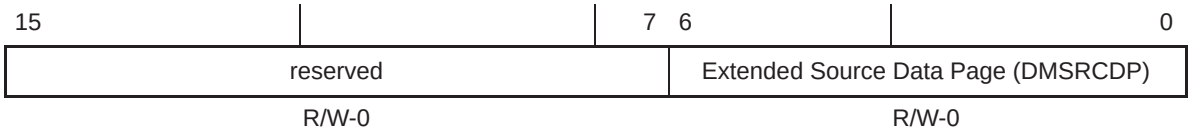
Legend: R/W-x = Read/Write-Reset value

Table A–16. DMA Frame Address Index Register 1 (DMFRI1) Field Values (DMA_DMFRI1_field_symval)

Bit	*field*	*symval*	Value	Description
15-0	FRAMEIDX	OF(*value*)	0-FFFFh	A 16-bit unsigned index value used to modify the source or destination address following the completion of blocks (or frames) of element transfers. When both element and frame indexes are used, the address is modified by the element index after each transfer and then modified by the frame index at the end of each frame.

A.1.17 DMA Global Extended Source Data Page Register (DMSRCDP)

Figure A–17. DMA Global Extended Source Data Page Register (DMSRCDP)



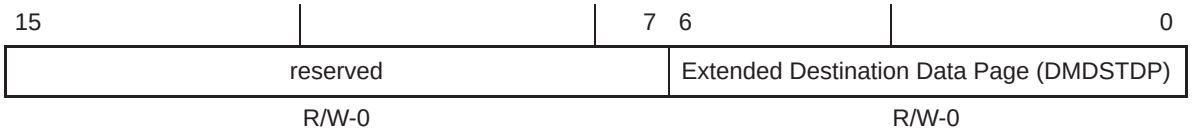
Legend: R/W-x = Read/Write-Reset value

Table A–17. DMA Global Extended Source Data Page Register (DMSRCDP) Field Values (DMA_DMSRCDP_field_symval)

Bit	*field*	*symval*	Value	Description
15-7	reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
6-0	DMSRCDP	OF(*value*)	0-127	Specifies 1 of the 128 extended source data pages.

A.1.18 DMA Global Extended Destination Data Page Register (DMDSTDP)

Figure A–18. DMA Global Extended Destination Data Page Register (DMDSTDP)



Legend: R/W-x = Read/Write-Reset value

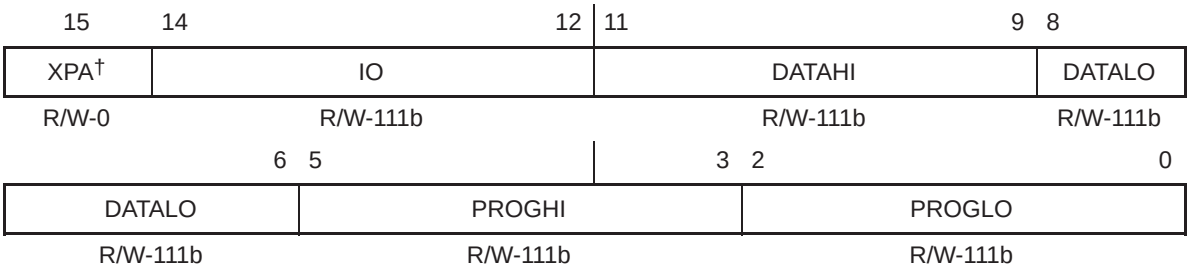
Table A–18. DMA Global Extended Destination Data Page Register (DMDSTDP) Field Values (DMA_DMDSTDP_field_symval)

Bit	*field*	*symval*	Value	Description
15-7	reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
6-0	DMDSTDP	OF(*value*)	0-127	Specifies 1 of the 128 extended destination data pages.

A.2 EBUS Registers

A.2.1 Software Wait-State Register (SWWSR)

Figure A–19. Software Wait-State Register(SWWSR)-All devices except C5440 and C5441



† XPA bit only on selected devices with extended program memory.

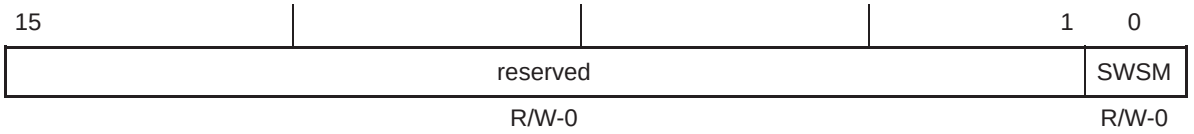
Legend: R/W-x = Read/Write-Reset value

Table A–19. Software Wait-State Register (SWWSR) Field Values (EBUS_SWWSR_field_symval)

Bit	*field*	*symval*	Value	Description
15	XPA			**For devices with extended program memory:** Extended program address control bit. Selects the address ranges selected by the program fields.
		ADDRLO	0	Address range: xx0000 - xxFFFFh
		ADDREXT	1	Address range: 000000h-7FFFFF
14-12	IO	OF(*value*)	0-7	The value corresponds to the number of wait states for I/O space 0000-FFFFh.
11-9	DATAHI	OF(*value*)	0-7	The value corresponds to the number of wait states for data space 8000-FFFFh.
8-6	DATALO	OF(*value*)	0-7	The value corresponds to the number of wait states for data space 0000-7FFFh.
5-3	PROGHI	OF(*value*)	0-7	The value corresponds to the number of wait states for program space 8000-FFFFh.
2-0	PROGLO	OF(*value*)	0-7	The value corresponds to the number of wait states for program space 0000-7FFFh.

A.2.2 Software Wait-State Control Register (SWCR)

Figure A–20. Software Wait-State Control Register (SWCR)-All devices except C5440 and C5441



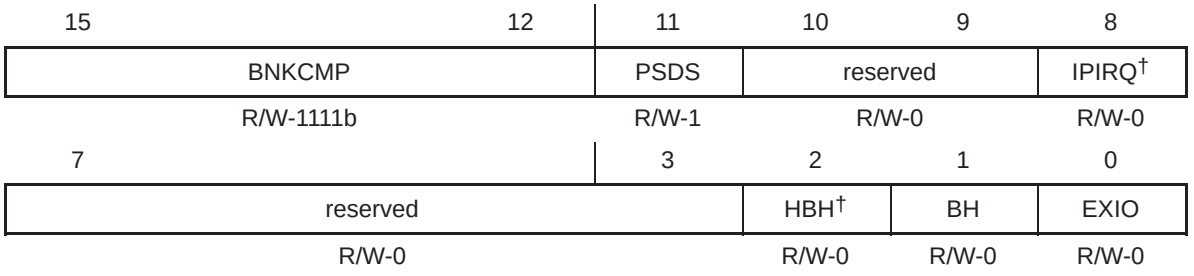
Legend: R/W-x = Read/Write-Reset value

Table A–20. Software Wait-State Control Register (SWCR) Field Values (EBUS_SWCR_field_symval)

Bit	*field*	*symval*	Value	Description
15-1	reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
0	SWSM			Software wait-state multiplier bit.
		NOMULT	0	The wait states specified in SWWSR are unchanged (not multiplied).
		MULTBY2	1	The wait states specified in SWWSR are multiplied by 2, extending the maximum number of wait states from 7 to 14.

A.2.3 Bank-Switching Control Register (BSCR)

Figure A–21. Bank-Switching Control Register (BSCR) — C5401, C5402, C5409, C5420, C5421, and 5471



† HBH, IPIRQ, and EXIO bits only on selected devices.

Legend: R/W-x = Read/Write-Reset value

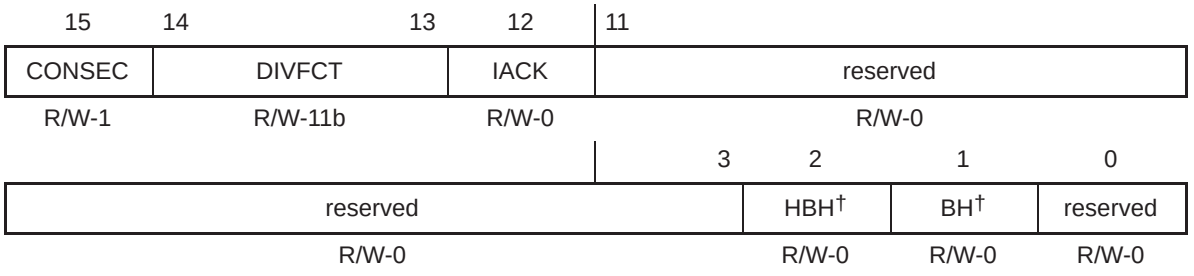
Table A–21. Bank-Switching Control Register (BSCR) Field Values — C5401, C5402, C5409, and C5420, and C5471 (EBUS_BSCR_field_symval)

Bit	*field*	*symval*	Value	Description
15-12	BNKCMP			Bank compare bit determines the number of MSBs of an address to be compared and the external memory-bank size. Bank sizes from 4K words to 64K words are allowed.
		64K	0000	No bits are compared, resulting in a bank size of 64K words.
			0001-0111	Reserved
		32K	1000	The MSB (bit 15) is compared, resulting in a bank size of 32K words.
			1001-1011	Reserved
		16K	1100	The 2 MSBs (bits 15-14) are compared, resulting in a bank size of 16K words.
			1101	Reserved
		8K	1110	The 3 MSBs (bits 15-13) are compared, resulting in a bank size of 8K words.
		4K	1111	The 4 MSBs (bits 15-12) are compared, resulting in a bank size of 4K words.
11	PSDS			Program read-data read access bit controls the insertion of an extra cycle between consecutive program and data reads, or data and program reads.
		NOEXCY	0	No extra cycles are inserted by this feature except when banks are crossed.
		INSCY	1	One extra cycle is inserted between consecutive program and data reads, or data and program reads.
10-9	reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
8	IPIRQ			**For C5420:** Interprocessor interrupt request enable bit is used to send an interprocessor interrupt to the other subsystem. IPIRQ must be cleared before any subsequent interrupts can be made.
		CLR	0	No interprocessor interrupt request is sent.
		INTR	1	An interprocessor interrupt request is sent.
7-3	reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
2	HBH			**For C5401, C5402, C5409, and 5416:** HPI data bus holder enable bit.
		DISABLE	0	The HPI data bus holder is disabled. When HPI16 pin is set to a logic high, HPI data bus holder is enabled.
		ENABLE	1	The HPI data bus holder is enabled. When not driven, the HPI data bus, HD(7-0), is held in the previous logic level.

Table A–21. Bank-Switching Control Register (BSCR) Field Values — C5401, C5402, C5409, and C5420, and C5471 (EBUS_BSCR_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
	BH			**For C5420:** Data bus holder enable bit.
		DISABLE	0	The data bus holder is disabled.
		ENABLE	1	The data bus holder is enabled. When not driven, the data bus, PPD(15-0), is held in the previous logic level.
1	BH			**For C5401, C5402, C5409, and 5416:** Bus holder enable bit.
		DISABLE	0	The bus holder is disabled. When HPI16 pin is set to a logic high, address bus holder is enabled.
		ENABLE	1	The data bus holder is enabled. When not driven, the data bus, D(15-0), is held in the previous logic level. When HPI16 pin is set to a logic high, address bus holder is enabled.
0	EXIO			Hot any call on **C5420** External bus interface off enable bit controls the external-bus-off function.
		NORMAL	0	The external-bus-off function is disabled.
		INACTIF	1	The external-bus-off function is enabled. The address bus, data bus, and control signals become inactive after completing the current bus cycle. The DROM, MP/$\overline{MC}$, and OVLY bits in PMST and the HM bit in ST1 cannot be modified.

Figure A–22. Bank-Switching Control Register (BSCR) — C5410, C5410A, and C5416



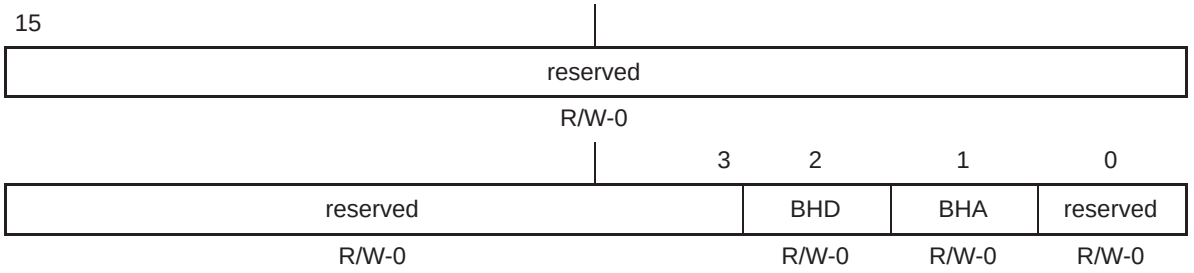
† BH and HBH bits only on selected devices.

Legend: R/W-x = Read/Write-Reset value

Table A–22. Bank-Switching Control Register (BSCR) Field Values — C5410, and C5416 (EBUS_BSCR_field_symval)

Bit	*field*	*symval*	Value	Description
15	CONSEC			Consecutive bank switching bit specifies the bank-switching mode. This bit is cleared if fast access is desired for continuous memory reads (that is, no starting and trailing cycles between read cycles).
		32KFASTREAD	0	Bank-switching on 32K bank boundaries only.
		EXTMEM	1	Consecutive bank switches on external memory reads. Each read cycle consists of 3 cycles: starting, read, and trailing.
14-13	DIVFCT			CLKOUT output divide factor. The CLKOUT output is driven by an on-chip source having a frequency equal to 1/(DIVFCT + 1) of the DSP clock.
		ZERO	00	CLKOUT is not divided.
		CLKBYTWO	01	CLKOUT is divided by 2 from the DSP clock.
		CLKBYTHREE	10	CLKOUT is divided by 3 from the DSP clock.
		CLKBYFOUR	11	CLKOUT is divided by 4 from the DSP clock.
12	IACK			$\overline{\text{IACK}}$ signal output off enable bit controls the $\overline{\text{IACK}}$ signal output off function.
		ON	0	$\overline{\text{IACK}}$ signal output off function is disabled.
		OFF	1	$\overline{\text{IACK}}$ signal output off function is enabled.
11-3	reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
2	HBH			**For C5416:** HPI data bus holder enable bit.
		DISABLE	0	The HPI data bus holder is disabled. When HPI16 pin is set to a logic high, HPI data bus holder is enabled.
		ENABLE	1	The HPI data bus holder is enabled. When not driven, the HPI data bus, HD(7-0), is held in the previous logic level.
1	BH			**For C5416:** Bus holder enable bit.
		DISABLE	0	The bus holder is disabled. When HPI16 pin is set to a logic high, address bus holder is enabled.
		ENABLE	1	The data bus holder is enabled. When not driven, the data bus, D(15-0), is held in the previous logic level. When HPI16 pin is set to a logic high, address bus holder is enabled.
0	reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.

Figure A–23. Bank-Switching Control Register (BSCR) — C5440 and C5441



Legend: R/W-x = Read/Write-Reset value

Table A–23. Bank-Switching Control Register (BSCR) Field Values — C5440 and C5441 (EBUS_BSCR_field_symval)

Bit	*field*	*symval*	Value	Description
15-3	reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
2	BHD			HPI data bus holder enable bit.
		DISABLE	0	The HPI data bus holder is disabled.
		ENABLE	1	The HPI data bus holder is enabled. When not driven, the HPI data bus, HD(15-0), is held in the previous logic level.
1	BHA			HPI address bus holder enable bit.
		DISABLE	0	The HPI address bus holder is disabled.
		ENABLE	1	The HPI address bus holder is enabled. When not driven, the HPI address bus, HA(15-0), is held in the previous logic level.
0	reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.

A.3 GPIO Registers (C5440 and C5441)

A.3.1 General Purpose I/O Register (GPIO)

Figure A–24. General Purpose I/O Register (GPIO)

15	14 12	11	10	9	8
TOUT†	reserved	DIR3	DIR2	DIR1	DIR0
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

7 4	3	2	1	0
reserved	DAT3	DAT2	DAT1	DAT0
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

† Only available on devices with a second on-chip timer.

Legend: R/W-x = Read/Write-Reset value

Table A–24. General Purpose I/O Register (GPIO) Field Values (GPIO_GPIO_field_symval)

Bit	*field*	*symval*	Value	Description
15	TOUT			Timer output enable
		ENABLE	1	Enables the timer output
		DISABLE	0	Disables the timer output
14-12	reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
11	DIR3			GPIO pin direction 3
		INPUT	0	GPIO pin 3 is used as input
		OUTPUT	1	GPIO pin 3 is used as output
10	DIR2			GPIO pin direction 2
		INPUT	0	GPIO pin 2 is used as input
		OUTPUT	1	GPIO pin 2 is used as output
9	DIR1			GPIO pin direction 1
		INPUT	0	GPIO pin 1 is used as input
		OUTPUT	1	GPIO pin 1 is used as output
8	DIR0			GPIO pin direction 0
		INPUT	0	GPIO pin 0 is used as input
		OUTPUT	1	GPIO pin 0 is used as output

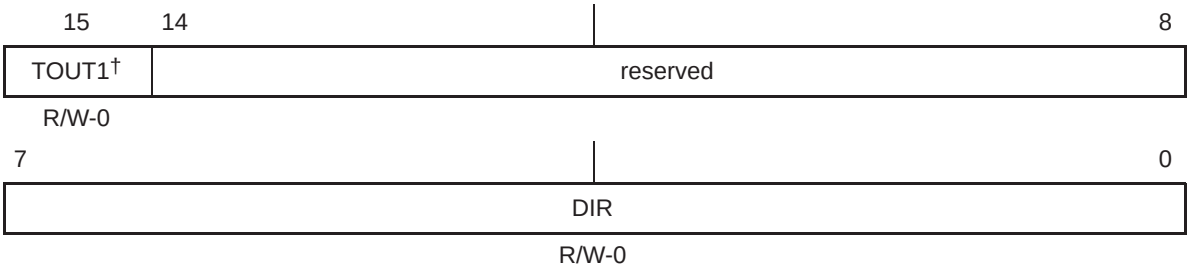
Table A–24. General Purpose I/O Register (GPIO) Field Values (GPIO_GPIO_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
7-4	reserved			Reserved. The reserved but location is always read as zero. A value written to this field has no effect.
3	DAT3			GPIO data bit 3
		LOW	0	GPIO0 is driven with a 0 (DIR3 = 1)
				GPIO0 is read as 0 (DIR3 = 0)
		HI	1	GPIO0 is driven with a 1 (DIR3 = 1)
				GPIO0 is read as 1 (DIR3 = 0)
2	DAT2			GPIO data bit 2
		LOW	0	GPIO0 is driven with a 0 (DIR2 = 1)
				GPIO0 is read as 0 (DIR2 = 0)
		HI	1	GPIO0 is driven with a 1 (DIR2 = 1)
				GPIO0 is read as 1 (DIR2 = 0)
1	DAT1			GPIO data bit 1
		LOW	0	GPIO0 is driven with a 0 (DIR1 = 1)
				GPIO0 is read as 0 (DIR1 = 0)
		HI	1	GPIO0 is driven with a 1 (DIR1 = 1)
				GPIO0 is read as 1 (DIR1 = 0)
0	DAT0			GPIO data bit 0
		LOW	0	GPIO0 is driven with a 0 (DIR0 = 1)
				GPIO0 is read as 0 (DIR0 = 0)
		HI	1	GPIO0 is driven with a 1 (DIR0 = 1)
				GPIO0 is read as 1 (DIR0 = 0)

A.4 HPI Registers

A.4.1 General Purpose I/O Control Register (GPIOCR)

Figure A–25. General Purpose I/O Control Register (GPIOCR)



† Only available on devices with a second on-chip timer.

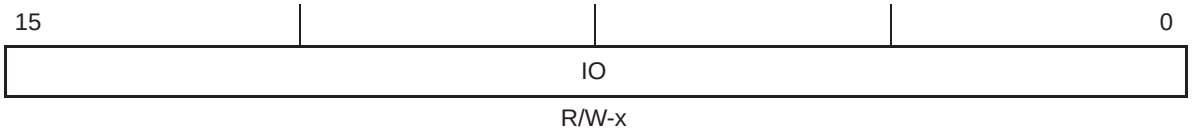
Legend: R/W-x = Read/Write-Reset value

Table A–25. General Purpose I/O Control Register (GPIOCR) Field Values (HPI_GPIOCR_field_symval)

Bit	*field*	*symval*	Value	Description
15	TOUT1			**For C5402:** Timer1 output enable bit enables or disables the timer1 output on the HINT pin. The timer1 output is only available when the HPI-8 is disabled. This bit is reserved on devices that have only one timer.
			0	The timer1 output is not available externally.
		MASK	1	The timer1 output is driven on the HINT pin.
14-8	reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
7-0	DIR			I/O pins direction field to configure the HD pins as inputs or outputs. Bit 7-Bit 0 corresponds to the direction for PIN7 to 0 respectively.
			0	The HD corresponding pin is configured as an input.
		MASK	1	The HD corresponding pin is configured as an output. When the HPI-8 is enabled, this bit is forced to 0 and is not affected by writes.

A.4.2 General Purpose I/O Status Register (GPIOSR)

Figure A–26. General Purpose Status Register (GPIOSR)



Legend: R/W-x = Read/Write-Reset value

Table A–26. General Purpose I/O Status Register (GPIOSR) Field Values (HPI_GPIOSR_field_symval)

Bit	*field*†	*symval*	Value	Description
15-0	IO			I/O pin status bit reflects the logic level on the HD pin. When the HD pin is configured as an input (DIR = 0 in GPIOCR), the IO bit latches the logic value (1 or 0) of the HD pin. Writes to the IO bit have no effect when the HD pin is configured as an input. When the HD pin is configured as an output (DIR = 1 in GPIOCR), the HD pin is driven to the logic level (1 or 0) written in the IO bit.
			0	The HD input is externally driven low, or the HD output is internally driven low.
		MASK	1	The HD input is externally driven high, or the HD output is internally driven high.

† The GPIOSR register can be treated as a single field register (IO).

A.4.3 HPI Control Register (HPIC) (for 5401, 5402, 5409, and 5410 only)

Figure A–27. HPI Control Register (HPIC) (for 5410)

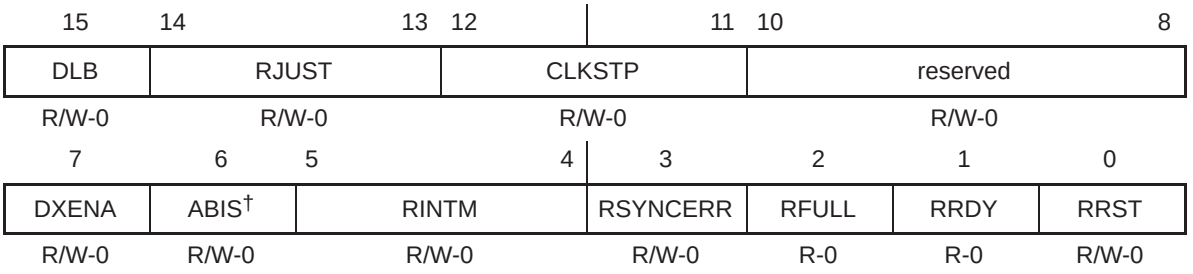
Note: X = Any value can be written

Refer to the specific device data sheet for an explanation fo the fields of this register.

A.5 Multichannel BSP (McBSP) Registers

A.5.1 McBSP Serial Port Control Register (SPCR1)

Figure A–28. McBSP Serial Port Control Register 1 (SPCR1)



† Only available on specific devices.

Legend: R/W-x = Read/Write-Reset value

Table A–27. McBSP Serial Port Control Register 1 (SPCR1) Field Values (MCBSP_SPCR1_field_symval)

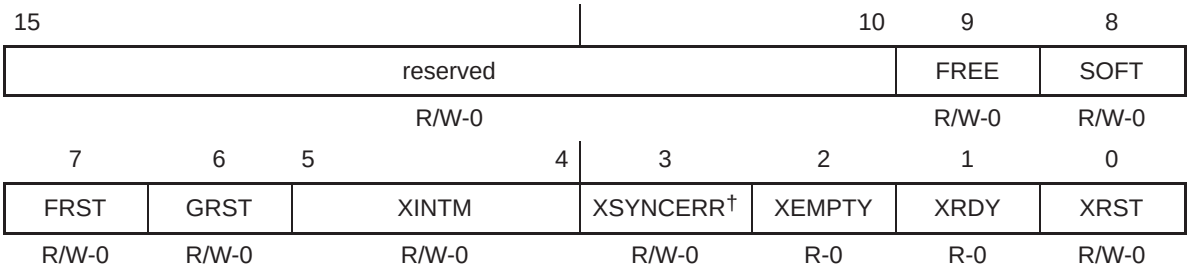
Bit	*field*	*symval*	Value	Description
15	DLB			Digital loop back mode enable bit.
		OFF	0	Digital loop back mode is disabled.
		ON	1	Digital loop back mode is enabled.
14-13	RJUST			Receive sign-extension and justification mode bit.
		RZF	00	Right-justify and zero-fill MSBs in DRR[1, 2].
		RSE	01	Right-justify and sign-extend MSBs in DRR[1, 2].
		LZF	10	Left-justify and zero-fill LSBs in DRR[1, 2].
			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.

Table A–27. McBSP Serial Port Control Register 1 (SPCR1) Field Values (MCBSP_SPCR1_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
7	DXENA			DX enabler bit.
		OFF	0	DX enabler is off.
		ON	1	DX enabler is on.
6	ABIS			**For C5410, C5410A, and C5416:** A-bis enable mode bit.
		DISABLE	0	A-bis mode is disabled.
		ENABLE	1	A-bis mode is enabled.
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[1, 2] is not in overrun condition.
		YES	1	DRR[1, 2] is not read, RBR[1, 2] is full, and RSR[1, 2] 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[1, 2].
0	RRST			Receiver reset bit resets or enables the receiver.
		DISABLE	0	The serial port receiver is disabled and in reset state.
		ENABLE	1	The serial port receiver is enabled.

A.5.2 McBSP Serial Port Control Register 2 (SPCR2)

Figure A–29. McBSP Serial Port Control Register 2 (SPCR2)



† Caution: Writing a 1 to this bit sets the error condition; thus, it is mainly used for testing purposes or if this operation is desired.

Legend: R/W-x = Read/Write-Reset value

Table A–28. McBSP Serial Port Control Register 2 (SPCR2) Field Values (MCBSP_SPCR2_field_symval)

Bit	*field*	*symval*	Value	Description
15-10	reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
9	FREE			Free-running enable mode bit.
		NO	0	Free-running mode is disabled.
		YES	1	Free-running mode is enabled.
8	SOFT			Soft bit enable mode bit.
		NO	0	Soft mode is disabled.
		YES	1	Soft mode is enabled.
7	FRST			Frame-sync generator reset.
		RESET	0	Frame-synchronization logic is reset. Frame-sync signal (FSG) is not generated by the sample-rate generator.
		FSG	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.
6	GRST			Sample-rate generator reset.
		RESET	0	Sample-rate generator is reset.
		CLKG	1	Sample-rate generator is taken out of reset. CLKG is driven as per programmed value in sample-rate generator registers (SRGR[1, 2]).

Table A–28. McBSP Serial Port Control Register 2 (SPCR2) Field Values (MCBSP_SPCR2_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
5-4	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.
3	XSYNCERR			Transmit synchronization error bit.
		NO	0	No synchronization error is detected.
		YES	1	Synchronization error is detected.
2	XEMPTY			Transmit shift register empty bit.
		YES	0	XSR[1, 2] is empty.
		NO	1	XSR[1, 2] is not empty.
1	XRDY			Transmitter ready bit.
		NO	0	Transmitter is not ready.
		YES	1	Transmitter is ready for new data in DXR[1, 2].
0	XRST			Transmitter reset bit resets or enables the transmitter.
		DISABLE	0	Serial port transmitter is disabled and in reset state.
		ENABLE	1	Serial port transmitter is enabled.

A.5.3 McBSP Pin Control Register (PCR)

Figure A–30. McBSP Pin Control Register (PCR)

15	14	13	12	11	10	9	8
reserved		XIOEN	RIOEN	FSXM	FSRM	CLKXM	CLKRM
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
SCLKME†	CLKSSTAT	DXSTAT	DRSTAT	FSXP	FSRP	CLKXP	CLKRP
R/W-0	R-0	R-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0

† Only available on specific devices with 128-channel selection capability.

Legend: R/W-x = Read/Write-Reset value

Table A–29. McBSP Pin Control Register (PCR) Field Values (MCBSP_PCR_field_symval)

Bit	*field*	*symval*	Value	Description
15-14	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 SPCR2).
		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 SPCR1).
		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.
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 SRGR2.

Table A–29. McBSP Pin Control Register (PCR) Field Values (MCBSP_PCR_field_symval) (Continued)

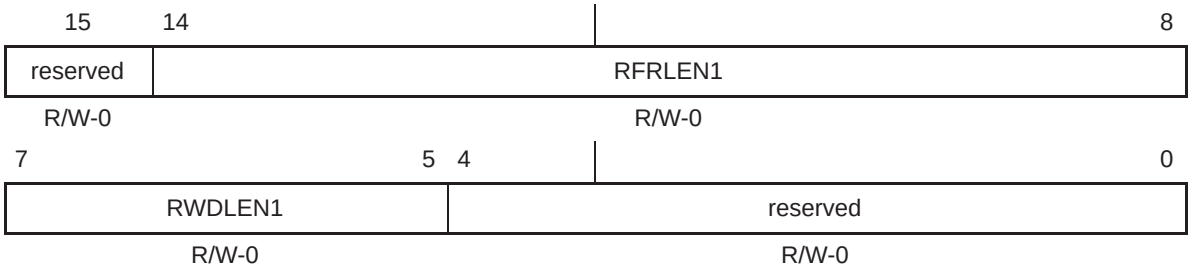
Bit	*field*	*symval*	Value	Description
10	FSRM			Receive frame-synchronization mode bit.
		EXTERNAL	0	Frame-synchronization signal is derived from an external source. FSR is an input pin.
		INTERNAL	1	Frame-synchronization signal is generated internally by the sample-rate generator. FSR is an output pin, except when GSYNC = 1 in SRGR2.
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 SPCR1 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 SPCR1):
		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 SPCR1):
		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.
7	SCLKME			**For devices with 128-channel selection capability:** Sample-rate clock mode extended enable bit.
		NO	0	BCLKR and BCLKX are not used by the sample-rate generator for external synchronization.
		BCLK	1	BCLKR and BCLKX are used by the sample-rate generator for external synchronization.

Table A–29. McBSP Pin Control Register (PCR) Field Values (MCBSP_PCR_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
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.

A.5.4 Receive Control Register 1 (RCR1)

Figure A–31. Receive Control Register 1 (RCR1)



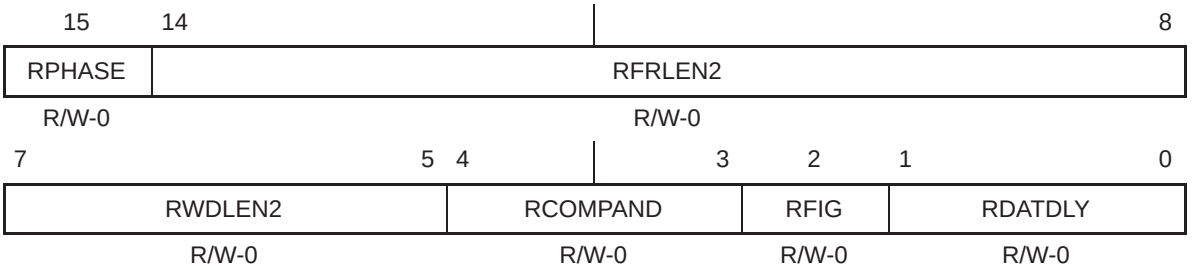
Legend: R/W-x = Read/Write-Reset value

Table A–30. Receive Control Register 1 (RCR1) Field Values (MCBSP_RCR1_field_symval)

Bit	*field*	*symval*	Value	Description
15	reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
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
4-0	reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.

A.5.5 Receive Control Register 2 (RCR2)

Figure A–32. Receive Control Register 2 (RCR2)



Legend: R/W-x = Read/Write-Reset value

Table A–31. Receive Control Register 2 (RCR2) Field Values (MCBSP_RCR2_field_symval)

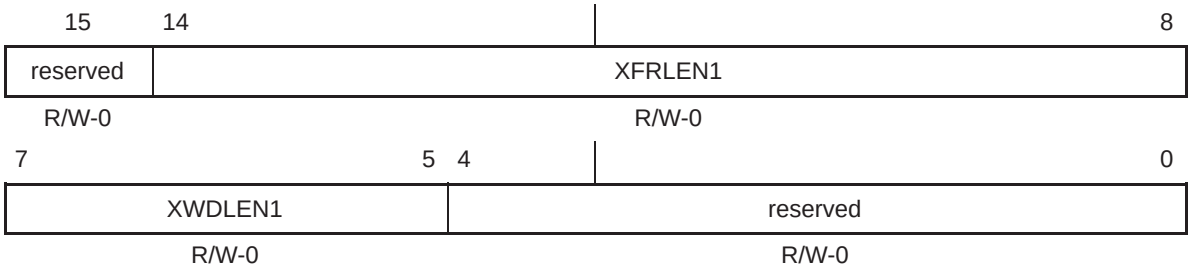
Bit	*field*	*symval*	Value	Description
15	RPHASE			Receive phases bit.
		SINGLE	0	Single-phase frame
		DUAL	1	Dual-phase frame
14-8	RFRLEN2	OF(*value*)	0-127	Specifies the number of words (length) in the receive frame.
7-5	RWDLEN2			Specifies the number of bits (length) in the receive word.
		8BIT	000	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
4-3	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.

Table A–31. Receive Control Register 2 (RCR2) Field Values (MCBSP_RCR2_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
2	RFIG			Receive frame ignore bit.
		YES	0	Receive frame-synchronization pulses after the first pulse restarts the transfer.
		NO	1	Receive frame-synchronization pulses after the first pulse are ignored.
1-0	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

A.5.6 Transmit Control Register 1 (XCR1)

Figure A–33. Transmit Control Register 1 (XCR1)



Legend: R/W-x = Read/Write-Reset value

Table A–32. Transmit Control Register 1 (XCR1) Field Values (MCBSP_XCR1_field_symval)

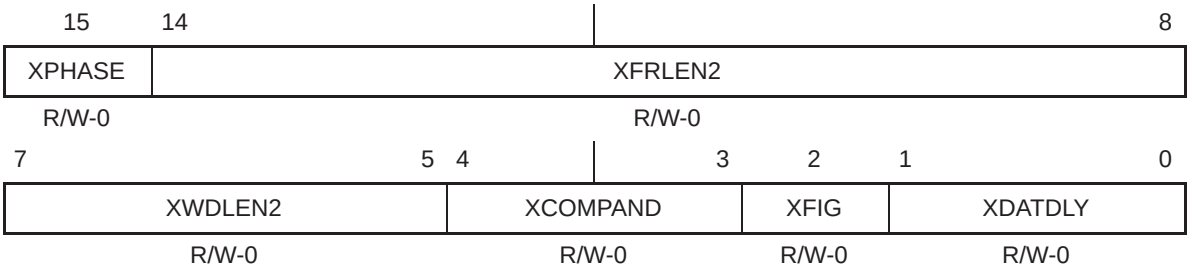
Bit	*field*	*symval*	Value	Description
15	reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
14-8	XFRLEN1	OF(*value*)	0-127	Specifies the number of words (length) in the transmit frame.
7-5	XWDLEN1			Specifies the number of bits (length) 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.

Table A–32. Transmit Control Register 1 (XCR1) Field Values (MCBSP_XCR1_field_symval)

Bit	*field*	*symval*	Value	Description
		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
4-0	reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.

A.5.7 Transmit Control Register 2 (XCR2)

Figure A–34. Transmit Control Register 2 (XCR2)



Legend: R/W-x = Read/Write-Reset value

Table A–33. Transmit Control Register 2 (XCR2) Field Values (MCBSP_XCR2_field_symval)

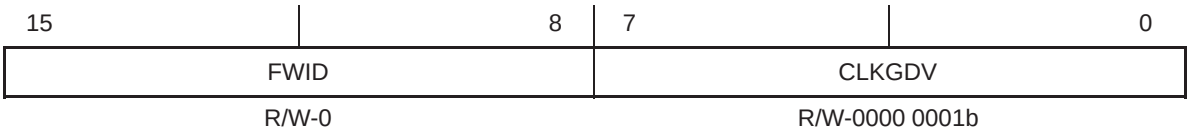
Bit	*field*	*symval*	Value	Description
15	XPHASE			Transmit phases bit.
		SINGLE	0	Single-phase frame
		DUAL	1	Dual-phase frame
14-8	XFRLEN2	OF(*value*)	0-127	Specifies the number of words (length) in the transmit frame.
7-5	XWDLEN2			Specifies the number of bits (length) in the transmit word.
		8BIT	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.

Table A–33. Transmit Control Register 2 (XCR2) Field Values (MCBSP_XCR2_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
		32BIT	101	Transmit word length is 32 bits.
			110	Reserved
			111	Reserved
4-3	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.
2	XFIG			Transmit frame ignore bit.
		YES	0	Transmit frame-synchronization pulses after the first pulse restarts the transfer.
		NO	1	Transmit frame-synchronization pulses after the first pulse are ignored.
1-0	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

A.5.8 Sample Rate Generator Register 1 (SRGR1)

Figure A–35. Sample Rate Generator Register 1 (SRGR1)



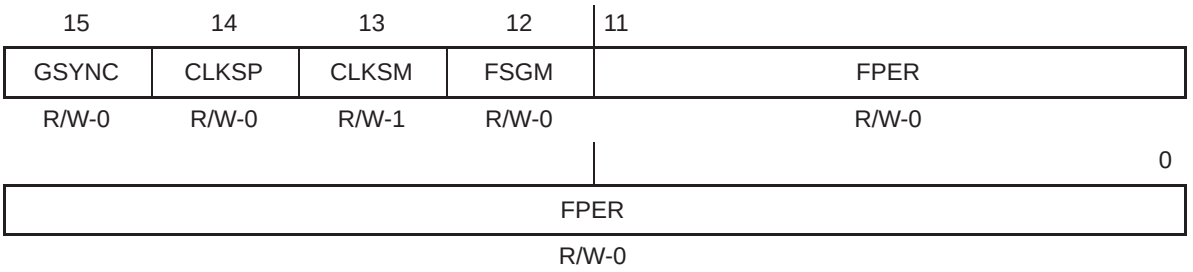
Legend: R/W-x = Read/Write-Reset value

Table A–34. Sample Rate Generator Register 1 (SRGR1) Field Values (MCBSP_SRGR1_field_symval)

Bit	*field*	*symval*	Value	Description
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.

A.5.9 Sample Rate Generator Register 2 (SRGR2)

Figure A–36. Sample Rate Generator Register 2 (SRGR2)



Legend: R/W-x = Read/Write-Reset value

Table A–35. Sample Rate Generator Register 2 (SRGR2) Field Values (MCBSP_SRGR2_field_symval)

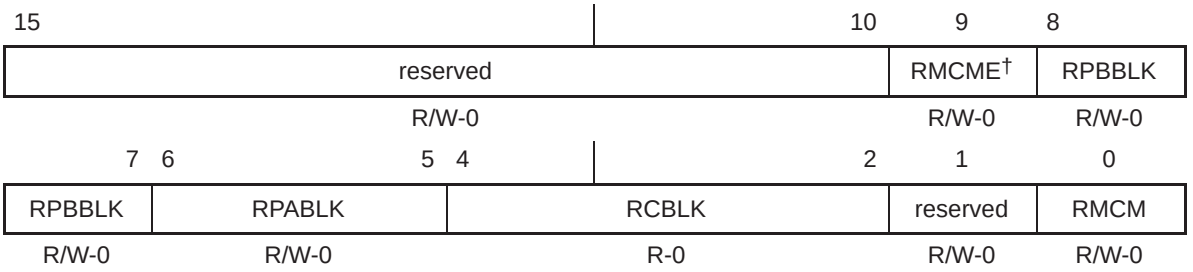
Bit	*field*	*symval*	Value	Description
15	GSYNC			Sample-rate generator clock synchronization bit only used when the external clock (CLKS) drives the sample-rate generator clock (CLKSM = 0).
		FREE	0	The sample-rate generator clock (CLKG) is free running.
		SYNC	1	The sample-rate generator clock (CLKG) 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.

Table A–35. Sample Rate Generator Register 2 (SRGR2) Field Values (MCBSP_SRGR2_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
14	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.
13	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.
12	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[1, 2]-to-XSR[1, 2] copy. When FSGM = 0, FWID bit in SRGR1 and FPER bit are ignored.
		FSG	1	Transmit frame-sync signal (FSX) driven by the sample-rate generator frame-sync signal (FSG).
11-0	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.

A.5.10 Multichannel Control Register 1 (MCR1)

Figure A–37. Multichannel Control Register 1 (MCR1)



† Only available on specific devices that provide 128-channel selection capability.

Legend: R/W-x = Read/Write-Reset value

Table A–36. Multichannel Control Register 1 (MCR1) Field Values (MCBSP_MCR1_field_symval)

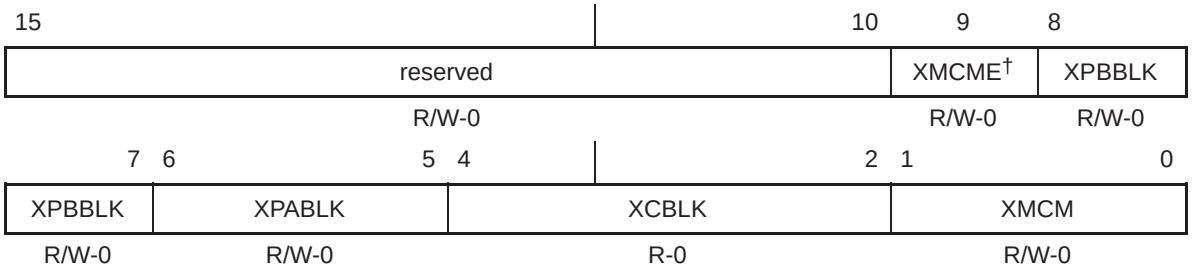
Bit	*field*	*symval*	Value	Description
15-10	reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
9	RMCME			**For devices with 128-channel selection capability:** Receive 128-channel selection enable bit.
		NO	0	Normal 32-channel selection is enabled.
		ATOH	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
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

Table A–36. Multichannel Control Register 1 (MCR1) Field Values (MCBSP_MCR1_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
		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
	RCBLK	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.

A.5.11 Multichannel Control Register 2 (MCR2)

Figure A–38. Multichannel Control Register 2 (MCR2)



† Only available on specific devices that provide 128-channel selection capability.

Legend: R/W-x = Read/Write-Reset value

Table A–37. Multichannel Control Register 2 (MCR2) Field Values (MCBSP_MCR2_field_symval)

Bit	*field*	*symval*	Value	Description
15-10	reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
9	XMCME			**For devices with 128-channel selection capability:** Transmit 128-channel selection enable bit.
		NO	0	Normal 32-channel selection is enabled.
		ATOH	1	Six additional registers (XCERC-XCERH) are used to enable 128-channel selection.
8-7	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
6-5	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
4-2	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

Table A–37. Multichannel Control Register 2 (MCR2) Field Values (MCBSP_MCR2_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
1-0	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.

† 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.

A.5.12 Receive Channel Enable Register (RCERn)

Figure A–39. Receive Channel Enable Register (RCERn)

15	14	13	12	11	10	9	8
RCE15	RCE14	RCE13	RCE12	RCE11	RCE10	RCE9	RCE8
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
RCE7	RCE6	RCE5	RCE4	RCE3	RCE2	RCE1	RCE0
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-x = Read/Write-Reset value

Table A–38. Receive Channel Enable Register (RCERn) Field Values (MCBSP_RCERn_field_symval)

Bit	*field*†	*symval*	Value	Description
				For devices with only 32-channel selection capability:
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 MCR1.

† The register is also treated as having a single field (RCEn).

Table A–38. Receive Channel Enable Register (RCERn) Field Values (MCBSP_RCERn_field_symval) (Continued)

Bit	*field*†	*symval*	Value	Description
15-0	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 MCR1.
				For devices with 128-channel selection capability:
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 channels 0-15 within the 16-channel-wide block.
15-0	RCEB	OF(*value*)	0-FFFFh	A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) reception of channels 16-31 within the 16-channel-wide block.
15-0	RCEC	OF(*value*)	0-FFFFh	A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) reception of channels 32-47 within the 16-channel-wide block.
15-0	RCED	OF(*value*)	0-FFFFh	A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) reception of channels 48-63 within the 16-channel-wide block.
15-0	RCEE	OF(*value*)	0-FFFFh	A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) reception of channels 64-79 within the 16-channel-wide block.
15-0	RCEF	OF(*value*)	0-FFFFh	A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) reception of channels 80-95 within the 16-channel-wide block.
15-0	RCEG	OF(*value*)	0-FFFFh	A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) reception of channels 96-111 within the 16-channel-wide block.
15-0	RCEH	OF(*value*)	0-FFFFh	A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) reception of channels 112-127 within the 16-channel-wide block.

† The register is also treated as having a single field (RCEn).

A.5.13 Transmit Channel Enable Register (XCERn)

Figure A–40. Transmit Channel Enable Register (XCERn)

15	14	13	12	11	10	9	8
XCE15	XCE14	XCE13	XCE12	XCE11	XCE10	XCE9	XCE8
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
XCE7	XCE6	XCE5	XCE4	XCE3	XCE2	XCE1	XCE0
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-x = Read/Write-Reset value

Table A–39. Transmit Channel Enable Register (XCERn) Field Values (MCBSP_XCERn_field_symval)

Bit	*field*†	*symval*	Value	Description
				For devices with only 32-channel selection capability:
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 MCR2.
15-0	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 MCR2.
				For devices with 128-channel selection capability:
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 channels 0-15 within the 16-channel-wide block.
15-0	XCEB	OF(*value*)	0-FFFFh	A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) transmission of channels 16-31 within the 16-channel-wide block.
15-0	XCEC	OF(*value*)	0-FFFFh	A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) transmission of channels 32-47 within the 16-channel-wide block.
15-0	XCED	OF(*value*)	0-FFFFh	A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) transmission of channels 48-63 within the 16-channel-wide block.
15-0	XCEE	OF(*value*)	0-FFFFh	A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) transmission of channels 64-79 within the 16-channel-wide block.

† The registers are also treated as having a single field (XCEn).

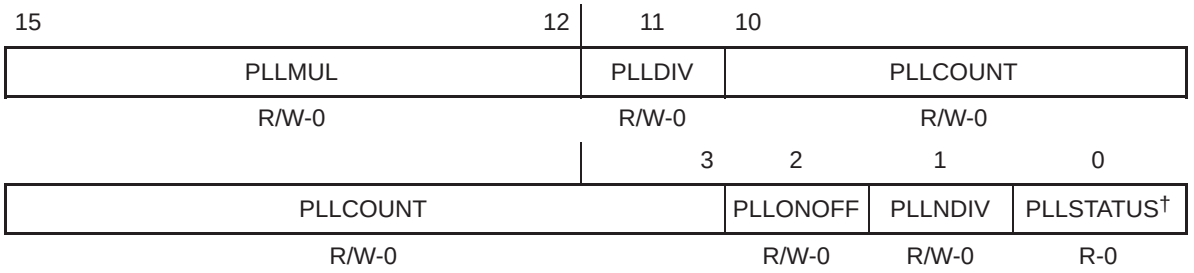
Table A–39. Transmit Channel Enable Register (XCERn) Field Values (MCBSP_XCERn_field_symval) (Continued)

Bit	*field*†	*symval*	Value	Description
15-0	XCEF	OF(*value*)	0-FFFFh	A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) transmission of channels 80-95 within the 16-channel-wide block.
15-0	XCEG	OF(*value*)	0-FFFFh	A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) transmission of channels 96-111 within the 16-channel-wide block.
15-0	XCEH	OF(*value*)	0-FFFFh	A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) transmission of channels 112-127 within the 16-channel-wide block.

† The registers are also treated as having a single field (XCEn).

A.6 PLL Registers (CLKMD)

Figure A–41. Clock Mode Register (CLKMD)



† When in DIV mode (PLLSTATUS is low), PLLMUL, PLLDIV, PLLCOUNT, and PLLONOFF are don't cares, and their contents are indeterminate.

Legend: R/W-x = Read/Write-Reset value

Table A–40. Clock Mode Register (CLKMD) Field Values (PLL_CLKMD_field_symval)

Bit	*field*	*symval*	Value	Description
15-12	PLLMUL	OF(*value*)	0-15	This PLL multiplier value defines the frequency multiplier in conjunction with the PLLDIV and PLLNDIV bits.
11	PLLDIV			PLL divider. Defines the frequency multiplier in conjunction with the PLLMUL and PLLNDIV bits.
		OFF	0	
		ON	1	
10-3	PLLCOUNT	OF(*value*)	0-255	This PLL counter value specifies the number of input clock cycles (in increments of 16 cycles) for the PLL lock timer to count before the PLL begins clocking the processor after the PLL is started. The PLL counter is a down-counter, which is driven by the input clock divided by 16; therefore, for every 16 input clocks, the PLL counter decrements by 1. The PLL counter can be used to ensure that the processor is not clocked until the PLL is locked, so that only valid clock signals are sent to the device.
2	PLLONOFF			PLL on/off mode bit. Enables or disables the PLL part of the clock generator in conjunction with the PLLNDIV bit.
		OFF	0	PLL is off unless PLLNDIV = 1.
		ON	1	PLL is on regardless of the PLLNDIV bit status.

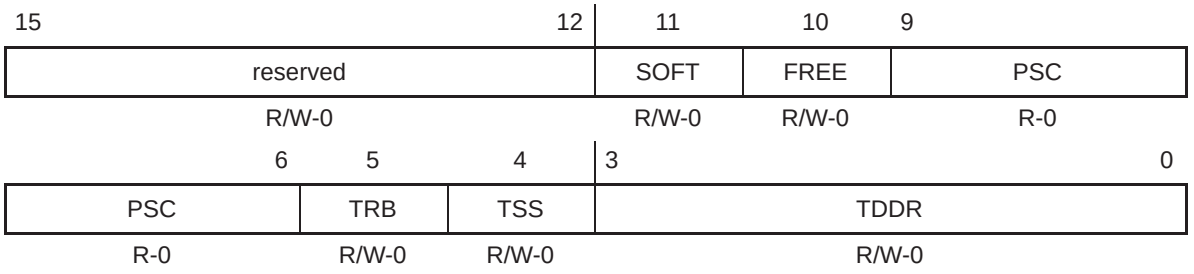
Table A–40. Clock Mode Register (CLKMD) Field Values (PLL_CLKMD_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
1	PLLNDIV			PLL clock generator mode select bit. Determines whether the clock generator works in PLL mode or in divider (DIV) mode, thus defining the frequency multiplier in conjunction with the PLLMUL and PLLDIV bits.
		OFF	0	DIV mode is used.
		ON	1	PLL mode is used.
0	PLLSTATUS			This read-only bit indicates the mode that the clock generator is operating.
			0	Divider (DIV) mode
			1	PLL mode

A.7 Timer Registers

A.7.1 Timer Control Register (TCR)

Figure A–42. Timer Control Register (TCR)



Legend: R/W-x = Read/Write-Reset value

Table A–41. Timer Control Register (TCR) Field Values (TIMER_TCR_field_symval)

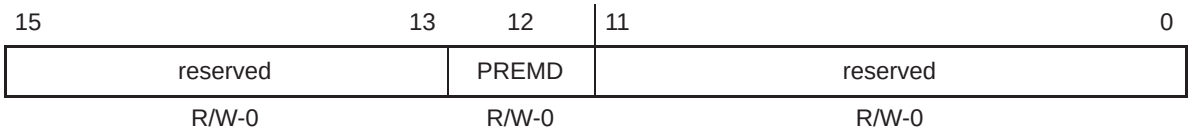
Bit	*field*	*symval*	Value	Description
15-12	reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
11	SOFT			Used in conjunction with FREE bit to determine the state of the timer when a breakpoint is encountered in the HLL debugger. When FREE bit is cleared, SOFT bit selects the timer mode.
		BRKPTNOW	0	The timer stops immediately.
		WAITZERO	1	The timer stops when the counter decrements to 0.
10	FREE			Used in conjunction with SOFT bit to determine the state of the timer when a breakpoint is encountered in the HLL debugger. When FREE bit is cleared, SOFT bit selects the timer mode.
		WITHSOFT	0	SOFT bit selects the timer mode.
		NOSOFT	1	The timer runs free regardless of SOFT bit status.
9-6	PSC			Timer prescalar counter. This read-only bit specifies the count for the on-chip timer when in direct mode (PREMD bit is cleared in the TSCR). When PSC bit is decremented past 0 or the timer is reset, PSC bit is loaded with the contents of TDDR bit and the TIM is decremented.
5	TRB			Timer reload bit. TRB bit is always read as a 0.
		NORESET	0	The on-chip timer is not reset.
		RESET	1	The on-chip timer is reset. When TRB bit is set, the TIM is loaded with the value in the PRD and PSC bit is loaded with the value in TDDR bit when in direct mode (PREMD bit is cleared in the TSCR).

Table A–41. Timer Control Register (TCR) Field Values (TIMER_TCR_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
4	TSS			Timer stop status bit. Stops or starts the on-chip timer. At reset, TSS bit is cleared and the timer immediately starts timing.
		START	0	The timer is started.
		STOP	1	The timer is stopped.
3-0	TDDR			The timer prescalar for the on-chip timer.
				In prescalar direct mode (PREMD = 0 in TSCR):
		OF(*value*)	0-15	This value specifies the prescalar count for the on-chip timer. When PSC bit is decremented past 0, PSC bit is loaded with this TDDR content.
				In prescalar indirect mode (PREMD = 1 in TSCR):
		OF(*value*)		This value relates to an indirect prescalar count, up to 65535, for the on-chip timer. When PSC bit is decremented past 0, PSC bit is loaded with this prescalar value.
			0000	Prescalar value: 0001h
			0001	Prescalar value: 0003h
			0010	Prescalar value: 0007h
			0011	Prescalar value: 000Fh
			0100	Prescalar value: 001Fh
			0101	Prescalar value: 003Fh
			0110	Prescalar value: 007Fh
			0111	Prescalar value: 00FFh
			1000	Prescalar value: 01FFh
			1001	Prescalar value: 03FFh
			1010	Prescalar value: 07FFh
			1011	Prescalar value: 0FFFh
			1100	Prescalar value: 1FFFh
			1101	Prescalar value: 3FFFh
			1110	Prescalar value: 7FFFh
			1111	Prescalar value: FFFFh

A.7.2 Timer Secondary Control Register (TSCR)

Figure A–43. Timer Secondary Control Register (TSCR) — C5440, C5441, and C5471



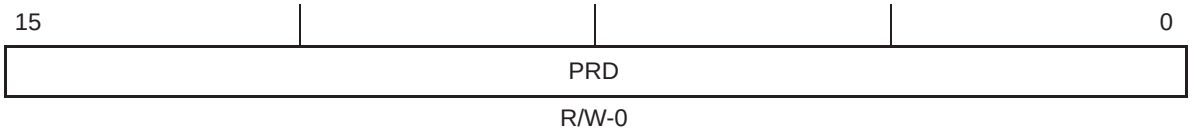
Legend: R/W-x = Read/Write-Reset value

Table A–42. Timer Secondary Control Register (TSCR) Field Values (TIMER_TSCR_field_symval)

Bit	*field*	*symval*	Value	Description
15-13	reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
12	PREMD			Prescalar mode select bit.
		DIRECT	0	Direct mode. When PSC bit in TCR is decremented past 0, PSC bit is loaded with TDDR content in TCR.
		INDIRECT	1	Indirect mode. When PSC bit in TCR is decremented past 0, PSC bit is loaded with the prescalar value associated with TDDR bit in TCR.
11-0	reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.

A.7.3 Timer Period Register (PRD)

Figure A–44. Timer Period Register (PRD)



Legend: R/W-x = Read/Write-Reset value

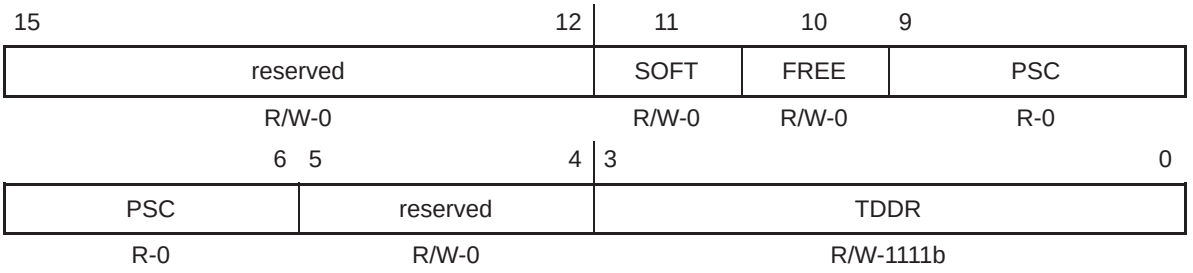
Table A–43. Timer Period Register (PRD)

Bit	*field*	*symval*	Value	Description
15-0	PRD	0F(value)	0-FFFFh	Used to store a 16-bit value that is used to load the timer register (TIM). Set to FFFFh at reset.

A.8 Watchdog Timer Registers (C5441)

A.8.1 Watchdog Timer Control Register (WDTCR)

Figure A–45. Watchdog Timer Control Register (WDTCR)



Legend: R/W-x = Read/Write-Reset value

Table A–44. Watchdog Timer Control Register (WDTCR) Field Values (WDTIM_WDTCR_field_symval)

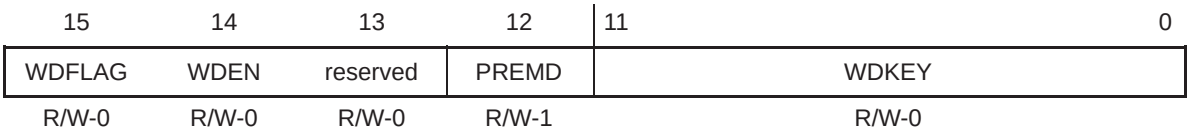
Bit	*field*	*symval*	Value	Description
15-12	reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
11	SOFT			Used in conjunction with FREE bit to determine the state of the watchdog timer when a breakpoint is encountered in the HLL debugger. When FREE bit is cleared, SOFT bit selects the watchdog timer mode.
		BRKPTNOW	0	The watchdog timer stops immediately.
		WAITZERO	1	The watchdog timer stops when the counter decrements to 0.
10	FREE			Used in conjunction with SOFT bit to determine the state of the watchdog timer when a breakpoint is encountered in the HLL debugger. When FREE bit is cleared, SOFT bit selects the watchdog timer mode.
		WITHSOFT	0	SOFT bit selects the watchdog timer mode.
		NOSOFT	1	The watchdog timer runs free regardless of SOFT bit status.
9-6	PSC			Timer prescalar counter. This read-only bit specifies the count for the on-chip watchdog timer when in direct mode (PREMD bit is cleared in the WDTSCR). When PSC bit is decremented past 0 or the watchdog timer is reset, PSC bit is loaded with the contents of TDDR bit and the WDTIM is decremented.
5-4	reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.

Table A–44. Watchdog Timer Control Register (WDTCR) Field Values (WDTIM_WDTCR_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
3-0	TDDR			The timer prescalar for the on-chip watchdog timer.
				In prescalar direct mode (PREMD = 0 in WDTSCR):
		OF(*value*)	0-15	This value specifies the prescalar count for the on-chip watchdog timer. When PSC bit is decremented past 0, PSC bit is loaded with this TDDR content.
	TDDR			**In prescalar indirect mode (PREMD = 1 in WDTSCR):**
		OF(*value*)		This value relates to an indirect prescalar count, up to 65535, for the on-chip watchdog timer. When PSC bit is decremented past 0, PSC bit is loaded with this prescalar value.
			0000	Prescalar value: 0001h
			0001	Prescalar value: 0003h
			0010	Prescalar value: 0007h
			0011	Prescalar value: 000Fh
			0100	Prescalar value: 001Fh
			0101	Prescalar value: 003Fh
			0110	Prescalar value: 007Fh
			0111	Prescalar value: 00FFh
			1000	Prescalar value: 01FFh
			1001	Prescalar value: 03FFh
			1010	Prescalar value: 07FFh
			1011	Prescalar value: 0FFFh
			1100	Prescalar value: 1FFFh
			1101	Prescalar value: 3FFFh
			1110	Prescalar value: 7FFFh
			1111	Prescalar value: FFFFh

A.8.2 Watchdog Timer Secondary Control Register (WDTSCR)

Figure A–46. Watchdog Timer Secondary Control Register (WDTSCR)



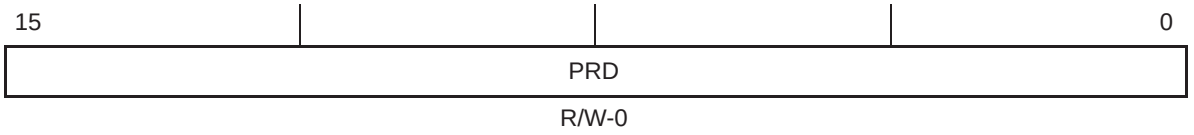
Legend: R/W-x = Read/Write-Reset value

Table A–45. Watchdog Timer Secondary Control Register (WDTSCR) Field Values (WDTIM_WDTSCR_field_symval)

Bit	*field*	*symval*	Value	Description
15	WDFLAG			Watchdog timer flag bit. This bit can be cleared by enabling the watchdog timer, by a device reset, or by being written with a 1.
		TIMEOUT	0	No watchdog timer time-out event occurred.
		NOTIMEOUT	1	Watchdog timer time-out event occurred.
14	WDEN			Watchdog timer enable bit.
		DISABLE	0	Watchdog timer is disabled. Watchdog timer output pin is disconnected from the watchdog timer time-out event and the counter starts to run.
		ENABLE	1	Watchdog timer is enabled. Watchdog timer output pin is connected to the watchdog timer time-out event. Watchdog timer can be disabled by a watchdog timer time-out event or by a device reset.
13	reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
12	PREMD			Prescalar mode select bit.
		DIRECT	0	Direct mode. When PSC bit in WDTCR is decremented past 0, PSC bit is loaded with TDDR content in WDTCR.
		INDIRECT	1	Indirect mode. When PSC bit in WDTCR is decremented past 0, PSC bit is loaded with the prescalar value associated with TDDR bit in WDTCR.
11-0	WDKEY			Watchdog timer reset key. A 12-bit value that before a watchdog timer times out, only a write sequence of a 5C6h followed by an A7Eh services the watchdog timer. Any other writes triggers a watchdog timer time-out event immediately.
		PREACTIVE	5C6h	
		ACTIVE	A7Eh	

A.8.3 Watchdog Timer Period Register (WDPRD)

Figure A–47. Watchdog Timer Period Register (WDPRD)



Legend: R/W-x = Read/Write-Reset value

Table A–46. Watchdog Timer Period Register (WDPRD)

Bit	*field*	*symval*	Value	Description
15-0	PRD	0F(value)	0-FFFFh	Stores a value that is used to reload the Watchdog timer counter register. (WDTIM)

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X