TMS320C54x Chip Support Library API Reference Guide

SPRU420
April 2001

Texas Instruments

IMPORTANT NOTICE

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

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

Customers are responsible for their applications using TI components.

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

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

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. Representation or reproduction of this information with alteration voids all warranties provided for an associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Resale of TI's products or services with *statements different from or beyond the parameters* stated by TI for that products or service voids all express and any implied warranties for the associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Also see: Standard Terms and Conditions of Sale for Semiconductor Products.
www.ti.com/sc/docs/stdterms.htm

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. It is intended to make it easier to get algorithms running in a real system. The goal is peripheral ease of use, shortened development time, portability, and hardware abstraction, with some level of standardization and compatibility among devices. A version of the CSL is available for all TMS320C54x™ DSP devices.

This document provides reference information for the CSL library and is organized as follows:

- Overview – high level overview of the CSL
- How to use CSL – Configuration and use of the DSP/BIOS™ Configuration Tool, installation, coding, compiling, linking, macros, etc.
- Using the DSP/BIOS™ Configuration Tool with the different CSL Modules
- Using CSL functions and macros with each individual CSL module.
- Using the individual registers.

How to Use This Manual

The information in this document describes the contents of the TMS320C5000™ DSP Chip Support Library (CSL) as follows:

- Chapter 1 provides an overview of the CSL, includes tables showing CSL API module support for various C5000 devices, and lists the API modules.
- Chapter 2 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 in the Code Composer Studio™ environment.

- Chapter 3 provides basic examples of how to configure the individual CSL modules using the DSP/BIOS™ Configuration Tool.
- Chapters 4-15 provide basic examples, functions, and macros 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, briefly describes its architecture, and provides a generic overview of the collection of functions, macros, and constants that help you program DSP peripherals.

Topic **Page**

1.1 Introduction to CSL

The Chip Support Library(CSL) is a fully scalable component of DSP/BIOS that provides C program functions to configure and control on-chip peripherals. It is intended to simplify the process of running algorithms in a real system. The goal is peripheral ease of use, shortened development time, portability, hardware abstraction, and a small level of standardization and compatibility among devices.

How the CSL Benefits You

- **Standard Protocol to Program Peripherals**

  A standard protocol to each programming of on-chip peripherals. This includes data types and macros to define peripheral configurations, and functions to implement the various operations of each peripheral.

- **Basic Resource Management**

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

- **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.

CSL integrates GUI, graphic user interface, into the DSP/BIOS configuration tool. The CSL tree of the configuration tool allows the pre-initialization of some peripherals by generating C files using CSL APIs. The peripherals are pre-configured with the pre-defined configuration objects (see Chapter 2, *How To Use CSL)*.

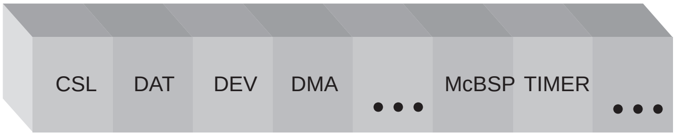
Chapter 3, *DSP/BIOS Configuration Tool: CSL Modules*, details the available CSL modules found in the DSP/BIOS Configuration tool.

CSL Architecture

The CSL consists of discrete 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 API modules. This architecture allows for future expansion as new modules are added and new peripherals emerge.

Figure 1–1. API Modules



Although each API module provides a unique API, 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.

Device support for an API module depends on whether or not the device actually uses the associated peripheral. For example, the Watchdog Timer WDTIM is not supported on a C5402 because this device does not support a Watchdog Timer. Other modules such as the IRQ, however, are supported on all devices.

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 and the module support symbol. these components must be included in your application.

Table 1–1. CSL Modules and Include Files

Peripheral Module (PER)	Description	Include File	Module Support Symbol
DAT	Device independent data copy/fill module	csl_dat.h	_DAT_SUPPORT
CHIP	Device specific module	csl_chip.h	_CHIP_SUPPORT
DMA	Direct memory access	csl_dma.h	_DMA_SUPPORT
EBUS	External memory bus interface	csl_ebus.h	_EBUS_SUPPORT
GPIO	General purpose I/O	csl_gpio.h	_GPIO_SUPPORT
HPI	Host port interface	csl_hpi.h	_HPI_SUPPORT
IRQ	Interrupt controller	csl_irq.h	_IRQ_SUPPORT
MCBSP	Multi-channel buffered serial port	csl_mcbsp.h	_MCBSP_SUPPORT
PLL	PLL	csl_pll.h	_PLL_SUPPORT
PWR	Power-down	csl_pwr.h	_PWR_SUPPORT
TIMER	Timer peripheral	csl_timer.h	_TIMER_SUPPORT
WDTIM	Watch Dog Timer Peripheral	csl_wdtim.h	_WDTIM_SUPPORT

Table 1–2 lists the C5000 devices that CSL supports and the far and near libraries included in CSL. The device support symbol to be used with the compiler.

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
C5402	csl5402.lib	csl5402x.lib	CHIP_5402
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_5440
C5441	csl5441.lib	csl5441x.lib	CHIP_5441
C5472	csl5472.lib	csl5472x.lib	CHIP_5472

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 is the Peripheral module name listed in Table 1–1) in capital letters.
- Function names use all small letters. Capital letters are used only if the function name consists of two separate words. (for example, PER_getConfig()).
- Macro names use all capital letters (for example, DMA_DMPREC_RMK).
- Data types start with a capital letter followed by small letters (for example, DMA_Handle).

1.3 Data Types

The CSL provides its own set of data types that all begin with a capital letter. Table 1–4 lists the CSL data types as defined in the *stdinc*.h file.

Table 1–4. CSL Data Types

Data Type	Description
Bool	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

1.3.1 Resource Management

CSL provides a limited set of functions to enable resource management for applications that support multiple algorithms and may reuse the same peripheral device.

Resource management in CSL is achieved through API calls to the PER_open and PER_close functions. The PER_open function normally takes a device number and reset flag as the primary arguments and returns a pointer to a Handle structure that contains information about which channel (DMA) or port (MCBSP) was opened. When given a specific device number, the open function checks a global flag to determine its availability. If the device/channel is available, then it returns a pointer to a predefined Handle structure for this device. If the device has already been opened by another process, then an invalid Handle is returned with a value equal to the CSL symbolic constant, INV.

Note: To ensure that no resource usage conflicts occur, CSL performs other function calls, other than returning an invalid handle from the PER_open function. You must check the value returned from the PER_open function to guarantee that the resource has been allocated.

Before accepting a handle object as an argument, API functions first check to ensure that a valid Handle has been passed.

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

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

1.3.1.1 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 */
```

Note: Handles are required only for peripherals that have multiple channels or ports, such as DMA, MCBSP, TIMER, and DAT.

1.4 Symbolic Constant Values

To facilitate initialization of values in your application code, the CSL provides symbolic constants for registers and writable field values as described in Table 1–5. The following naming conventions are used:

- *PER* indicates a peripheral module as listed in Table 1–1 on page 1-4.
- *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 1–5. 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 specified 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.

1.5 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 on page 1-4.
- *REG#* indicates, if applicable, a register with the channel number. (For example: 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 1.4; 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. These 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: ❏ Defined only for registers with more than one field. ❏ 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, gets the memory address (or sub-address) of the peripheral register REG#.

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

Macro	Description
*PER*_**RGET_H**(handle, *REG*)	Returns the value of the peripheral register REG associated with Handle.
*PER*_**RSET_H**(handle, *REG,* *regval*)	Writes the value to the peripheral register REG associated with Handle.
*PER*_**ADDR_H**(handle, *REG*)	If applicable, gets the memory address (or sub-address) of the peripheral register REG associated with Handle.
*PER*_**FGET_H**(handle, *REG,* *FIELD*)	Returns the value of the specified *FIELD* in the peripheral register REG associated with Handle.
*PER*_**FSET_H**(handle, *REG*, *FIELD,* fieldval)	Sets the value of the specified *FIELD* in the peripheral register REG to fieldval.

1.6 Functions

Table 1–8 provides a generic description of the most common CSL functions where *PER* indicates a peripheral module as listed in Table 1–1 on page 1-4. **Because not all of the functions are available for all the modules, specific descriptions and functions are listed in each module chapter.**

The following conventions are used in Table 1–8:

- Italics indicate variable names.
- Brackets [...] indicate optional parameters.
  - *[handle]* is required only for the handle-based peripherals: DAT, DMA, MCBSP, and TIMER. See Section 1.3.1.1 on page 1-7, *Using CSL Handles*.
  - *[priority]* is required only for the DAT peripheral module.

Table 1–8. Generic CSL Functions

Function	Description
handle = *PER*_**open**(*channelNumber,* *[priority]* *flags*)	Opens a peripheral channel and then performs the operation indicated by *flags*; must be called before using a channel. The return value is a unique device handle to use in subsequent API calls. The *priority* parameter applies only to the DAT module.
*PER*_**config**(*[handle,]* **configStructure*)	Writes the values of the configuration structure to the peripheral registers. You can initialize the configuration structure with: ❑ Integer constants ❑ Integer variables ❑ CSL symbolic constants, *PER_REG*_DEFAULT (see Section 1.4 on page 1-8, 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.

Table 1–8. Generic CSL Functions

Function	Description
*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.

1.6.1 Initializing Registers

The CSL provides two types of functions for initializing the registers of a peripheral: *PER*_config and *PER*_configArgs (where *PER* is the peripheral as listed in Table 1–1 on page 1-4).

- *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. 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 1–2 shows an example of this method.

PER_config and PER_configArgs can be used interchangeably, but it is still necessary to generate the register values. To simplify the process of defining the values to be written to the peripheral registers, the CSL provides the PER_REG_RMK (make) macros, which form merged values from a list of field arguments. Macros are covered in Section 1.5, on page 1-9, *CSL Macros*.

Example 1–1. Using PER_Config

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

Example 1–2. Using PER_configArgs

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

1.7 Support for Device-Specific Features

Not all C54x peripherals offer the exact same type of features across the different C54x devices. Table 1–9 lists specific features that are not common across the C54x family. Also listed are 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, 5440, 5409a, 5410a, 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, 5410, 5420, 5472

(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) MCBSP Module-C2KS Support

C2KS Support	No C2KS Support
5409A, 5410A, 5416, 5421, 5440, 5441	All other C54x supported devices

(e) Watch-dog Module

Watch-dog Timer Support	No Watch-dog Timer Support
5440, 5441	All other C54x supported devices

Table 1–9. Device-Specific Features Support (Continued)

(f) Timer Module

Timer Extended Pre-Scaler Support	**No Timer Extended Pre-Scaler Support**
5472, 5440, 5441	All other C54x supported devices

(g) Chip Module

Device ID Support	**No Secure ID Support**
5416, 5409A, 5410A, 5440, 5441, 5421	All other C54x supported devices

Chapter 2

How to Use CSL

This chapter provides instructions and examples that explain the configuration and use of CSL DSP/BIOS. Specific examples are provided in each module chapter.

Topic **Page**

2.1 Installing the Chip Support Library

Code Composer Studio™ (CCS) release version 2.0 and greater automatically installs the CSL. If you are using an earlier version of CCS, follow these steps to install CSL:

1) Unzip csl.zip into a temporary folder.

2) Copy all C header files (*.h) into c:\ti\C5400\bios\include

3) Copy all library files (*.lib) into c:\ti\C5400\bios\lib

2.2 Overview

With a few exceptions (GPIO, PLL), all of the CSL module functions operate on two types of objects:

- The PER_Handle object
- The PER_Config object

These objects are predefined C structure types which when properly declared and initialized, contain all the information necessary to configure and control the peripheral device.

There are two ways to configure peripherals when using CSL. One is manual configuration by declaring and initializing objects and C source.

The other option is by using the DSP/BIOS Configuration Tool. This method is preferred because the graphical user interface provision that is part of the DSP/BIOS configuration tool is integrated into Code Composer Studio.

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

2.3 DSP/BIOS Configuration Tool: CSL Tree

The DSP/BIOS Configuration Tool allows you to access the CSL graphical interface and configure some of the on-chip peripherals. Each peripheral is represented as a subdirectory of the CSL Tree as shown in Figure 2–1.

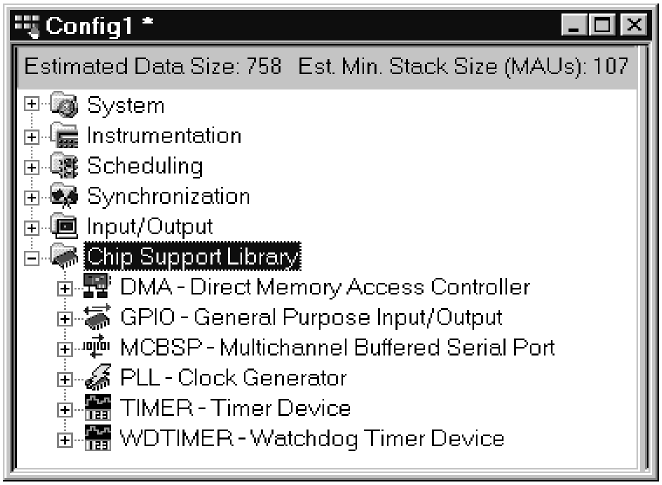
The process consists of three main steps:

1) Creation of the DSP/BIOS configuration file (.cdb file). In Code Composer Studio, select File –> New –> DSP/BIOS Configuration.

2) Configuration of the on-chip peripherals through the CSL hierarchy tree.

3) Automatic generation of the C-code files when saving the configuration file.

Figure 2–1. CSL Tree

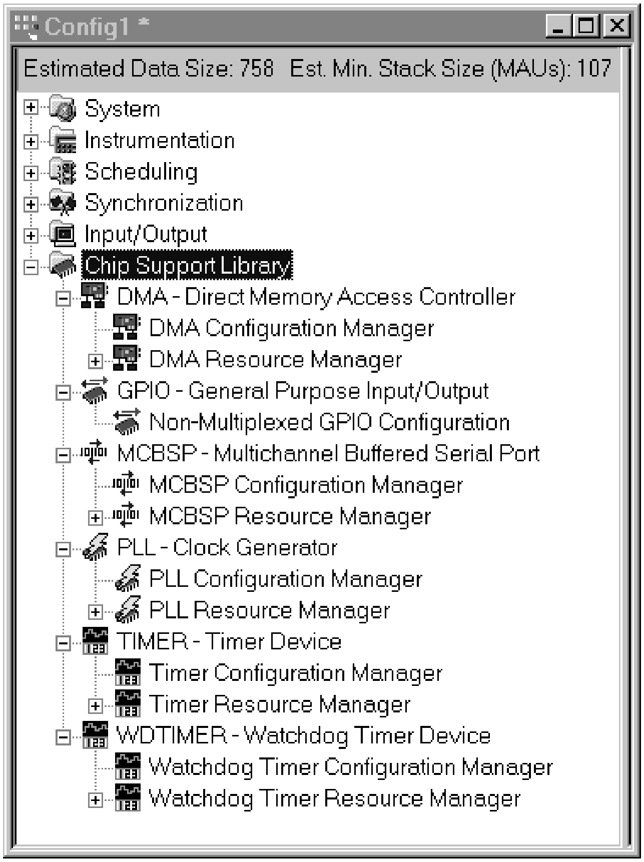


For the TMS320C5400 DSP platform, the peripherals available in the DSP/BIOS Configuration Tool are:

- DMA
- GPIO
- MCBSP
- PLL
- TIMER
- WATCHDOG TIMER

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

Figure 2–2. Expanded CSL Tree



Each peripheral is organized into several sections:

- *PERIPHERAL* **Configuration Manager** – Allows you to set the peripheral register values by selecting the options through the Properties pages. Several configuration objects can be created by selecting the InsertdmaCfg 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 allocate the on-chip device which will be used like a DMA channel, a MCBSP port , or a TIMER device. The handle objects can be renamed only (no deletions permitted).

  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

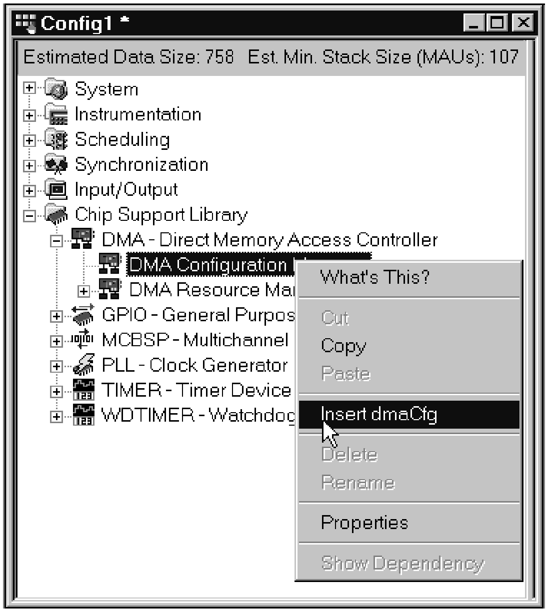


Figure 2–4. Delete/Rename Options

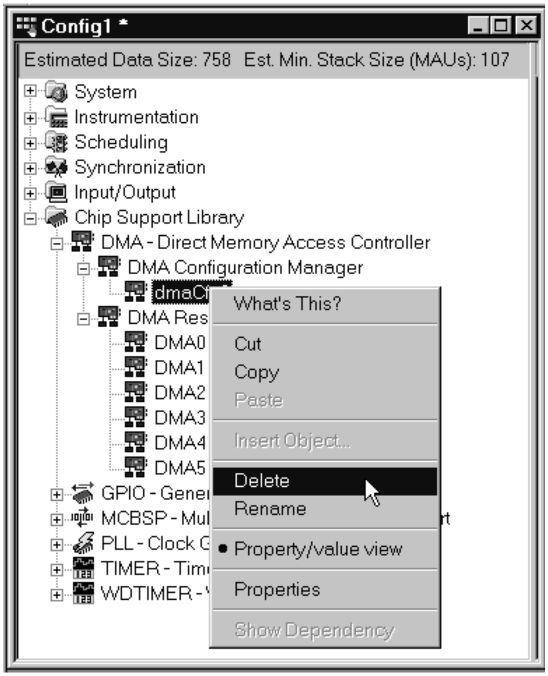
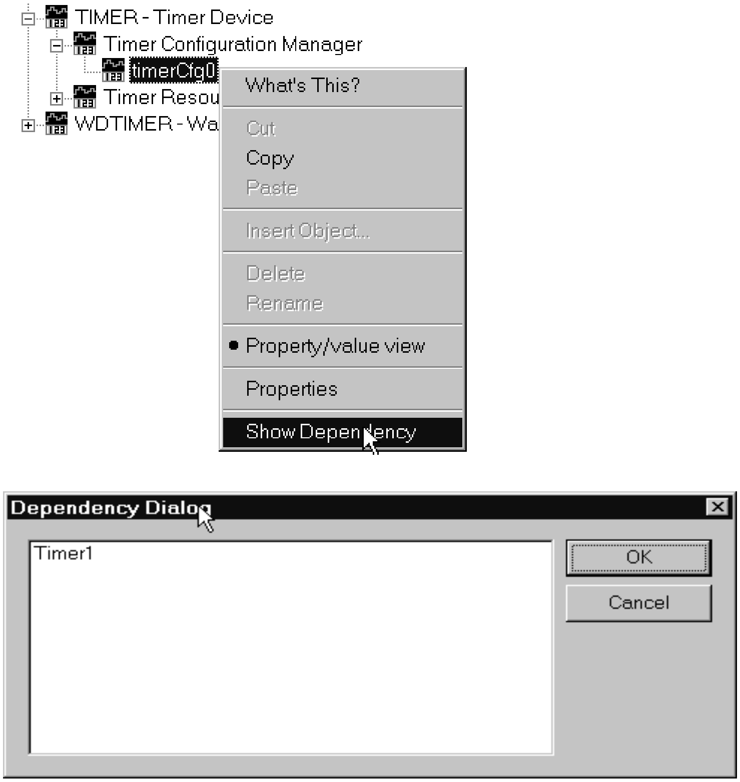


Figure 2–5. Show Dependency Option



2.4 Generation of the C Files (CSL APIs)

After saving the configuration file *project***.cdb**, the following C files are generated:

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

In these examples, *project* is your .cdb file name. The bold characters are attached automatically.

2.4.1 Header File *project*cfg.h

The header file contains several elements:

- The definition of the chip. For example, if the selected chip is 5402, the definition is:

  ```
  #define CHIP_5402 1
  ```

- The csl header files of the CSL tree

  ```
  #include <csl_dma.h>
  #include <csl_emif.h>
  #include <csl_timer.h>
  ```

- The declaration list of the *variables* Handle and configuration names defined in the *project*.cdb. These are declared external, as shown below:

  ```
  extern TIMER_Config timerCfg1;
  extern MCBSP_Config MCBSPmcbspCfg0;
  extern TIMER_Handle hTimer1;
  extern MCBSP_Handle hMcbsp0;
  ```

In order to access the predefined handle and configuration objects, the header file must be included in your project C file:

```
/* User's main .c file */
```

The following line is mandatory and must also be included in your C file:

```
#include <projectcfg.h>
```

2.4.2 Source File *project*cfg_c.c

The source file consists of the Include section, the Declaration section, and the Body section:

- **Include** section:

  This section defines the header file. The source file has access to the data declared in the header file.

  ```
  #include <projectcfg.h>
  ```

Note: If this line is added before the other csl header files (csl_pll, csl_timer, ...), 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 configuration structures and the Handle objects previously defined in the configuration tool.

  The values of the registers reflect the options selected through the Properties pages of each device, as shown in Example 2–1.

Example 2–1. Properties Page Options

```
/*  Config Structures */
TIMER_Config timerCfg0 = {
    0x0010,        /*  Timer Control Register (TCR)   */
    0x0000,        /*  Timer Period Register (PRD)   */
    0x0000         /*  Timer Prescaler Register (PRSC)   */
};


DMA_Config dmaCfg0 = {
    0x0000,        /*  Source Destination Register (CSDP)  */
    0x0000,        /*  Control Register (CCR)  */
    0x0000,        /*  Interrupt Control Register (CICR)  */
    NULL,          /*  Lower Source Address (CSSA_L) – Symbolic(Byte Address)
*/
    NULL,          /*  Upper Source Address (CSSA_U) – Symbolic(Byte Address)
*/
    NULL,          /*  Lower Destination Address (CDSA_L) – Symbolic(Byte Ad-
dress)  */
    NULL,          /*  Upper Destination Address (CDSA_U) – Symbolic(Byte Ad-
dress)  */
    0x0001,        /*  Element Number (CEN)  */
    0x0001,        /*  Frame Number (CFN)  */
    0x0000,        /*  Frame Index (CFI)  */
    0x0000         /*  Element Index (CEI)  */
};



/*  Handles  */
TIMER_Handle hTimer1;
DMA_Handle hDma0;
```

- **Body** section

  The body is composed of a unique function, CSL_cfgInit(), which is called from your C file.

  The function CSL_cfgInit() allows you to allocate/open and configure a device by calling the Peripheral_open() and Peripheral_config() APIs, respectively.

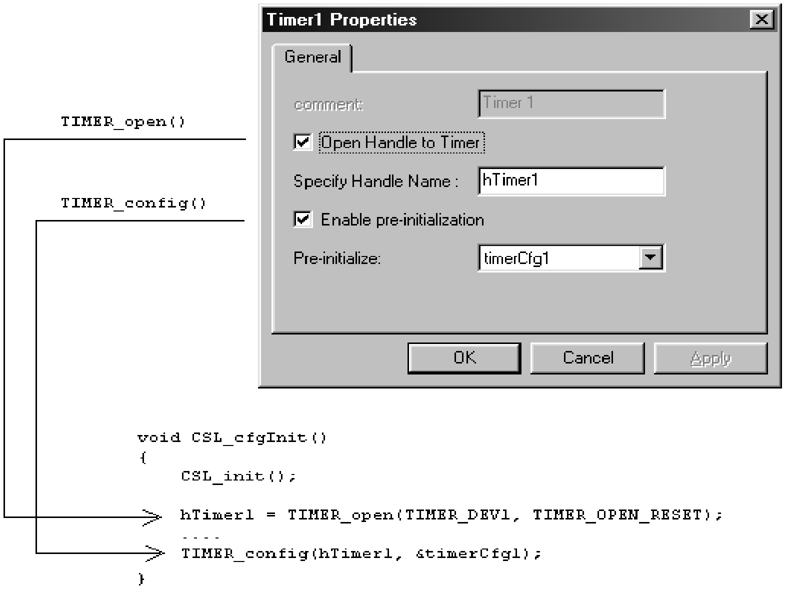
These two functions are generated when the Open Handle to Timer and Enable pre-initialization options are checked in the Properties page of the related Resource Manager (see Figure 2–6).

Note: A device can be allocated/opened without being configured.

In Figure 2–6:

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

Figure 2–6. Resource Manager Properties Page



Before using these predefined APIs, CSL_cfgInit must be called. This function is automatically called by the DSP/BIOS CSL boot/start-up routine.

```
/* User’s file main.c */
void main ()
{
```

2.5 Creating a Configuration

To create a configuration, you must:

1) Modify the Project folder on the Code Composer Studio Interface

2) Modify the C code (main.c).

3) In Code Composer Studio, select File → New → DSP/BIOS Configuration: open Config1.cdb window (default name)

4) Select File → Save as: *project*.cdb (user cdb name)

5) Select Project → Add Files to Project: *project*.cdb (the files *project*cfg.s54 and *projectcfg_c.c* will appear in “generated files” folder)

6) Configure the CSL peripherals Properties pages as needed: Create the configuration objects and Opening of Handles objects. (see section 2.3 and section 2.4.2).

7) Save *project*.cdb

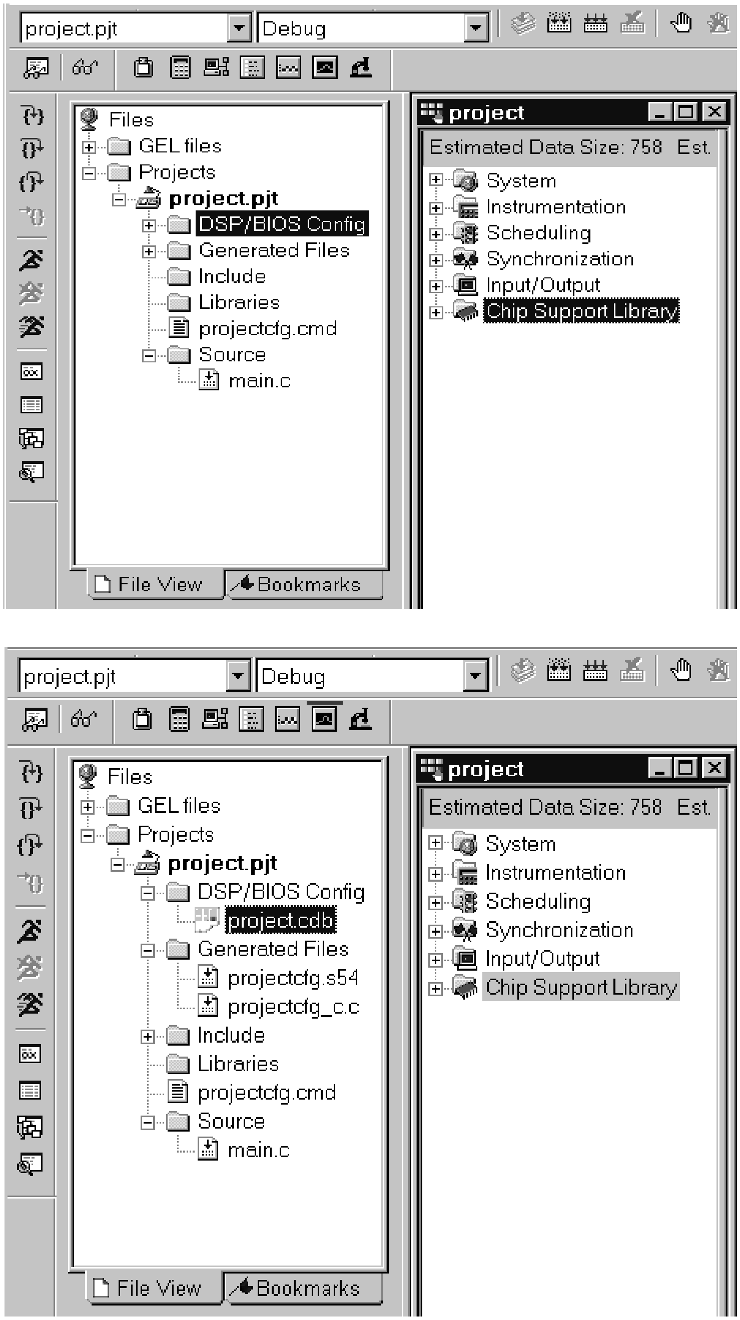
8) Select Project → Add Files to Project

9) Include the following files in your Project:

   - command file: *project*cfg.cmd
   - asm source file: *project.s54* (CSL predefined APIs)

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

Figure 2–7. Practice Summary



2.5.1 Modification of C code (main.c)

To modify the C code (main.c):

1) Add the header file ***#include projectcfg.h*** to your main.c file, As shown In Example 2–2. These lines are required to provide access to the Handle and configuration objects.

Note: CSL_cfgInit() is automatically called by the DSP/BIOS CSL boot/start-up routine. This function pre-opens and pre-configures the peripherals **ONLY**. It does not start device operation. A call to the PER_Start function is required within your code to begin peripheral operation with the pre-chosen settings.

Example 2–2. Modifying the C File

```
/* Include file */
#include projectcfg.h

/* main program */
void main()
{

...
}
```

2.6 Example of CSL APIs Generation (TIMER Module)

This section provides an example using the 5402 Timer1 device, which demonstrates how to open and define a configuration for a TIMER device using the graphical user interface. It also provides a full example of C files generated from a .cdb file by using the Chip Support Library APIs.

Warning:

First, go to Global Settings (System Folder) and select the chip type present on your board.

This step is very important because the chip type affects the setting of the default values of the peripheral registers. Make sure that you have not already created any configuration objects with the wrong chip type selected. Before switching chip types, it is recommended that you delete any existing configuration objects, which have default values that are not identical from one chip to another.

2.6.1 Configuration of the TIMER1 Device

The configuration file *mytimer*.cdb is assumed to be created previously and opened (see section 2.5, *Getting Started,* for more details).

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

Follow these steps:

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

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.

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 2–8. CCS Project View

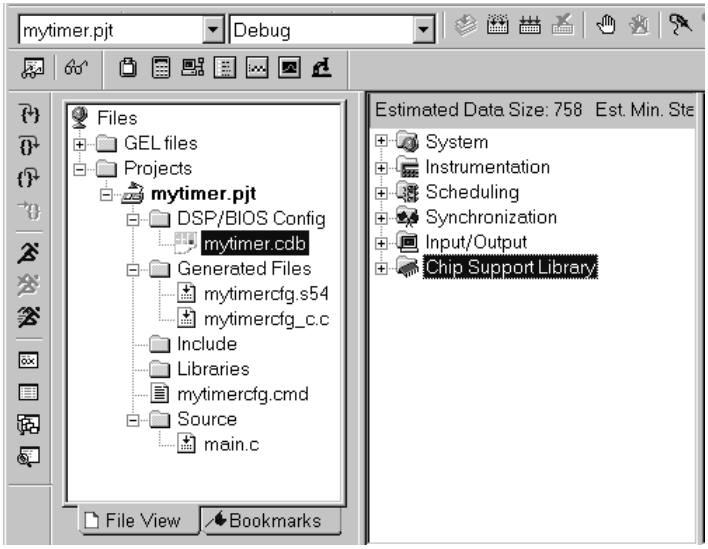
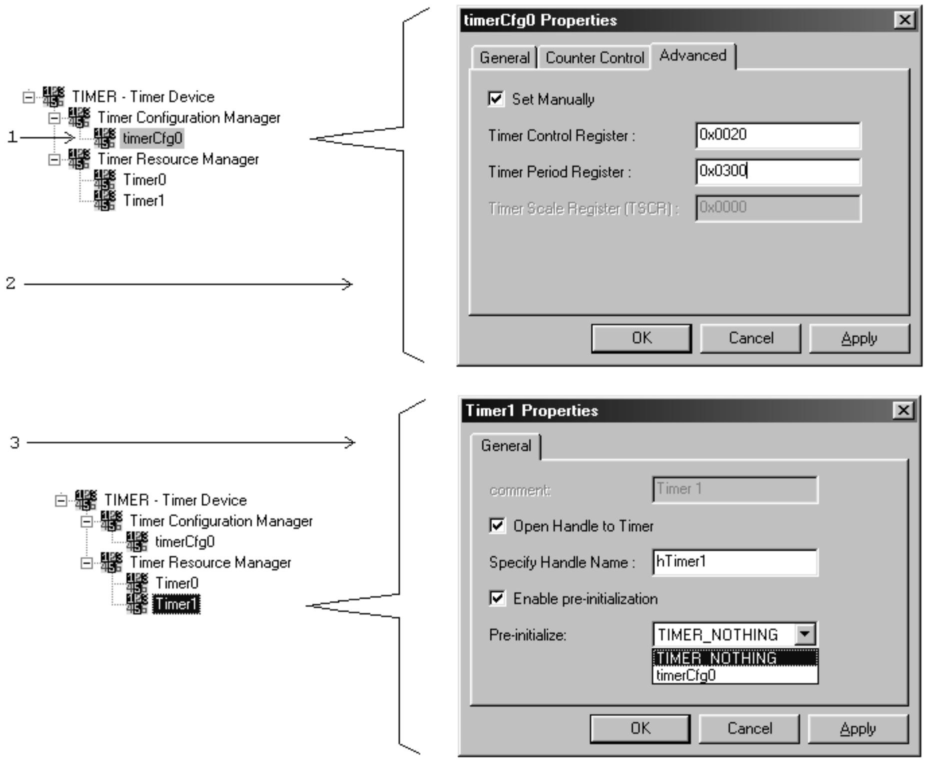


Figure 2–9. Configuring the TIMER1 Device



2.6.2 Generation of C Files

After saving the configuration file *mytimer*.cdb, the header file *mytimer*cfg.h and the source file *mytimer*cfg_c.c are generated (see Figure 2–10 and Figure 2–11).

Figure 2–10. Header File mytimercfg.h

```
/*   Do *not* directly modify this file.  It was    */
/*   generated by the Configuration Tool; any  */
/*   changes risk being overwritten.                */

/* INPUT mytimer.cdb */
#define CHIP_5402 1
/*  Include Header Files  */
#include <std.h>
#include <hst.h>
#include <swi.h>
#include <tsk.h>
#include <log.h>
#include <sts.h>
#include <csl_timer.h>

#ifdef __cplusplus
extern ”C” {
#endif

extern far HST_Obj RTA_fromHost;
extern far HST_Obj RTA_toHost;
extern far SWI_Obj KNL_swi;
extern far TSK_Obj TSK_idle;
extern far LOG_Obj LOG_system;
extern far STS_Obj IDL_busyObj;
extern far TIMER_Config timerCfg0;
extern far TIMER_Handle htimer1;
extern far void CSL_cfgInit();

#ifdef __cplusplus
}
#endif /* extern ”C” */
```

csl header files of the peripherals implemented under the CSL tree

The Handle and Configuration objects are defined and can be used by other C files (User’s files).

Figure 2–11. Source File mytimercfg_c.c

```
/*   generated by the Configuration Tool; any  */
/*   changes risk being overwritten.              */

/* INPUT mytimer.cdb */

/*  Include Header File  */
#include <mytimercfg.h>

/*  Config Structures */
TIMER_Config timerCfg0 = {
    0x0020,        /*  Timer Control Register   */
    0x0300         /*  Timer Period Register   */
};

/*  Handles  */
TIMER_Handle hTimer1;

/*
 *  ======== CSL_cfgInit() ========
 */
void CSL_cfgInit()
{
    CSL_init();

    hTimer1 = TIMER_open(TIMER_DEV1, TIMER_OPEN_RESET);
    TIMER_config(hTimer1, &timerCfg0);
}
```

TIMER Configuration structure timerCfg0 with full TIMER peripheral register values

Handle **hTimer1** declaration

The **TIMER_open()** function returns the handle value in the handle variable hTimer1 previously declared.

TIMER_config() function sets the register values defined by the configuration object timerCfg0.

Figure 2–12. Example of main.c File Using Data Generated by the Configuration Tool

```
#include <csl.h>
#include <csl_timer.h>
#include <csl_irq.h>
#include <mytimercfg.h>

static Uint32 TIMEREventId1;

void main() {


  /* Obtain the event IDs for the TIMER devices */
  TIMEREventId1 = TIMER_getEventId(hTimer1);

  /* Enable the TIMER events */
  IRQ_enable(TIMEREventId1);

  /* Start the TIMERs – */
   TIMER_start(hTimer1);

  /* Waiting for TIMER Interrupt: */
    while( !IRQ_test(TIMEREventId1));

  /* Close TIMER */
  TIMER_close(hTimer1);

  }
```

This line is required and must be included in order to use the peripheral pre-initialization defined through the Configuration Tool.

Handle object “hTimer1” is used directly by the TIMER CSL APIs.

2.7 Configuring Peripherals Without GUI

Note: If you choose not to configure peripherals using GUI, you must pre-define the PER_Handle and PER_Config objects.

Example 2–3 illustrates the use of CSL to initialize DMA channel 0 and to copy a table from address 0x3000 to address 0x2000 using the _config() function. Example 2–4 is similar except that it uses the _configArgs() function.

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

2.7.1 Using DMA_config()

Example 2–3 uses the DMA_config() function to initialize the registers.

Example 2–3. Initializing a DMA Channel with DMA_config()

```
// Step 1:   Include the
//           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_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 = {
       0                                /*priority */
       DMA_DMMCR_RMK(0,0,0,0,1,1,1,1),  /*DMMCR   */
       DMA_DMSFC_RMK(0,0,0),            /*DMSFC    */
       (DMA_Adr_Ptr)&src[0],            /*DMSRC    */
       (DMA_Adr_Ptr)&dst[0],            /*DMDST    */
       (Uint16)(N-1)                    /*DMCTR    */
};
```

Example 2–3. Initializing a DMA Channel with DMA_config() (Continued)

```
//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: Open, configure and start the DMA channel.
//        To configure the channel you can use the
//        DMA_config() or DMA_configArgs() functions.

   myhDma = DMA_open(DMA_CHA0,0);   /* Open Channel            */
   DMA_config(myhDma, &myConfig);  /* Configure Channel      */
   DMA_start(myhDma);              /*  Begin Transfer          */

//Step 6: (Optional).
//        Use CSL DMA APIs to track DMA channel status

   while(DMA_getStatus(myhDma));   /* Wait for complete */

//Step 7: Close DMA channel.

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

2.7.2 Using DMA_configArgs()

Example 2–4 performs the same task as Example 2–3 but uses DMA_configArgs() to initialize the registers.

Example 2–4. Initializing a DMA Channel with DMA_configArgs()

```
// Step 1:   Include the
//           the header file of the module/peripheral you
//           will use <csl_dma.h>. The different header files are shown
//           in  Table 1–1 on page 1-4.
//
```

Example 2–4. Initializing a DMA Channel with DMA_configArgs() (Continued)

```
#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 a DMA_Handle pointer. DMA_open will initialize this handle
//           when a DMA channel is opened.

DMA_Handle myhDma;

void main(void) {

// .....

//Step 3:    Initialize the CSL Library One-time only initialization of the CSL
//           library must be done before calling any CSL module API.

   CSL_init();                      /* Init CSL      */

//Step 4: Open, configure and start the DMA channel.
//        To configure the channel you can use the
//        DMA_config() or DMA_configArgs() functions.

   myhDma = DMA_open(DMA_CHA0,0);/*Open Channel(Optional) */
   DMA_configArgs(
      0                             /* priority   */
       DMA_DMMCR_RMK(0,0,0,0,1,1,1,1),  /* DMMCR     */
       DMA_DMSFC_RMK(0,0,0),            /* DMSFC     */
       (DMA_Adr_Ptr)&src[0],            /* DMSRC     */
       (DMA_Adr_Ptr)&dst[0],            /* DMDST     */
       (Uint16)(N-1)                    /* DMCTR     */
   );
   DMA_start(myhDma);            /* Begin Transfer    */

//Step 6: (Optional)
//        Use CSL DMA APIs to track DMA channel status

   while(DMA_getStatus(myhDma));   /* Wait for complete */

//Step 7: Close DMA channel.

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

2.8 Compiling and Linking With CSL

After writing your program, you have two methods available for compiling and linking your project:

- Use the DOS command line.
- Use the Code Composer Studio project build environment.

Table 2–1 lists the location of the CSL components after installation. Use this information when you set up the compiler and linker search paths. Section 2.8.3, *Creating a Linker File*, on page 2-27, explains specific requirements for the linker command file.

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\src
Include files	c:\ti\c5400\bios\include
Examples	c:\ti\examples\csl
Documentation	c:\ti\docs

2.8.1 Using the DOS Command Line

To compile and link your project using the DOS Command line:

1) Set the include file and library search paths.

Before you compile and link your program, you must verify that the include file search paths are correctly set for the compiler and that the library search path is correctly set for the linker. You can set these paths either in the autoexec.bat file or with the -i option.

- To set the include and library search paths, using the autoexec.bat file, add the following line to the autoexec.bat file:

```
SET C54X_C_DIR=.;C:\ti\c5400\bios\include;C:\ti\c5400\bios\lib;%C54X_C_DIR%
```

- To set the include and library search paths using the -i option, add the following when compiling and linking:

```
-i c:\ti\c5400\bios\include   (for the compiler)
-i c:\ti\c5400\bios\lib       (for the linker)
```

2) Select the correct C54x device and library to link to.

❑ To compile and link for near mode, type the following on the command line:

```
cl500 -dCHIP_5402 ex1.c  csl5402.lib linker.cmd -oex1.out
```

❑ To compile and link for far mode, type the following on the command line:

```
cl500 -mf -v548 -dCHIP_5402 ex1.c csl5402x.lib linker.cmd -oex1far.out
```

Notice the usage of the device support symbol CHIP_5402 (see Table 1–2, on page 1-4) to control conditional compilation. This usage is required because the C54x family offers different peripheral features that are specific to a particular C54x device.

2.8.2 Using the Code Composer Studio Project Environment

You must configure the CCS project environment to work with CSL. To configure the CCS Project environment, follow these steps listed below.

❑ Specify the target device you are using:

1) In Code Composer Studio, select Project→Options

2) In the Build Options dialog box, select the Compiler tab (see Figure 2–13).

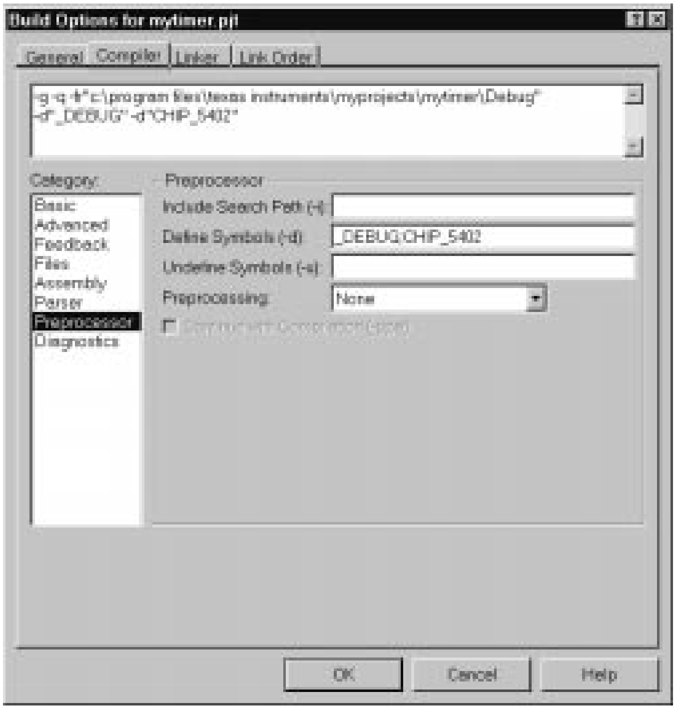
3) In the Category list box, highlight Preprocessor.

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

   For example, if you are using the 5402 device, enter CHIP_5402.

5) Click OK.

Figure 2–13. Defining the Target Device in the Build Options Dialog



- ❑ If you use any far-mode libraries, define far mode for the compiler and link with the far mode runtime library (rts_ext.lib):

1) In Code Composer Studio, select Project→Options

2) In the Build Options dialog box, select the Compiler Tab (Figure 2–14),

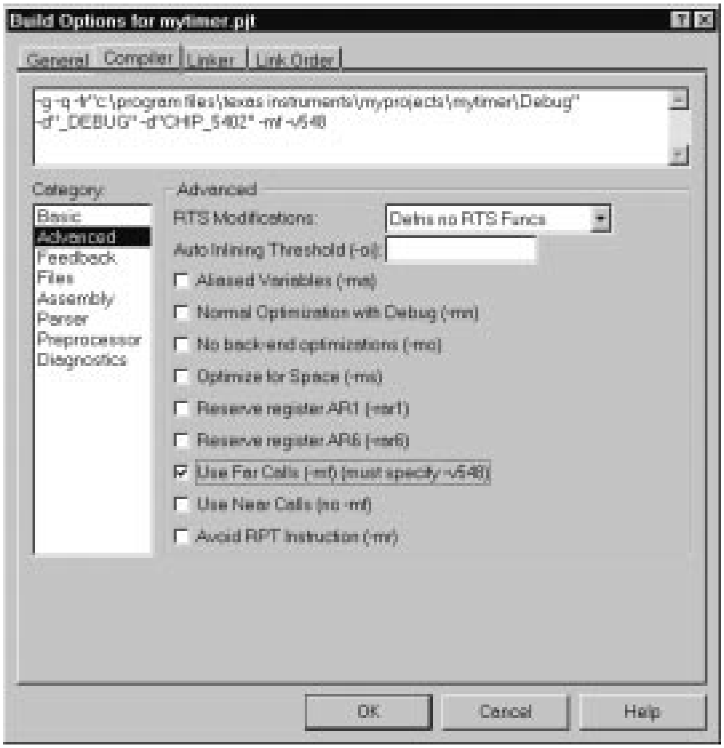
3) In the Category list box, highlight advanced.

4) Select Use Far Calls.

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

6) Click OK.

Figure 2–14. Defining Far Mode



- ❑ If you are using Code Composer Studio releases prior to 2.0, add the search path for the header files:

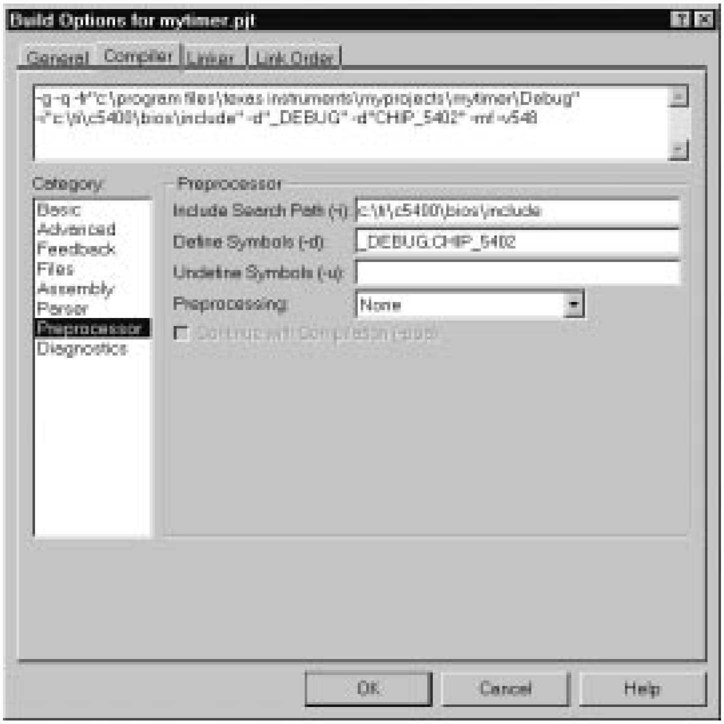
1) In Code Composer Studio, select Project→Options...

2) In the Build Options Dialog box, select the Compiler Tab (see Figure 2–15).

3) In the Include Search Path field (-i), type:
c:\ti\c5400\bios\include

4) Click OK.

Figure 2–15. Adding the Include Search Path



- ❑ Specify the search path for the CSL library:

1) In Code Composer Studio, select Project→Options

2) In the Build Options dialog box, Select the Linker Tab (see Figure 2–16).

3) In the Category list, highlight Basic.

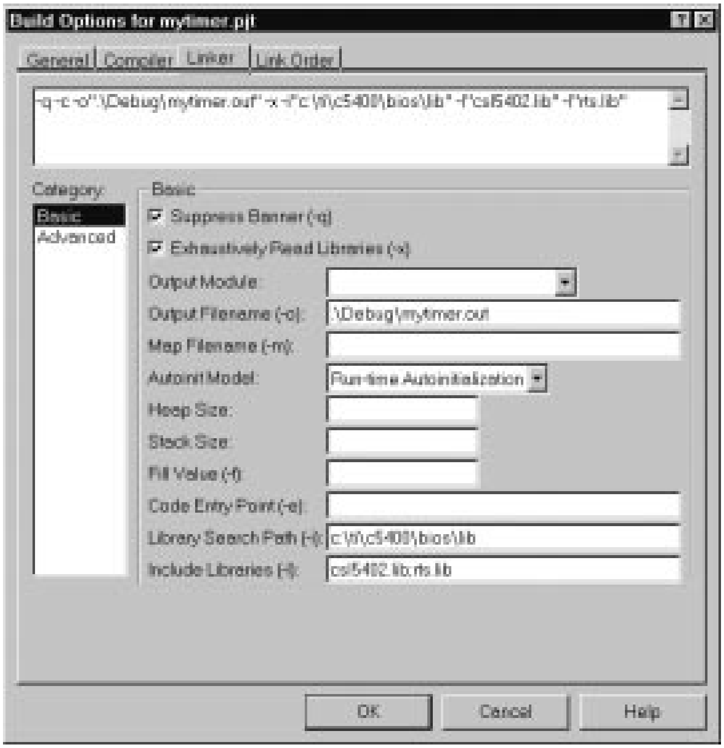
4) In the Library search Path field (-l), type:
   c:\ti\c5400\bios\lib

5) In the Include Libraries (-i) field, enter the correct library from Table 1–2, on page 1-4.

   For example, if you are using the 5402 device, enter csl5402.lib for near mode or csl5402x.lib for far mode.

6) Click OK.

Figure 2–16. Defining Library Paths



2.8.3 Creating a Linker Command File

The CSL has two requirements for the linker command file:

- **You must allocate the .csl data 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 .csl data 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 2–5 illustrates these requirements which must be included in the linker command file.

Example 2–5. Using a Linker Command File

```
MEMORY
{
   PAGE 0:   PROG0: origin =  4000h, length = 0D000h
             PROG1: origin =  18000h, length = 08000h
   PAGE 1:   DATA:  origin =  0800h, length = 03800h
}
SECTIONS
{
   .text   > PROG0 PAGE 0
   .cinit  > PROG0 PAGE 0
   .switch > PROG0 PAGE 0

   .data   > DATA PAGE 1
   .bss    > DATA PAGE 1
   .const  > DATA PAGE 1
   .sysmem > DATA PAGE 1
   .stack  > DATA PAGE 1
   .csl data > DATA PAGE 1
   table1 : load =  3000h PAGE 1
   table2 : load =  2000h PAGE 1
}
```

2.9 Rebuilding CSL

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

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

2.10 Using Function Inlining

Because some CSL functions are short, they set only a single bit field. In this case, incurring the overhead of a C function call is not always necessary. If you enable inline, the API declares these functions as *static inline*. Using this technique can help reduce code size. In order to allow for future changes, the CSL documentation does not identify which functions are inlined; however, if you enable function inlining with the compiler -x option, you see an increase in CSL code performance.

Chapter 3

DSP/BIOS Configuration Tool: CSL Modules

Note: In most cases, you are not required to use the DSP/BIOS™ configuration tool to configure peripherals.

The Chip Support Library (CSL) graphical user interface is part of the DSP/BIOS™ configuration tool integrated in Code Composer Studio (CCS). This graphical user interface (GUI) benefits you by reducing manual C-code generation and offering an easy way to use on-chip peripherals by programming the associated Peripheral registers through the properties pages.

Topic **Page**

3.1 Overview

Chapter 2 outlined the basic CSL program flow and illustrated the use of CSL macros in C source for declaring and defining the necessary PER_Handle and PER_Config objects needed for peripheral operation in CSL.

As an alternative to the manual declaration and initialization of the peripheral configuration objects within the C source described in chapter 2, CSL also provides a graphical user interface (GUI) that is part of the DSP/BIOS configuration tool and is integrated into Code Composer Studio.

The CSL graphical user interface (GUI) provides the benefit of a visual tool that allows you to view the chosen register settings, determine which flags/options have been set by a particular mode selection, and most importantly, have the code for the configuration settings automatically be created and stored in a C source file that can be integrated directly into your application.

3.2 DMA Module

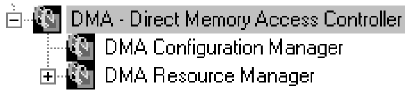
3.2.1 Overview

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 3–1 illustrates the DMA sections menu on the CSL graphical user interface (GUI).

Figure 3–1. 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.

3.2.2 DMA Configuration Manager

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

3.2.2.1 Creating/Inserting a configuration

There is no predefined configuration object available.

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

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.

3.2.2.2 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 Section 2.3, *DSP/BIOS Configuration Tool: CSL Tree,* on page 2-4).

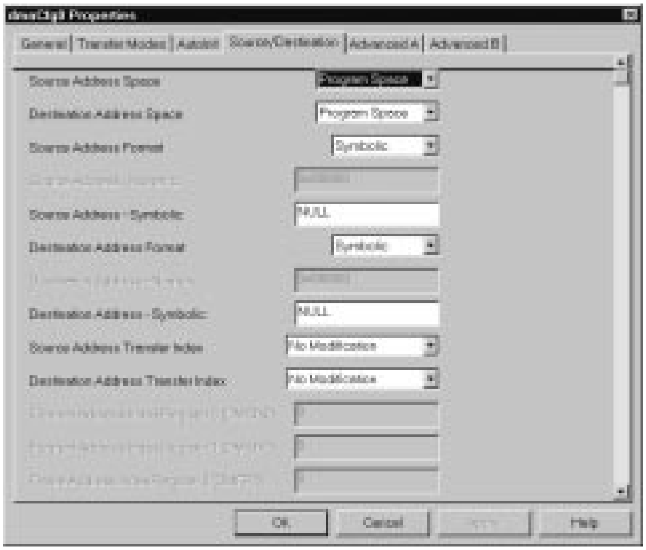
3.2.2.3 Configuring the Object Properties

You can configure object properties through the Properties dialog box. (See Figure 3–2). 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:

- 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: This page 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 3–2. DMA Properties Page



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

3.2.2.4 Address Formats

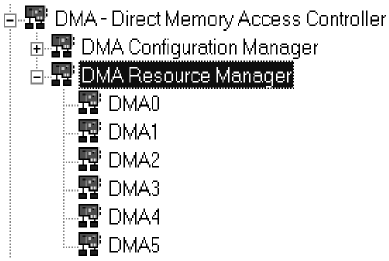
The source, destination, and addresses can be specified in either a numeric format (hard coded address) or a symbolic format. Before setting any addresses, it is suggested that you ensure that the right format is selected in the Source Address Format and Destination Address Format pull-down menus located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

3.2.3 DMA Resource Manager

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

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

Figure 3–3. DMA Resource Manager Menu



3.2.3.1 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

3.2.3.2 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 3–4).

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.

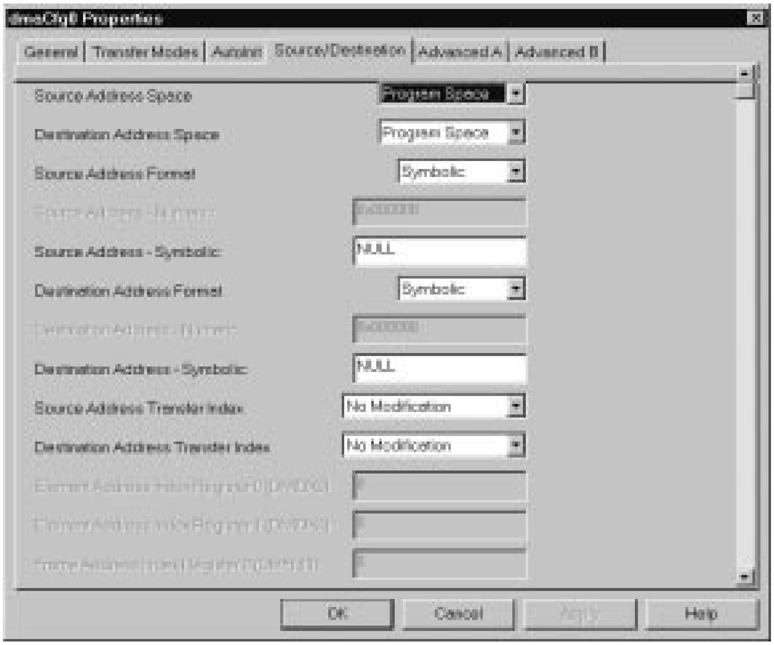
DMA_NOTHING is used to indicate that there is no configuration object selected for this DMA.

To pre-initialize the DMA channel, check the Enable pre-initialization check-box. You can then select one of the available configuration objects (see section 3.2.2 , *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 3.2.4, *C Code Generation for DMA Module, on page 3-7)*.

In the example shown in Figure 3–4, 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 3–4. DMA Properties Page With Handle Object Accessible



3.2.4 C Code Generation for DMA Module

Two C files are generated from the configuration tool:

- ❑ Header file
- ❑ Source file.

3.2.4.1 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 3–1).

Example 3–1. DMA Header File

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

3.2.4.2 Source File

The source file includes the declaration of the channel handle objects and the configuration structures (see Example 3–2).

Example 3–2. 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 3–3).

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 3–3. DMA Source File (Body Section)

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

3.3 GPIO Module

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 of either the input or output of the GPIO pins.

Figure 3–5 illustrates the GPIO sections menu on the CSL graphical user interface (GUI)

3.3.1 Overview

Figure 3–5. GPIO Sections Menu

The Non-Multiplexed GPIO includes the following section:

- **Non-Multiplexed GPIO Configuration Manager**: Allows you to configure the GPIO Pin directions.

3.3.2 Non-Multiplexed GPIO Configuration Manager

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

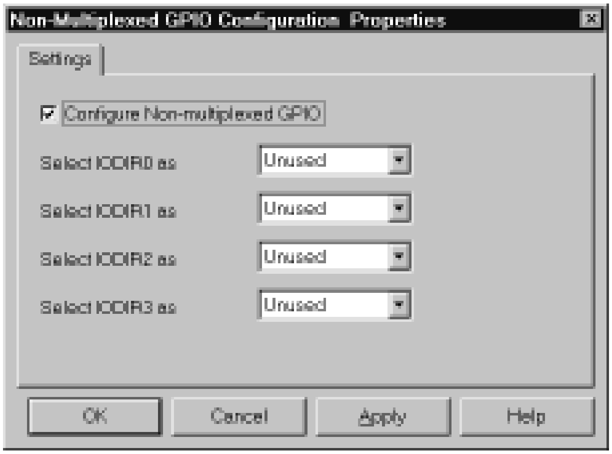
3.3.2.1 Properties Pages of the Non-Multiplexed GPIO Configuration

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 3–6, Non-Multiplexed GPIO *Properties Page*, depicts the Properties Page dialog box.

Figure 3–6. 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; the associated field name is shown in parenthesis. For further details of the fields and registers, refer to the GPIO section of the *TMS320C54x DSP Enhanced Peripherals Reference Set* (literature number SPRU302).

3.3.3 C Code Generation for GPIO Module

Two C files are generated from the configuration tool:

- Header file
- Source file.

3.3.3.1 Header File

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

3.3.3.2 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 3–6.

Example 3–4. GPIO Source File (Body Section)

```
void CSL_cfgInit()
{
    CSL_init();

    GPIO_RSET(IODIR, 3840);
}
```

3.4 MCBSP Module

3.4.1 Overview

The MCBSP module facilitates configuration/control of the Multi Channel 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 3–7 illustrates the GPIO sections menu on the CSL graphical user interface (GUI)

Figure 3–7. 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.

3.4.2 MCBSP Configuration Manager

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

3.4.2.1 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.

3.4.2.2 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 Section 2.2, *Introduction to DSP/BIOS Configuration Tool: CSL Tree*, on page 2-3).

3.4.2.3 Configuring the Object Properties

The Properties pages allow you to set the Peripheral registers related to the MCBSP Port (see Figure 3–8). 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.

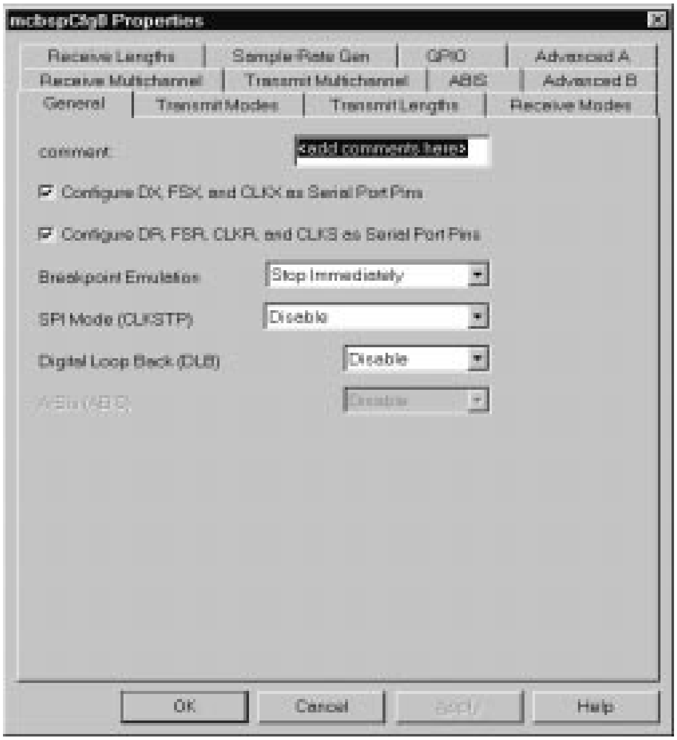
The Properties pages allow you to set the Peripheral registers related to the MCBSP. you can set the configuration options through the following pages:

- General: Allows you to configure the Digital Loopback, ABIS Mode, Breakpoint Emulation.
- Transmit Modes: Allows you to configure the Interrupt mode, Frame Sync, Clock control.
- Transmit Lengths: Allows you to configure the Phase, elements-per-word, elements per frame.
- Receiver Modes: Allows you to configure the Interrupt mode, Frame Sync, Clock control.
- Receiver Lengths: Allows you to configure the Phase, elements-per-word, elements per frame.
- Sample-Rate Generator: Allows you to configure the Sample-Rate Generator (Frame Setup).
- Receive Multi-channel: Allows you to configure the Element and Block partitioning.
- Transmit Multi–channel: Allows you to configure the Element and Block partitioning.
- 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. This page contains the full hexadecimal register values and reflects the setting of the options done under the previous pages
- ❑ The full register values can be entered directly and the new options will be mirrored on the corresponding pages automatically.

Figure 3–8, *MCBSP Properties Page*, depicts the Properties Page.

Figure 3–8. MCBSP Properties Page



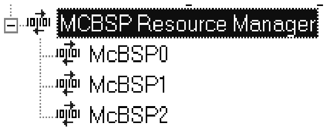
Each Tab page is composed of several options that are set to a default value (at device reset).

3.4.3 MCBSP Resource Manager

The MCBSP Resource Manager allows you to generate the MCBSP_open() and the MCBSP_config() CSL functions.

Figure 3–9 illustrates the MCBSP Resource Manager menu on the CSL graphical user interface (GUI).

Figure 3–9. MCBSP Resource Manager Menu



3.4.3.1 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 3.4.2, *MCBSP Configuration Manager*, on page 3-12.

3.4.3.2 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 3–10).

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 3.4.2, *MCBSP Configuration Manager*, on page 3-12) 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 3.4.4, *C Code Generation for MCBSP Module*, on page 3-16).

In the example shown in Figure 3–10, 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 3–10. MCBSP Properties Page With Handle Object Accessible

3.4.4 C Code Generation for MCBSP Module

Two C files are generated from the configuration tool:

- ❑ Header file
- ❑ Source file.

3.4.4.1 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 3–5).

Example 3–5. MCBSP Header File

```
extern MCBSP_Config mcbsCfg0;
extern MCBSP_Handle hMcbsp1;
```

3.4.4.2 Source File

The source file includes the declaration of the handle object and the configuration structures (see Example 3–6).

Example 3–6. 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 API functions, MCBSP_open() and MCBSP_config() (see Example 3–7). 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 DMA and Enable pre-initialization (with a selected configuration other than MCBSP_NOTHING) are, respectively, checked under the MCBSP Resource Manager Properties page.

Example 3–7. MCBSP Source File (Body Section)

```
void CSL_cfgInit()
{
    CSL_init();

    hMcbsp1 = MCBSP_open(MCBSP_PORT1, MCBSP_OPEN_RESET);
    MCBSP_config(hMcbsp1, &mcbspCfg0);
}
```

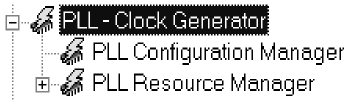
3.5 PLL Module

3.5.1 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 3–11 illustrates the PLL sections menu on the CSL graphical user interface (GUI).

Figure 3–11. 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.

3.5.2 PLL Configuration Manager

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

3.5.2.1 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: 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.

3.5.2.2 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 one of the predefined handle objects 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 Section 2.2, *Introduction to DSP/BIOS Configuration Tool: CSL Tree*.

3.5.2.3 Configuring the Object Properties

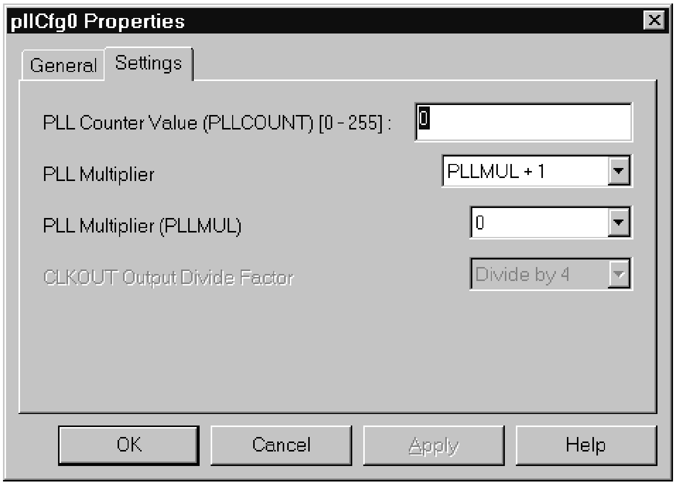
You can configure object properties through the Properties dialog box (see Figure 3–12). To access the Properties dialog box, right-click on a configuration object and select Properties. By default, the General page of the Properties dialog box is displayed.

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 3–12, *PLL Properties Page*, depicts the Properties Page dialog box.

Figure 3–12. 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 of the fields and registers, refer to the *Expansion Bus* chapter of the *TMS320C54x DSP CPU and Peripherals References Set* (literature number SPRU131F).

3.5.3 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 3–13 illustrates the PLL Resource Manager menu on the CSL graphical user interface (GUI).

Figure 3–13. PLL Resource Manager Menu

3.5.3.1 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 3–14).

The first time the properties page appears, only the Enable Configuration PLL check box can be selected. Select this to enable the PLL configuration.

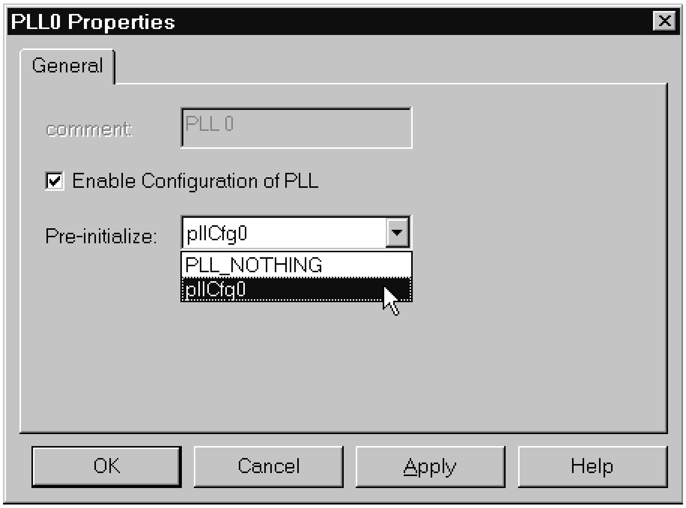
PLL_NOTHING is used to indicate that there is no configuration object selected for this peripheral.

To pre-initialize the PLL channel, check the Enable Configuration of PLL box. One of the available configuration objects(see Section 3.2.2 , *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 3–14, the pllCfg0 is selected and the PLL_config function will be generated. (See Section 3.5.4, *C Code Generation for PLL Module*, on page 3-22.)

Figure 3–14. PLL Properties Page



3.5.4 C Code Generation for PLL Module

Two C files are generated from the configuration tool:

- Header file
- Source file.

3.5.4.1 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 3–8).

Example 3–8. PLL Header File

```
extern PLL_Config pllCfg0;
```

3.5.4.2 Source File

The source file includes the declaration of the configuration structures (values of the peripheral registers) (see Example 3–9).

Example 3–9. 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 3–10).

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 3–14, on page 3-22).

Example 3–10. PLL Source File (Body Section)

```
void CSL_cfgInit()
{
    CSL_init();

   PLL_config(&PLLCfg0);
}
```

3.6 TIMER Module

3.6.1 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 3–15 illustrates the Timer sections menu on the CSL graphical user interface (GUI).

Figure 3–15. 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. Three handle objects are predefined.

3.6.2 TIMER Configuration Manager

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

3.6.2.1 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: 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.

3.6.2.2 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 3.7.3, *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 Section 2.2, *Introduction to DSP/BIOS Configuration Tool: CSL Tree*, on page 2-3).

3.6.2.3 Configuring the Object Properties

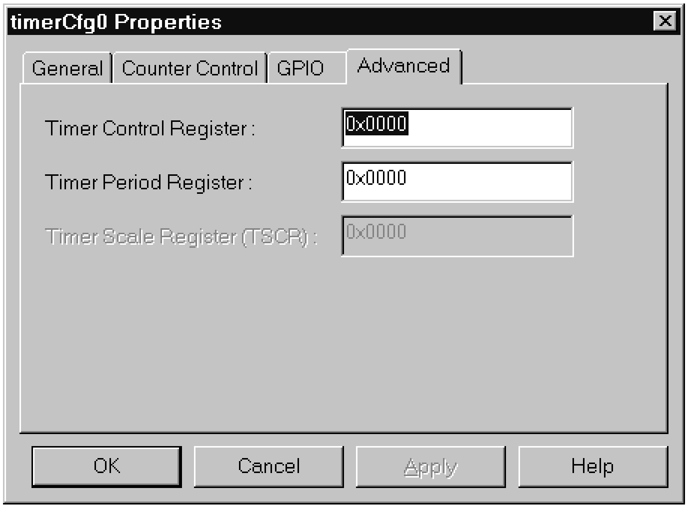
You can configure object properties through the Properties dialog box (see Figure 3–16). To access the Properties dialog box, right-click on a configuration object and select Properties. By default, the General page of the Properties dialog box is displayed.

The Properties pages allow you to set the 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
- ❑ 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 3–20, *TIMER Properties Page*, depicts the Properties Page dialog box.

Figure 3–16. TIMER Properties Page



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

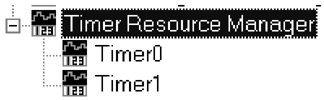
The options represent the fields of the TIMER registers; the associated field name is shown in parenthesis. For further details on the fields and registers, refer to the *Timers* chapter in the *TMS320C54x Chip Support Library API Reference Guide* (literature number SPRU420).

3.6.3 TIMER Resource Manager

The TIMER Resource Manager allows you to generate the TIMER_open() and the TIMER_config() CSL functions.

Figure 3–17 illustrates the DMA Resource Manager menu.

Figure 3–17. Timer Resource Manager Menu



3.6.3.1 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 3.7.2, *TIMER Configuration Manager*, on page 3-30.

3.6.3.2 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 3–18).

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.

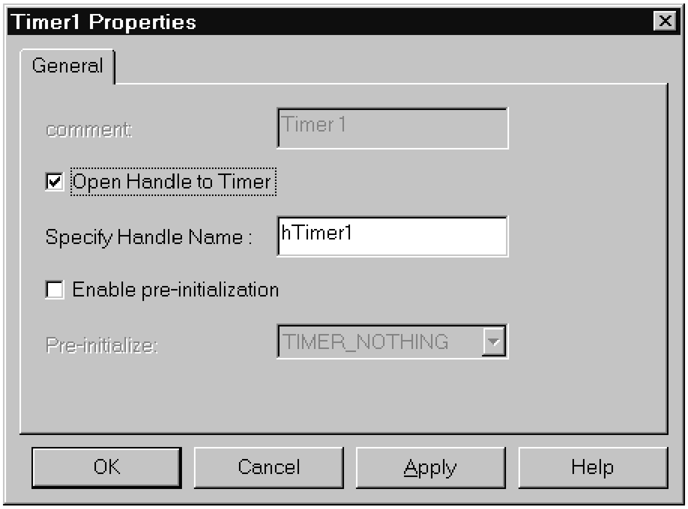
TIMER_CFGNULL is used to indicate that there is no configuration object selected for this device.

To pre-initialize the TIMER channel, check the Enable Pre-Initialization box. One of the available configuration objects(see Section 3.2.2 , *TIMER Configuration Manager*) can then be selected for this channel through the Pre–Initialize drop-down list.

If TIMER_CFGNULL is selected, no configuration object will be generated for the related TIMER handle. (See Section 3.7.4, *C Code Generation for TIMER*, on page 3-33.)

In Figure 3–22, *Timer Properties Page With Handle Object Accessible*, the Open Handle to TIMER option is checked and the handle object hTimer0 is now accessible (renaming allowed). The TIMER_open() function will be generated with hTimer0 containing the return handle address.

Figure 3–18. Timer Properties Page With Handle Object Accessible



3.6.4 C Code Generation for TIMER

Two C files are generated from the configuration tool:

- Header file
- Source file.

3.6.4.1 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 3–11).

Example 3–11. Timer Header File

```
extern TIMER_Config timerCfg0;
extern TIMER_Handle hTimer1
```

3.6.4.2 Source File

The source file includes the declaration of the handle object and the configuration structures (see Example 3–12).

Example 3–12. 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 3–13). 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.

Example 3–13. Timer Source File (Body Section)

```
    void CSL_cfgInit()
    {
       CSL_init();

       hTimer1 = TIMER_open(TIMER_DEV1, TIMER_OPEN_RESET);
       TIMER_config(hTimer1, &timerCfg1);

    }
```

3.7 WATCHDOG TIMER Module

3.7.1 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 3–19 illustrates the WATCHDOG TIMER sections menu on the CSL graphical user interface (GUI).

Figure 3–19. 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.

3.7.2 WATCHDOG TIMER Configuration Manager

The WATCHDOG TIMER Configuration Manager allows you to create device configurations through the Properties page and generate the configuration objects.

3.7.2.1 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.

3.7.2.2 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 Section 2.2, *Introduction to DSP/BIOS Configuration Tool: CSL Tree*, on page 2-3.

3.7.2.3 Configuring the Object Properties

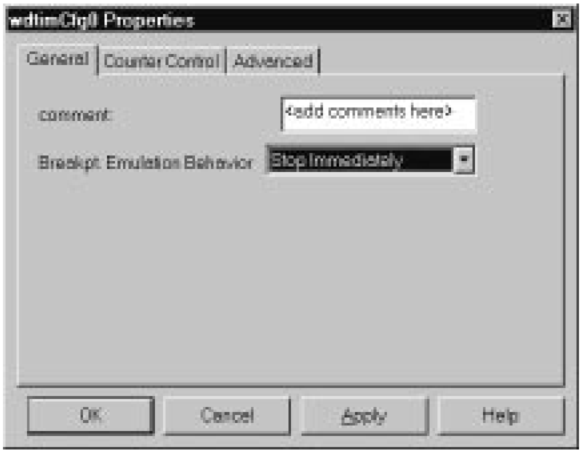
You can configure object properties through the Properties dialog box (see Figure 3–20). To access the Properties dialog box, right-click on a configuration object and select Properties. By default, the General page of the Properties dialog box is displayed.

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 three pages
- This page contains the full hexadecimal register values and reflects the setting of the three pages
- The full register values can be entered directly and the new options will be mirrored on the previous three pages automatically

Figure 3–20, *WATCHDOG TIMER Properties Page*, depicts the Properties Page dialog box.

Figure 3–20. 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. For further details on the fields and registers, refer to the *WATCHDOG TIMER* chapter in the *TMS320C55xx Chip Support Library API Reference Guide* (SPRU420).

3.7.3 WATCHDOG TIMER Resource Manager

The WATCHDOG TIMER Resource Manager allows you to generate the WDTIM_config() CSL function.

Figure 3–21 illustrates the WATCHDOG TIMER Resource Manager Menu.

Figure 3–21. WATCHDOG TIMER Resource Manager Menu

3.7.3.1 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 3–22).

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.

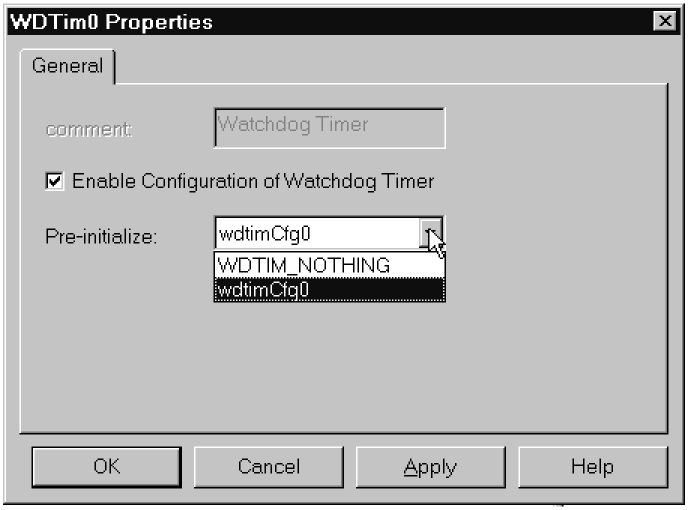
WDTIM_NOTHING is used to indicate that there is no configuration object selected for this device.

To pre-initialize the Watchdog Timer, check the Enable Configuration of WATCHDOG TIMER box. One of the available configuration objects (see Section 3.2.2 , *Watchdog Timer Configuration Manager*) 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 handle. (See Section 3.7.4, *C Code Generation for WATCHDOG TIMER*, on page 3-33.)

In Figure 3–22, the Enable Configuration of Watchdog Timer option is checked and wdtimCfg0 is now accessible. The Wdtim_open() function will be generated.

Figure 3–22. WATCHDOG TIMER Properties Page



3.7.4 C Code Generation for WATCHDOG TIMER

Two C files are generated from the configuration tool:

❑ Header file

❑ Source file.

3.7.4.1 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 3–14).

Example 3–14. WATCHDOG TIMER Header File

```
extern WDTIM_Config wdtimCfg0;
```

3.7.4.2 Source File

The source file includes the declaration of the configuration structures (see Example 3–15).

Example 3–15. WATCHDOG TIMER Source File (Declaration Section)

```
/*  Config Structures */
WDTIM_Config WdtimCfg0 = {
    0x0000,        /*  Timer Control Register   */
    0x0000,        /*  Timer Period Register   */
    0x0000,        /*  Timer Register (TIM)  */
    0x0000        /*  Timer Scale Register (TSCR)  */
};
```

The source file contains the Configuration Pre-Initialization using the CSL WATCHDOG TIMER API WDTIM_config() (see Example 3–16). 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.

Example 3–16. WATCHDOG TIMER Source File (Body Section)

```
    void CSL_cfgInit()
    {
       CSL_init();

       WDTIM_config(&wdtimCfg0);
    }
```

Chapter 4

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**

4.1 Overview

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

Table 4–1 lists the functions available as part of the CHIP module.

Table 4–1. CHIP Functions

Function	Purpose	See page ...
CHIP_getCpuId	Returns the CPU ID field of the CSR register.	4-3
CHIP_getEndian	Returns the current endian mode of the device	4-3
CHIP_getRevID	Returns the CPU revision ID.	4-4
CHIP_getSubsysID	Returns sub-system ID (or core) for a multi-core device.	4-5
CHIP_getMapMode	Returns the current MAP mode of the device.	4-4

4.2 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 ID

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

Example

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

CHIP_getEndian

Get endian mode (C5416 and C5421 only)

Function Uint16 CHIP_getEndian();

Arguments None

Return Value Endian mode CHIP_ENDIAN_LITTLE = 1

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

Example

```
UINT16 Endian;
…
Endian = CHIP_getEndian();
```

CHIP_getMapMode *Read 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 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 (C5410, C5411, 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 CSR register.

Example

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

CHIP_getSubsysId — *Get subsystem ID (5440 only)*

Function Uint32 CHIP_SubsysId();

Arguments None

Return Value Subsytem ID

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

Example

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

Chapter 5

DAT Module

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

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_open()	Opens a DAT channel	5-8
DAT_copy()	Copies a linear block of data from src to dst using DMA hardware	5-3
DAT_copy2D()	Copies 2D data from src to dst using DMA hardware	5-5
DAT_fill()	Fills a linear block of memory with the specified fill value using DMA hardware	5-7
DAT_wait()	Waits for a previous transfer to complete	5-9
DAT_close()	Closes a DAT channel	5-3

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
- ❑ DAT_DATA_SPACE
- ❑ DAT_IO_SPACE

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_CHAANY,DAT_PRI_LOW,0);
   DAT_copy(
      hDat,
      BuffA | DAT_DATA_SPACE,
      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
- DAT_2D1D
- DAT_2D2D

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

- DAT_PROGRAM_SPACE
- DAT_DATA_SPACE
- DAT_IO_SPACE

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 Depending on the type of 2D DMA data transfer request, this function copies 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_CHAANY,DAT_PRI_LOW,0);
   DAT_copy2D(
      hDat,
      DAT_2D2D,
      BuffA | DAT_DATA_SPACE,
      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

```
Uint16 BUFF_SIZE 256;
Uint16 Buff[BUFF_SIZE];
Uint16 FillValue = 0xA5A5;
DAT_Handle hDat;
...
   hDat = DAT_open(DAT_CHAANY,DAT_PRI_LOW,0);
   DAT_fill(
      hDat, Buff | DAT_DATA_SPACE, BUFF_SIZE, &FillValue
);
```

DAT_open

Opens 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_CHAANY (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_CHAANY,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;
man(){
…
   hDat = DAT_open(DAT_CHAANY, DAT_PRI_LOW, 0);
   …
   TransferStat = DAT_copy(hDat, src,dst,len);
   /* custom DAT configuration */
   if (TransferStat)
   DAT_wait(hDat, TransferStat);
...
}
```

Chapter 6

DMA Module

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

Topic **Page**

6.1 Overview

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

The C54x DMA is not exactly the same across different 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, please refer to Table 1–9 *Device-specific Features Support*.

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) Primary Functions

Function	Purpose	See page ...
DMA_open()	Opens a DMA channel	6-16
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_start()	Starts a DMA channel	6-18
DMA_stop()	Disables a DMA channel	6-18
DMA_close()	Closes a DMA channel	6-7
DMA_reset()	Resets DMA channel register to their power-on reset value	6-17
DMA_pause()	Pauses a DMA channel. Identical to DMA_stop().	6-18

(b) DMA Global Register Function

DMA_globalAlloc()	Allocates a global DMA register	6-10
DMA_globalConfig()	Sets up the DMA channel using the configuration structure	6-7
DMA_globalConfigArgs()	Sets up the DMA channel using the register values passed in	6-8
DMA_globalFree()	Frees a global DMA register that was previously allocated	6-15
DMA_resetGbl()	Resets the DMA global register	6-17

(c) Auxiliary Functions

DMA_getEventId()	Returns the IRQ Event ID for the DMA completion interrupt	6-10
DMA_getStatus()	Get DMA channel status	6-19
DMA_getChan()	Returns channel number used in given handle	6-17
DMA_getConfig()	Get DMA channel configuration	6-17
DMA_globalGetConfig()	Get DMA global register configuration	6-13

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
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 1.7 *Device-Specific Features Support.*

Description

This is the DMA configuration structure used to set up a DMA channel. You create and initialize this structure then pass its address to the DMA_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
Uint16 dmsrcp	global program page for src
Uint16 dmdstp	global program page for dst
Uint16 dmidx0	global element index 0
Uint16 dmfri0	global frame index 0
Uint16 dmidx1	global element index 1
Uint16 dmfri1	global frame index 1

For devices offering global channel reload registers (see note)*, add:*

DMA_AdrPtr dmgsa	global src address reload
DMA_AdrPtr dmgda	global dst address reload
Uint16 dmgcr	global element count reload
Uint16 dmgfr	global frame count reload

For devices supporting global extended data memory addressing (see note), *add:*

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

Note: For more information concerning these devices, see section 1.7, *Device-specific Features Support.*

Description

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

Example

```
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    */
(DMA_AdrPtr) 100,    /* src data page    */
(DMA_AdrPtr) 101     /* dst data page    */
}
```

For a complete example, see Example 2 in section 6.5.

6.3 Functions

This section describes the functions in the DMA CSL module.

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, you must 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 6.5, *Examples*.

DMA_configArgs *Sets up DMA channel with register values*

Function

```
void DMA_configArgs(
    DMA_Handle hDma,
    Uint16 priority,
    Uint16 dmmcr,
    Uint16 dmsfc,
    DMA_AdrPtr dmsrc,
    DMA_AdrPtr dmdst,
    Uint16 dmctr,
```

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

```
    DMA_AdrPtr dmgsa,
    DMA_AdrPtr 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 1.7, *Device-specific Features Support*

Arguments

hDma	Handle to DMA channel; see DMA_open()
priority	DMA channel priority
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_getChan *Returns Channel number used in given handle*

Function

```
Uint16 DMA_getChan(
    DMA_Handle hDma
);
```

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

Return Value Channel number

Description Get channel number used by a specific handle.

Example

```
Uint16 chanNum;
chanNum = DMA_getChan(hDma);
```

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 6.5, Example 2.

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 be 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 (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 you 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 */
      (DMA_AdrPtr) 100,          /* src data page */
      (DMA_AdrPtr) 101,          /* dst data page */
       }

.....

      mask = DMA_globalAlloc (DMA_GBL_DMIDX1|DMA_GBL_DMFRI0);
      DMA_globalConfig (mask, &MyGblConfig);
```

For a complete example, see Section 6.5, 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 considered 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_DMFRI0);
      DMA_globalConfig (mask, &MyGblConfig);
```

For a complete example, see Section 6.5, Example 2.

DMA_globalGetConfig

Gets 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

Gets 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_globalConfigArgs

Sets up DMA global registers using arguments

Function

void DMA_globalConfigArgs(
  Uint16 RegMask,
  Uint16 free
  Uint16 dmidx0,
  Uint16 dmfri0,
  Uint16 dmidx1,
  Uint16 dmfri1,

For devices supporting global channel reload registers,(see section 1.7) *add:*
  DMA_AdrPtr dmgsa,
  DMA_AdrPtr gbldmgda,
  Uint16 dmgc,
  Uint16 dmgfr,

For all devices, add:
  Uint16 dmsrcp,
  Uint16 dmdstp,

For devices supporting extended DMA data support, (see Section 1.7) *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.

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 1.7)

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 1.7)

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

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_IDX0,);
   ...
   /* some time later on when you're done with it */
   DMA_globalFree(RegMask);
```

DMA_open *Opens DMA channel*

Function

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

Arguments

ChaNum DMA channel to open:
- DMA_CHAANY
- DMA_CHA0
- DMA_CHA1
- DMA_CHA2
- DMA_CHA3
- 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_CHAANY to allow the library pick an unused channel; you can see which channel has been allocated by calling DMA_getChan().

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_CHAANY,DMA_OPEN_RESET);
```

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_resetGbl — *Resets 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_reset(hDma);

/* or */
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 to 1, the enable channel bits in the DMA priority and enable control register (DMPREC) accordingly. 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 (ϕ) the enable channel bits in the DMA priority and enable control (DMPREC) register accordingly. See DMA_start().

Example

```
DMA_stop(hDma);
```

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_getStatus — *Get DMA channel status*

Function

```
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)

DMA_getConfig — *Get 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);
```

6.4 Macros

As covered in section 1.5, 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_RGET_H()
DMA_RSET_H()

(b) Macros to read/write DMA register field values(Applicable only to registers with more than 1-field)
DMA_FGET_H()
DMA_FSET_H()

(c) Macros to create value to write to a DMA register and fields (Applicable only to registers with more than 1-field)
DMA_REG_RMK_H()
DMA_FMK_H()

(d) Macros to read a register address
DMA_ADDR_H()

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
  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(DMSRCDP, 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.

- 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 1.4.

Example

```
Uint16 myregval;
myregval = DMA_DMSFC_RMK (0,0,3); /* dsyn,dblw,frame-
count fields */
```

or you can use the PER_REG_FIELD_SYMVAL symbolic constants provided in CSL (see section 1.4).

```
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 6.5).

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.
DMPREC
DMSFC
DMMCR

FIELD Symbolic name for field of register REG Possible values: Field names as listed in the C54x Register Reference Guide. (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.
- DMPREC
- DMSFC
- DMMCR

FIELD — Symbolic name for field of register REG Possible values: Field names as listed in the C54x Register Reference Guide. (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, INT0SEL);
```

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.
- DMPREC
- DMSFC#
- DMMCR#

FIELD — Symbolic name for field of register REG Possible values: Field names as listed in the C54x Register Reference Guide. (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
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, NT0SEL, 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_RGET_H *Get value of DMA register used in handle*

Macro Uint16 DMA_RGET_H (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_RGET_H (myHandle, DMMCR);
```

DMA_RSET_H — *Set value of DMA register*

Macro void DMA_RSET_H (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_RSET_H (myHandle, DMMCR, 0x123);
```

DMA_FGET_H — *Get value of register field*

Macro Uint16 DMA_FGET_H (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 register 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 (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_FGET_H (myHandle, DMMCR, CTMOD);
```

DMA_FSET_H — *Set value of register field*

Macro void DMA_FSET_H (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 register 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 (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_n ❑ value exceeds the number of bits allowed for that field, ❑ fieldval_n 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_FSET_H (myHandle, DMMCR, CTMOD, 1);
```

DMA_ADDR_H — *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_ADDR_H (myHandle, DMMCR);
```

6.5 Examples

The following CSL DMA initialization examples are provided under the \examples\dma directory.

- 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 6–1, on page 6-33.

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 are 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 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 6–1. DMA Channel Initialization Using DMA_config()

```
#include <csl_dma.h>   // include csl_PER.h required

#define N  16

#pragma DATA_SECTION(src,”table1”)    //table1 to be located
                                        at 0x3000@ds
Uint16 src[N] = {
;...
};

#pragma DATA_SECTION(dst, ”table2”)     // table2 to be
                                         located at 0x2000@ds
Uint16 dst[N];
```

```
DMA_Config  myconfig = {
    0x0             /* low priority channel */
    DMA_DMMCR_RMK(DMA_DMMCR_AUTOINIT_OFF,
                 DMA_DMMCR_DINM_OFF,
                 DMA_DMMCR_IMOD_FULL_ONLY,
                 DMA_DMMCR_CTMOD_MULTIFRAME,
                 DMA_DMMCR_SIND_POSTINC,
                 DMA_DMMCR_DMS_DATA,
                 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 */

    &src[0],                            /* DMSRC */
    &dst[0],                            /* DMDST */
    (Uint16)(N–1)                       /* DMCTR */
};


DMA_Handle myhDma;                    /* define a DMA handle*/

void main(void) {

;...

  CSL_init();                /* Init CSL – REQUIRED!!! */

  myhDma = DMA_open(DMA_CHA0, 0);  /* Open Channel      */
  DMA_config(myhDma, &myconfig);  /* Configure Channel */
  DMA_start(myhDma);                /* Begin Transfer    */
  while(DMA_getStatus(myhDma));    /* Wait for complete */
;...

  DMA_close(myhDma);
}
```

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 setup 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 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 setup 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)
EBUS_FSET()	Void EBUS_FSET(REG,FIELD, Uint16 fieldval)

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 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 non-multiplexed GPIO pins available in the C54x devices. (C544x devices only)

Currently, there are no functions available for the GPIO module. Macros that allows access to registers have been provided.

8.2 Macros

As covered in Section 1.3, CSL offers a collection of macros to gain individual access to the GPIO ”specific” registers (GPIOCR and GPIOSR) in C544x devices.

Table 8–1 contains a list of macros available for the GPIO module. To use them, include ”csl_gpio.h”.

Table 8–1. 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)
GPIO_FSET()	Void GPIO_FSET(REG,FIELD, Uint16 fieldval)

Where:
REG : include GPIOCR, GPIOSR
FIELD : register field name as specified in Appendix xxx.

- 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).

Chapter 9

HPI Module

Describes macros available for the HPI module.

9.1 Macros

As covered in Section 1.3, 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 (C544x devices only)

(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)
HPI_FSET()	Void HPI_FSET(REG,FIELD, Uint16 fieldval)

Where:
REG : include HPIC, GPIOCR, GPIOSR
FIELD : register field name as specified in Appendix xxx.

- 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 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 interrupts.

Topic **Page**

10.1 Overview

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

- Masking an interrupt in the IMR register.
- Polling for the interrupt status from the IFR register.
- 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 10–2(a) and (b) list the primary and auxiliary IRQ functions. Table 10–2(c) lists the API functions that enable DSP/BIOS dispatcher communication. These functions should be used only when DSP/BIOS is present **and** the DSP/BIOS dispatcher is enabled. Table Table 10–3 lists available interrupts for this feature.

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 API's operate on logical events.

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 function 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.

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       32

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

; ...
extern interrupt myIsr();

; ...
main (){

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

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

}
```

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)	10-12
IRQ_globalRestore()	Restores the status of global interrupt enable/disable (INTM).	10-13
IRQ_setVecs()	Sets the base address of the interrupt vector table.	10-15
IRQ_test()	Polls the interrupt flag in IFR register the specified event.	10-15

(b) Auxiliary Functions

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.	10-8

(c) DSP/BIOS Dispatcher Communication Functions

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_getConfig()	Returns current DSP/BIOS dispatch table entries for the specified event.	10-11

Function	Purpose	See page ...
(c) DSP/BIOS Dispatcher Communication Functions		
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_map()	Maps a logical event to its physical interrupt.	10-13
IRQ_setArg()	Sets the value of the argument for DSP/BIOS dispatch to pass to the interrupt service routine for the specified event.	10-14

Table 10–3. IRQ_EVT_NNNN Event List

(a) IRQ Events

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
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

(a) IRQ Events

Constant	Purpose
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
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

(a) IRQ Events

Constant	Purpose
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

10.2 Configuration Structure

IRQ_Config — *IRQ configuration structure*

Structure IRQ_Config

Members
IRQ_IsrPtr funcAddr; *Address of interrupt service routine*
Uint32 ierMask; *Interrupt to disable the existing ISR*
Uint32 funcArg; *Argument to pass to ISR when invoked*

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 */
  0x0300, /* ierMask  */
  0x0000, /* funcArg  */
};
```

10.3 Functions

This sections describes the 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_get XXX EventId() 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_configArgs — *Updates entry in DSPBIOS dispatch table*

Function

```
void IRQ_configArgs(
    Uint16 EventId,
    IRQ_IsrPtr funcAddr,
    Uint32 funcArg,
    Uint16 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.

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

Example

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

IRQ_disable — *Disables specified event*

Function

```
void 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_get XXX EventId() function to get the EventID.

Return Value None

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

Example

```
IRQ_disable(IRQ_EVT_TINT0);
```

IRQ_enable

Enables specified event

Function

void 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_get XXX EventId() function to get the EventID.

Return Value

None

Description

Enables the specified event.

Example

```
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_get XXX EventId() function to get the EventID.

Return Value

Value of argument

Description

Returns value for specified event.

Example

```
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_get XXX EventId() function to get the EventID.

Config   Pointer to configuration structure

Return Value

None

Description

Returns current values in DSP/BIOS dispatch table entry for the specified event.

Example

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

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

IRQ_globalEnable *Globally Enables Interrupts*

Function

int IRQ_globalEnable(
);

Arguments

None

Return Value

intm Returns the old INTM value

Description

This function globally Enables 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

None

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_get XXX EventId() 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

int 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_get XXX EventId() 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

0 or 1

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

```
void  MyIsr ();
.
.
.
IRQ_plug (IRQ_EVT_TINT0, &myIsr)
```

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_get XXX EventId() 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

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

IRQ_setVecs

Sets the base address of the interrupt vectors

Function

```
void IRQ_setVecs(
    Uint32 IVPD
    Uint32      val  Uint32 IVPH
);
```

Arguments

vecs IVPD pointer to the DSP interrupt vector table

Return Value

oldVecs Returns IVPH Pointer to the Host interrupt Vector table

Description

Use this function to set the base address of the interrupt vector table in the IVPD and IVPH registers.

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 ((void*) 0x8000);
```

IRQ_test

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

Function

```
Bool 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_get XXX EventId() 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 McBSP is a handle-based module that requires you to call MCBSP_open() to obtain a handle before calling any other functions.

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_open()	Opens a McBSP port	11-16
MCBSP_config()	Sets up the McBSP port using the configuration structure	11-12
MCBSP_configArgs()	Sets up the McBSP port using the register values passed in	11-13
MCBSP_start()	Start a transmit and/or receive for a MCBSP port	11-19
MCBSP_close()	Closes a McBSP port	11-11

(b) Channel Control Functions

MCBSP_channelDisable()	Disables one or several McBSP channels	11-6
MCBSP_channelEnable()	Enables one or several McBSP channels of the selected register	11-8
MCBSP_channelStatus()	Returns the channel status	11-10

(c) Interrupt Control Functions

MCBSP_getRcvEventId()	Retrieves the receive event ID for the given port	11-15
MCBSP_getXmtEventId()	Retrieves the transmit event ID for the given port	11-15

Table 11–2. McBSP Functions

Function	Purpose	See page ...
(d) Auxiliary Functions		
MCBSP_read16()	Performs a direct 16-bit read to the data receive register DRR1	11-16
MCBSP_write16()	Writes a 16-bit value to the serial port data transmit register, DXR1	11-20
MCBSP_read32()	Performs two direct 16-bit reads: data receive register 2 DRR2 (MSB) and data receive register 1 DRR1 (LSB)	11-17
MCBSP_write32()	Writes two 16-bit values to the two serial port data transmit registers, DXR2 (16-bit MSB) and DXR1 (16-bit LSB)	11-20
MCBSP_reset()	Resets the given serial port	11-26
MCBSP_rfull()	Reads the RFULL bit SPCR1 register	11-17
MCBSP_rrdy()	Reads the RRDY status bit of the SPCR1 register	11-18
MCBSP_xempty()	Reads the XEMPTY bit from the SPCR2 register	11-21
MCBSP_xrdy()	Reads the XRDY status bit of the SPCR2 register	11-21
MCBSP_getConfig()	Get MCBSP channel configuration	11–22
MCBSP_getPort()	Get MCBSP Port number used in given handle	11–22

11.2 Configuration Structure

This section lists the structure in 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

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 */

};
...
MCBSP_config(hMcbsp,&MyConfig);
```

11.3 Functions

This section lists the functions in the McBSP module.

MCBSP_channelDisable

Disables one or several McBSP channels

Function

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

Arguments

hMcbsp — Handle to McBSP port obtained by MCBSP_open()

RegAddr — Receive and Transmit Channel Enable Registers:

- MCBSP_RCERA
- MCBSP_RCERB
- MCBSP_XCERA
- MCBSP_XCERB

For devices supporting 128 channels (see Section 1.7)

- MCBSP_RCERC
- MCBSP_RCERD
- MCBSP_RCERE
- MCBSP_RCERF
- MCBSP_RCERG
- MCBSP_RCERH
- MCBSP_XCERC
- MCBSP_XCERD
- MCBSP_XCERE
- MCBSP_XCERF
- MCBSP_XCERG
- MCBSP_XCERH

Channels — Available values for the specific RegAddr 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.

This function does not check to see if valid data has been received. Use MCBSP_rrdy() for this purpose.

Example

```
/* Disables Channel 0 of the partition A */
MCBSP_channelDisable(hMcbsp,MCBSP_RCERA,(MCBSP_CHAN0);
/* Disables Channels  1, 2 and 8 of the partition B with “|”*/
MCBSP_channelDisable(hMcbsp,MCBSP_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 RegAddr,
    Uint16 Channels
    );
```

Arguments

hMcbsp Handle to McBSP port obtained by MCBSP_open()

RegAddr Receive and Transmit Channel Enable Registers:
- MCBSP_RCERA
- MCBSP_RCERB
- MCBSP_XCERA
- MCBSP_XCERB

For devices supporting 128 channels (See Section 1.7)
- MCBSP_RCERC
- MCBSP_RCERD
- MCBSP_RCERE
- MCBSP_RCERF
- MCBSP_RCERG
- MCBSP_RCERH
- MCBSP_XCERC
- MCBSP_XCERD
- MCBSP_XCERE
- MCBSP_XCERF
- MCBSP_XCERG
- MCBSP_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,MCBSP_RCERA,(MCBSP_CHAN0);
/* Enables Channel 1, 4 and 6 of the partition B  with  “|” */
MCBSP_channelEnable(hMcbsp,MCBSP_RCERB,(MCBSP_CHAN1|
MCBSP_CHAN4 | MCBSP_CHAN6));
```

MCBSP_channelStatus

Returns channel status

Function

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

Arguments

hMcbsp Handle to McBSP port obtained by MCBSP_open()

RegAddr Receive and Transmit Channel Enable Registers:
- MCBSP_RCERA
- MCBSP_RCERB
- MCBSP_XCERA
- MCBSP_XCERB

For devices supporting 128 channels (See Section 1.7)
- MCBSP_RCERC
- MCBSP_RCERD
- MCBSP_RCERE
- MCBSP_RCERF
- MCBSP_RCERG
- MCBSP_RCERH
- MCBSP_XCERC
- MCBSP_XCERD
- MCBSP_XCERE
- MCBSP_XCERF
- MCBSP_XCERG
- MCBSP_XCERH

Channel Selectable Channels for the specific RegAddr 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 Channel status 0 - Disabled
1 - Enabled

Description Returns the channel status by reading the associated bit into the the selected register (RegAddr). 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,MCBSP_RCERB,MCBSP_CHAN1);
/* Returns Channel Status of the channel  4 of the partition A
*/
C4=MCBSP_channelStatus(hMcbsp,MCBSP_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 hMcbsp port registers. MCBSP_config() initializes the MCBSP port registers, but does not start the MCBSP port.

To start a McBSP port, you must call the MCBSP_start() function (see also MCBSP_configArgs()).

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, /* rcera*/
   0x0000, /* rcerb*/
   0x0000, /* xcera*/
   0x0000, /* xcerb*/
   0x0000  /* pcr  */
};
...
MCBSP_config(hMcbsp,&MyConfig);
```

For complete examples, please 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,
    );
```

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
rcera	Receive channel enable register partition A value
rcerb	Receive channel enable register partition B value
xcera	Transmit channel enable register partition A value
xcerb	Transmit channel enable register partition B value

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. MCBSP_configArgs() initializes the McBSP port registers, but does not start the McBSP port.

To start a McBSP port, you must call the MCBSP_start() function (see also MCBSP_configArgs()).

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_getXmtEventID

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

Receiver 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_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_open *Opens McBSP port*

Function
```
MCBSP_Handle MCBSP_open(
    int devNum,
    Uint32 flags
    );
```

Arguments

devNum McBSP device (port) number:
- MCBSP_DEV0
- MCBSP_DEV1
- MCBSP_DEV2 (except for 5402)
- MCBSP_DEVANY

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_DEV0,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 Performs a direct 16-bit read from the data receive register DRR1.

Example
```
Uint16 Data;
...
Data = MCBSP_read16(hMcbsp);
```

This function doesn’t check if valid data has been received. Use MCBSP_rrdy() for this purpose.

MCBSP_read32

Performs 32-bit data read

Function

Uint16 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.

Example

```
Uint32 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

Bool 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

Bool 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 or 1

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)) {
  ...
}
```

MCBSP_start — *Starts transmit and/or receive operation for a McBSP port*

Function

```
void MCBSP_start(
      MCBSP_Handle hMcbsp,
      Uint16 txRxSelectorstartMask,
      Uint16 SampleRaterateGenDelay
      );
```

Arguments

hMcbsp	Handle to McBSP port obtained by MCBSP_open()
txRxSelector	Start transmit, receive or both:
❑	MCBSP_XMIT_START
❑	MCBSP_RCV_START
❑	MCBSP_XMIT_START | MCBSP_RCV_START
SampleRateGenDelay	Sample rate generates delay. MCBSP logic requires two sample_rate generator clock_periods after grabbing the sample rate generator logic to stabilize. Use this parameter to provide the appropriate delay before starting the MCBSP. A conservative value should be equal to:

$$SampleRateGenDelay = \frac{2\ x\ Sample\_Rate\_Generator\_Clock\_period}{4xC54x\_Instruction\_Cycle}$$

Return Value None

Description Starts a transmit and/or receive operation for a McBSP port.

Example

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, Before writing the value, **this function does not check if the transmitter is ready.** Use MCBSP_xrdy() for this purpose.

Example

```
MCBSP_write16(hMcbsp,0x1234);
```

MCBSP_write32 *Writes a 32-bit data value with overrun protection*

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 Directly writes two 16-bit values to the two serial port data transmit registers, DXR2 (16-bit MSW) and DXR1 (16-bit LSW);Before writing the value, **this function does not check to see if the transmitter is ready**. Use MCBSP_xrdy() for this purpose.

Example

```
MCBSP_write32(hMcbsp,0x12345678);
```

MCBSP_xempty

Reads XEMPTY bit from SPCR2 register

Function

```
Bool 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

```
Bool MCBSP_xrdy(
    MCBSP_Handle hMcbsp
    );
```

Arguments

hMcbsp — Handle to McBSP port obtained by MCBSP_open()

Return Value

XRDY — Returns XRDY status bit of SPCR2.

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);
  ...
}
```

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_DEV), 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));
```

11.4 Macros

As covered in Section 1.5, 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
MCBSP_RGET_H()
MCBSP_RSET_H()

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

Macro
MCBSP_FGET_H()
MCBSP_FSET_H()

(c) Macros to read a register address

Macro
MCBSP_ADDR_H()

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#
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 1

```
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)
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 1 For registers:

```
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 register 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 1.4.

Example 1

```
MCBSP_RCR1_RMK (4,3);   /* frame lenght, word length */

or you can use the PER_REG_FIELD_SYMVAL symbolic constants
provided in CSL (See section 1.4)
```

MCBSP_REG_RMK macros are typically used to initialize a MCBSP configuration structure used for the MCBSP_config() function. For more examples see Section 11.5.

MCBSP_FMK

Creates a registervalue based on individual field values

Macro Uint16 MCBSP_FMK (REG, FIELD, fieldval)

Arguments

REG **Only writable register 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 1

```
Uint16 myregval;
Myregval = MCBSP_FMK (RCR1, RFRLEN1, 1) | MCBSP_FMK (RCR1,
RWDLEN1,2);
```

MCBSP_FGET *Get 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#
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 1

```
Uint16 myVar;
...
myVar = MCBSP_FGET(RCR2,RPHASE);
```

MCBSP_FSET

Set 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)
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 1 For Registers:

```
MCBSP_FSET(RCR2,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 1 For Registers:

```
myVar = MCBSP_ADDR(RCR10); /*get register RCR1 of channel 0 */
```

MCBSP_RGET_H *Get the value of a MCBSP register used in a handle*

Macro Uint16 MCBSP_RGET_H (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 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 1

```
MCBSP_Handle myHandle;
Uint16 myVar;
...
myHandle = MCBSP_open (MCBSP_DEV0, MCBSP_OPEN_RESET);
...
myVar = MCBSP_RGET_H(myHandle, RCR1)
```

MCBSP_RSET_H — *Set the value of a MCBSP register*

Macro Void MCBSP_RSET_H (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 (#). 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 1

```
MCBSP_Handle myHandle;
Uint16 myVar;
...
myHandle = MCBSP_open (MCBSP_DEV0, MCBSP_OPEN_RESET);
...
myVar = MCBSP_FSET_H(myHandle, RCR1, 0x4)
```

MCBSP_FGET_H *Get the value of a register field*

Macro Uint16 MCBSP_FGET_H (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
		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 1

```
MCBSP_Handle myHandle;
Uint16 myVar;
...
myHandle = MCBSP_open (MCBSP_DEV0, MCBSP_OPEN_RESET);
...
myVar = MCBSP_FGET_H(myHandle, RCR2, RPHASE)
```

MCBSP_FSET_H *Set the value of a register field*

Macro Void MCBSP_FSET_H (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 (#).

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 1

```
MCBSP_Handle myHandle;
Uint16 myVar;
...
myHandle = MCBSP_open (MCBSP_DEV0, MCBSP_OPEN_RESET);
...
myVar = MCBSP_FSET_H(myHandle, RCR2, RPHASE,1)
```

MCBSP_ADDR_H *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_RGET_H 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_ADDR(myHandle, RCR1)
```

11.5 Examples

The following CSL MCBSP initialization examples are provided under the \examples\MCBSP directory.

Example 11–1 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.

Also, under the \examples\DMA directory, you will find the following combined DMA and MCBSP examples:

- Example: DMA channel data transfer from/to MCBSP in ABU digital loop-back mode.

Example 11–1. McBSP Port Initialization Using MCBSP_config()

```
#include <csl_mcbsp.h>

static MCBSP_Config ConfigLoopBack32= {
   ....
};


void main(void) {

  MCBSP_Handle mhMcbsp;
  Uint32 xmt, rcv;

  ....
   CSL_init();

   mhMcbsp = MCBSP_open(MCBSP_DEV0, MCBSP_OPEN_RESET);

   MCBSP_config(mhMcbsp, &ConfigLoopBack32);

   MCBSP_start (mhMcbsp,MCBSP_XMIT_START|MCBSP_RCV_START);
   ....

   while (!MCBSP_xrdy(mhMcbsp));
   MCBSP_write32(mhMcbsp,xmt);
   ....
   while (!MCBSP_rrdy(mhMcbsp));
   rcv = MCBSP_read32(mhMcbsp);
   ....
   MCBSP_close(mhMcbsp);
   ...
}
```

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 power consumption of different sections in the C54X device.

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.

Table 12–1. PLL Primary Summary

(a) 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

(a) PLL Functions

Function	Purpose	See page ...
PLL_config()	Configure the PLL with the values provided in a configuration structure.	12-4
PLL_configArgs()	Configure the PLL with the values provided as function arguments.	12-4

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 mode : available values
PLL_MODE_DIV = 0 (divide mode)
PLL_MODE_PLLINT = 2 (pll integer multiplier mode)
PLL_MODE_PLLFRCT = 3 (pll fractional 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.

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

Mode	pllmul	final multiplier
PLL_MODE_DIV	0-14	0.5
	15	0.25
PLL_MODE_PLLINT	0-14	pllmul+1
	15	1
PLL_MODE_PLLFRCT	even	(pllmul+1)/2
	odd	pllmul/4

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 (PLL_Config *pcfg)
PLL_configArgs (Uint16 mode, Uint16 pllmul, Uint16 pllcount);

PLL_config — *Writes 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_config (yConfig);
```

PLL_configArgs — *Writes to PLL using register values passed to function*

Function PLL_configArgs (Uint16 mode, Uint16 pllmul, Uint16 pllcount);

Arguments

Uint16 mode : available values
PLL_MODE_DIV = 0 (divide mode)
PLL_MODE_PLLINT = 2 (pll integer multiplier mode)
PLL_MODE_PLLFRCT = 3 (pll fractional 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.

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; or for readability, you may use the PLL_RMK macros to create the register values based on field values.

Clock out frequency is determined as follows:

Mode	pllmul	final multiplier
PLL_MODE_DIV	0-14	0.5
	15	0.25
PLL_MODE_PLLINT	0-14	pllmul+1
	15	1
PLL_MODE_PLLFRCT	even	(pllmul+1)/2
	odd	pllmul/4

Example

```
PLL_configArgs (PLL_MODE_DIV, 1, 20)
```

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 Using Timer Port Number

(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 indicates the register, xxx xxx.
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 writable field.

regval indicates the value to write in the register (*REG*).
fieldval indicates the value to write in the field (*FIELD*).

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

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 state	13-3

13.2 Functions

This section lists the functions in the PWR module.

PWR_powerDown *Forces DSP to enter power-down state*

Function

void PWR_powerDown (PWR_MODE pwrdMode, PWR_wakeMode wakeMode)

Arguments

mode

pwrdMode:

- ❑ PWR_CPUDOWN: CPU goes idle, but peripherals keep running. This corresponds to the IDLE #1 instruction.
- ❑ PWR_CPUPERDOWN: Both CPU and peripherals power-down. This corresponds to the IDLE #2 instruction.
- ❑ PWR_CPUPERPLLDOWN: CPU, peripherals, and PLL power-down. This corresponds to the IDLE #3 instruction.

wakeMode (Valid for all pwrdModes above)

- ❑ PWR_ENABGIE: Wakes up with an unmasked interrupt and jump to execute the ISR's executed.
- ❑ PWR_DISABGIE: Wakes up with an unmasked interrupt and executes the next following instruction (interrupt is not take).

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_CPUDOWN, PWR_ENABGIE);
```

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 structure for use with the TIMER modules. Table 14–2 lists the functions for use with the TIMER modules.

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_open()	Opens a TIMER device	14-6
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_start()	Starts the TIMER device running	14-7
TIMER_reload()	Reloads the TIMER	14-6
TIMER_stop()	Stops the TIMER device running	14-7
TIMER_reset()	Resets the TIMER device	14-7
TIMER_close()	Closes a previously opened TIMER device	14-4

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, C541, and C5472 devices only:

[Uint 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,
    Uint16 prd);

Arguments

hTimer	Device handle (see TIMER_open()).
tcr	Control register value
prd	Period register value
tim	Timer register value – loaded with PRD and decremented

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_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 in Chapter 10-11).

Example

```
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_DEVANY
- TIMER_DEV0
- TIMER_DEV1

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

Return Value

Device Handle — Device handle

Description

Before a TIMER device can be used, it must first be opened by this function. Once opened, it cannot be opened again until closed (see TIMER_close()).The return value is a unique device handle that are 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 the 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, C5472 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 writeable field.

3) *regval* indicates the value to write in the register (*REG*)

4) *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_RGET_H()	Uint16 TIMER_RGET_H(TIMER_Handle hTimer, *REG*)
TIMER_RSET()	void TIMER_RSET_H(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_FGET_H()	Uint16 TIMER_FGET_H(TIMER_Handle hTimer, *REG*, *FIELD*)
TIMER_FSET_H()	Void TIMER_FSET_H(TIMER_Handle hTimer, *REG*, *FIELD*, *fieldval*)

(c) Macros to read a register address

Macro	Syntax
TIMER_ADDR_H()	Uint16 TIMER_ADDR_H(TIMER_Handle hTimer, *REG*)

Notes:
1) *REG* indicates the register, TCR, PRD, TSCR (C5440, C5441, C5472 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 writable field.
3) *regVal* indicates the value to write in the register (*REG*)
4) *fieldVal* indicates the value to write in the field *(FIELD*)

Chapter 15

WDTIM Module

Lists the structure for use with the WDTIM modules.

Topic **Page**

15.1 Overview

Table 15–1 and Table 15–2 list the configuration structures and functions used with the WDTIM module.

Table 15–1. WDTIM Configuration Structure

Structure	Purpose	See page...
WDTIM_Config	WDTIM configuration structure used to setup a timer device	15-3

Table 15–2. WDTIM Functions

Function	Purpose	See page ...
WDTIM_open	Opens a WDTIM device	15-6
WDTIM_getConfig	Reads the current register values of the timer and stores the result in the configuration structure	
WDTIM_config	Sets up the WDTIM register using the configuration structure	15-4
WDTIM_configArgs	Sets up the WDTIM using the register values passed in	15-5
WDTIM_start	Starts the WDTIM device running	15-7
WDTIM_close	Closes a previously opened WDTIM device	15-4
WDTIM_service	Writes to the watchdog key of the timer	15-6

15.2 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 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(hWdTimer,&MyConfig);
```

15.3 Functions

This section lists the functions in the WDTIM module.

WDTIM_close

Closes previously opened WDTIM device

Function

```
void WDTIM_close(
    WDTIM_Handle hTimer
);
```

Arguments

hTimer   Device handle (see WDTIM_open()).

Return Value

None

Description

Closes a previously opened timer device (see WDTIM_open()).

The Following tasks are Performed:

- The timer IRQ event is disabled and cleared
- The timer registers are set to their default values

Example

```
WDTIM_close(hTimer);
```

WDTIM_config

Sets up WDTIM register using configuration structure

Function

```
void WDTIM_config(
    WDTIM_Handle hTimer,
    WDTIM_Config *Config
);
```

Arguments

hTimer   Device handle (see WDTIM_open()).
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(hTimer,&MyConfig);
```

WDTIM_configArgs

Sets up WDTIM using register values passed in

Function

void WDTIM_configArgs(
   WDTIM_Handle hTimer,
   Uint16 tcr,
   Uint16 prd,);

Arguments

hTimer	Device handle (see WDTIM_open()).
Wdtcr	Control register value
Wdtscr	
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 (tcr) 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 (hTimer,
   0x0010, /* tcr */
   0x1000, /* prd */
);
```

WDTIM_open

Opens WDTIM device

Function

WDTIM_Handle WDTIM_open(
    int DevNum,
    Uint16 Flags
);

Arguments

TimNum	Timer Number:
Flags	Open flags, logical OR of any of the following:
	❑ WDTIM_OPEN_RESET

Return Value

Device Handle   Device handle

Description

Before a WDTIM device can be used, it must first be opened by this function. Once opened, it cannot be opened again until closed (see WDTIM_close()). The return value is a unique device handle that are used in subsequent WDTIM API calls. If the open fails, INV (–1) is returned.

If the WDTIM_OPEN_RESET is specified, the timer device registers are set to their power-on defaults and any associated interrupts are disabled and cleared.

Example

```
WDTIM_Handle hTimer;
…
hTimer = WDTIM_open(WDTIM_DEV0,0);
```

WDTIM_service

Writes to the watchdog key of the timer

Function

void WDTIM_service(
    WDTIM_Handle hTimer
);

Arguments

hTimer   Device handle (see WDTIM_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

```
WDTIM_servicehTimer);
WDTIM_service(INV);
```

WDTIM_start — *Starts WDTIM device running*

Function

```
void WDTIM_start(
    WDTIM_Handle hTimer
);
```

Arguments

hTimer   Device handle (see WDTIM_open()).

Return Value

None

Description

Starts the timer device running. TSS field =0.

Example

```
WDTIM_start(hTimer);
```

15.4 Macros

CSL offers a collection of macros to access CPU control registers and fields. For additional details (see section 1.5).

Because the WDTIM peripheral typically has two independent timers in the C54x devices, the macros identify the correct timer through either the device number or the handle.

- Table 15–3 lists the WDTIM macros available that use the device number as part of the register name.
- Table 15–4 lists the WDTIM macros available that use a handle.

Both Table 15–3 and Table 15–4 use the following conventions:

To use the WDTIM 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 WDTIM macros, see section 6.4 in the DMA chapter or section 11.4 in the MCBSP chapter.

Table 15–3. 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*, *fieldval*)

(d) Macros to read a register address

Macro	Syntax
WDTIM_ADDR()	Uint16 WDTIM_ADDR(*REG*)

Notes:
1) *REG* indicates the register, TCR, PRD, TSCR (C5440, C5441, C5472 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*)

Table 15–4. WDTIM CSL Macros Using Handle

(a) Macros to read/write WDTIM register values

Macro	Syntax
WDTIM_RGET_H()	Uint16 WDTIM_RGET_H(WDTIM_Handle hTimer, *REG*)
WDTIM_RSET_H()	void WDTIM_RSET_H(WDTIM_Handle hTimer, *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_H()	Uint16 WDTIM_FGET_H(WDTIM_Handle hTimer, *REG*, *FIELD*)
WDTIM_FSET_H()	void WDTIM_FSET_H(WDTIM_Handle hTimer, *REG*, *FIELD*, *fieldval*)

(c) Macros to read a register address

Macro	Syntax
WDTIM_ADDR_H()	Uint16 WDTIM_ADDR_H(WDTIM_Handle hTimer, *REG*)

Notes:

1) *REG* indicates the register WDTCR, WDTSCR, WDPRD, or WDTIM.
WDTIM is not supported in _RSET/_RSET_H/_FSET/_FSET_H/_RMK macros because these macros apply only to writable registers (WDTIM is read-only)

2) *FIELD* indicates the register field name as specified in Appendix A.
   - For *REG*_FSET_H and *REG*__FMK_H, *FIELD* must be a writable field.
   - For *REG*_FGET_H, 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*)

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)

15	14	13–8	7–6	5–0
FREE	AUTOIX†	DPRC	INTOSEL	DE
R/W–0	R/W–0	R/W–0	R/W–0	R/W–0

† Only available on specific devices.

Note: 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 C5402, C5409, C5409A, C5440, C5441, and C5472:
		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
			11	Reserved

Table A–1. DMA Channel Priority and Enable Control Register (DMPREC) Field Values (DMA_DMPREC_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
	INTOSEL			**For C5410, C5416, C5420, and C5421:**
		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)

15–12	11	10–8	7–0
DSYN	DBLW	reserved	FRAMECNT
R/W-0	R/W-0	R/W-0	R/W-0

Note: 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 C5402:
		NONE	0000	No sync event (nonsynchronization operation)
		REVT0	0001	McBSP 0 receive event (REVT0)
		XEVT0	0010	McBSP 0 transmit event (XEVT0)
			0011	Reserved
			0100	Reserved

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
	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 C5472:
		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)
		REVTA2	1001	McBSP 2 receive event — ABIS mode (REVTA2)
		XEVTA2	1010	McBSP 2 transmit event — ABIS mode (XEVTA2)

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
	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

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
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)

15	14	13	12	11	10–8	7–6	5	4–2	1–0
AUTOINIT	DINM	IMOD	CTMOD	SLAXS†	SIND	DMS	DLAXS†	DIND	DMD
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

† Only available on specific devices with DMA extended data memory.

Note: 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.

Table A–3. DMA Channel n Transfer Mode Control Register (DMMCRn) Field Values (DMA_DMMCR_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
	IMOD			**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
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

Table A–3. DMA Channel n Transfer Mode Control Register (DMMCRn) Field Values (DMA_DMMCR_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
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)

15 … 0
Source Address (SRC)
R/W-0

Note: 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)

15 – 0
Global Source Address (GSA)
R/W-0

Note: 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)

15 – 7	6 – 0
reserved	Source Program Page Address (PAGE)
R/W-0	R/W-0

Note: 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)

15 — 0
Destination Address (DST)
R/W-0

Note: 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)

15 — 0
Global Destination Address (GDA)
R/W-0

Note: 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)

15 – 7	6 – 0
reserved	Destination Program Page Address (PAGE)
R/W-0	R/W-0

Note: 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)

15 – 0
Element Count (ELECNT)
R/W-0

Note: 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)

15–0
Element Count (ELECNT)
R/W-0

Note: 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)

15–8	7–0
reserved	Frame Count (FRAMECNT)
R/W-0	R/W-0

Note: 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)

15	0
Element Index (ELEIDX)	
R/W-0	

Note: 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)

15	0
Element Index (ELEIDX)	
R/W-0	

Note: 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)

15 … 0
Frame Index (FRAMEIDX)
R/W-0

Note: 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)

15 … 0
Frame Index (FRAMEIDX)
R/W-0

Note: 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)

15 – 7	6 – 0
reserved	Extended Source Data Page (DMSRCDP)
R/W-0	R/W-0

Note: 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)

15 – 7	6 – 0
reserved	Extended Destination Data Page (DMDSTDP)
R/W-0	R/W-0

Note: 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 Multichannel BSP (McBSP) Registers

A.2.1 McBSP Serial Port Control Register (SPCR1)

Figure A–19. McBSP Serial Port Control Register 1 (SPCR1)

15	14–13	12–11	10–8
DLB	RJUST	CLKSTP	reserved
R/W-0	R/W-0	R/W-0	R/W-0

7	6	5–4	3	2	1	0
DXENA	ABIS†	RINTM	RSYNCERR	RFULL	RRDY	RRST
R/W-0	R/W-0	R/W-0	R/W-0	R-0	R-0	R/W-0

† Only available on specific devices.

Note: R/W-x = Read/Write-Reset value

Table A–19. 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.

Table A–19. McBSP Serial Port Control Register 1 (SPCR1) Field Values (MCBSP_SPCR1_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
10–8	reserved			Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
7	DXENA			DX enabler bit.
		OFF	0	DX enabler is off.
		ON	1	DX enabler is on.
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.2.2 McBSP Serial Port Control Register 2 (SPCR2)

Figure A–20. McBSP Serial Port Control Register 2 (SPCR2)

15–10	9	8
reserved	FREE	SOFT
R/W-0	R/W-0	R/W-0

7	6	5–4	3	2	1	0
FRST	GRST	XINTM	XSYNCERR†	XEMPTY	XRDY	XRST
R/W-0	R/W-0	R/W-0	R/W-0	R-0	R-0	R/W-0

† 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.

Note: R/W-x = Read/Write-Reset value

Table A–20. 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–20. 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.2.3 McBSP Pin Control Register (PCR)

Figure A–21. 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†	CLKS_STAT	DX_STAT	DR_STAT	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.

Note: R/W-x = Read/Write-Reset value

Table A–21. 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–21. 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–21. 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.2.4 Receive Control Register 1 (RCR1)

Figure A–22. Receive Control Register 1 (RCR1)

15	14 … 8	7 … 5	4 … 0
reserved	RFRLEN1	RWDLEN1	reserved
R/W-0	R/W-0	R/W-0	R/W-0

Note: R/W-x = Read/Write-Reset value

Table A–22. 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.2.5 Receive Control Register 2 (RCR2)

Figure A–23. Receive Control Register 2 (RCR2)

15	14–8	7–5	4–3	2	1–0
RPHASE	RFRLEN2	RWDLEN2	RCOMPAND	RFIG	RDATDLY
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

Note: R/W-x = Read/Write-Reset value

Table A–23. 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.
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.

Table A–23. Receive Control Register 2 (RCR2) Field Values (MCBSP_RCR2_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
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.2.6 Transmit Control Register 1 (XCR1)

Figure A–24. Transmit Control Register 1 (XCR1)

15	14–8	7–5	4–0
reserved	XFRLEN1	XWDLEN1	reserved
R/W-0	R/W-0	R/W-0	R/W-0

Note: R/W-x = Read/Write-Reset value

Table A–24. 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.
		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.2.7 Transmit Control Register 2 (XCR2)

Figure A–25. Transmit Control Register 2 (XCR2)

15	14–8	7–5	4–3	2	1–0
XPHASE	XFRLEN2	XWDLEN2	XCOMPAND	XFIG	XDATDLY
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

Note: R/W-x = Read/Write-Reset value

Table A–25. 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.
		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.

Table A–25. Transmit Control Register 2 (XCR2) Field Values (MCBSP_XCR2_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
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.2.8 Sample Rate Generator Register 1 (SRGR1)

Figure A–26. Sample Rate Generator Register 1 (SRGR1)

15–8	7–0
FWID	CLKGDV
R/W-0	R/W-0000 0001b

Note: R/W-x = Read/Write-Reset value

Table A–26. 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.2.9 Sample Rate Generator Register 2 (SRGR2)

Figure A–27. Sample Rate Generator Register 2 (SRGR2)

15	14	13	12	11–0
GSYNC	CLKSP	CLKSM	FSGM	FPER
R/W-0	R/W-0	R/W-1	R/W-0	R/W-0

Note: R/W-x = Read/Write-Reset value

Table A–27. 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.
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.2.10 Multichannel Control Register 1 (MCR1)

Figure A–28. Multichannel Control Register 1 (MCR1)

15–10	9	8–7	6–5	4–2	1	0
reserved	RMCME†	RPBBLK	RPABLK	RCBLK	reserved	RMCM
R/W-0	R/W-0	R/W-0	R/W-0	R-0	R/W-0	R/W-0

† Only available on specific devices that provide 128-channel selection capability.

Note: R/W-x = Read/Write-Reset value

Table A–28. 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
		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

Table A–28. Multichannel Control Register 1 (MCR1) Field Values (MCBSP_MCR1_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
	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.2.11 Multichannel Control Register 2 (MCR2)

Figure A–29. Multichannel Control Register 2 (MCR2)

15–10	9	8–7	6–5	4–2	1–0
reserved	XMCME†	XPBBLK	XPABLK	XCBLK	XMCM
R/W-0	R/W-0	R/W-0	R/W-0	R-0	R/W-0

† Only available on specific devices that provide 128-channel selection capability.

Note: R/W-x = Read/Write-Reset value

Table A–29. 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

Table A–29. Multichannel Control Register 2 (MCR2) Field Values (MCBSP_MCR2_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
	XPBBLK	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
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.2.12 Receive Channel Enable Register (RCERn)

Figure A–30. 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

Note: R/W-x = Read/Write-Reset value

Table A–30. 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.
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.

Table A–30. Receive Channel Enable Register (RCERn) Field Values (MCBSP_RCERn_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
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.

A.2.13 Transmit Channel Enable Register (XCERn)

Figure A–31. 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

Note: R/W-x = Read/Write-Reset value

Table A–31. 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.

Table A–31. Transmit Channel Enable Register (XCERn) Field Values (MCBSP_XCERn_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
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.
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.

A.3 Clock Mode Register (CLKMD)

Figure A–32. Clock Mode Register (CLKMD)

15–12	11	10–3	2	1	0
PLLMUL	PLLDIV	PLLCOUNT	PLLON/OFF	PLLNDIV	PLLSTATUS†
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-0

† When in DIV mode (PLLSTATUS is low), PLLMUL, PLLDIV, PLLCOUNT, and PLLON/OFF are don’t cares, and their contents are indeterminate.

Note: R/W-x = Read/Write-Reset value

Table A–32. 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.
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.

Table A–32. Clock Mode Register (CLKMD) Field Values (PLL_CLKMD_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
0	PLLSTATUS			This read-only bit indicates the mode that the clock generator is operating.
			0	Divider (DIV) mode
			1	PLL mode

A.4 Timer Registers

A.4.1 Timer Control Register (TCR)

Figure A–33. Timer Control Register (TCR)

15–12	11	10	9–6	5	4	3–0
reserved	SOFT	FREE	PSC	TRB	TSS	TDDR
R/W-0	R/W-0	R/W-0	R-0	R/W-0	R/W-0	R/W-0

Note: R/W-x = Read/Write-Reset value

Table A–33. 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).
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.

Table A–33. *Timer Control Register (TCR) Field Values (TIMER_TCR_field_symval) (Continued)*

Bit	*field*	*symval*	Value	Description
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.4.2 Timer Secondary Control Register (TSCR)

Figure A–34. Timer Secondary Control Register (TSCR) — C5440, C5441, and C5472

15–13	12	11–0
reserved	PREMD	reserved
R/W-0	R/W-0	R/W-0

Note: R/W-x = Read/Write-Reset value

Table A–34. 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.5 Watchdog Timer Registers (C5440 and C5441)

A.5.1 Watchdog Timer Control Register (WDTCR)

Figure A–35. Watchdog Timer Control Register (WDTCR)

15–12	11	10	9–6	5–4	3–0
reserved	SOFT	FREE	PSC	reserved	TDDR
R/W-0	R/W-0	R/W-0	R-0	R/W-0	R/W-1111b

Note: R/W-x = Read/Write-Reset value

Table A–35. 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.
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.

Table A–35. Watchdog Timer Control Register (WDTCR) Field Values (WDTIM_WDTCR_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
	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.5.2 Watchdog Timer Secondary Control Register (WDTSCR)

Figure A–36. Watchdog Timer Secondary Control Register (WDTSCR)

15	14	13	12	11–0
WDFLAG	WDEN	reserved	PREMD	WDKEY
R/W-0	R/W-0	R/W-0	R/W-1	R/W-0

Note: R/W-x = Read/Write-Reset value

Table A–36. 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.6 Software Wait-State Registers

A.6.1 Software Wait-State Register (SWWSR)

Figure A–37. Software Wait-State Register (SWWSR)

15	14–12	11–9	8–6	5–3	2–0
XPA†	IO	DATAHI	DATALO	PROGHI	PROGLO
R/W-0	R/W-111b	R/W-111b	R/W-111b	R/W-111b	R/W-111b

† XPA bit only on selected devices with extended program memory.

Note: R/W-x = Read/Write-Reset value

Table A–37. 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.6.2 Software Wait-State Control Register (SWCR)

Figure A–38. Software Wait-State Control Register (SWCR)

15 ... 1	0
reserved	SWSM
R/W-0	R/W-0

Note: R/W-x = Read/Write-Reset value

Table A–38. 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.7 Bank-Switching Control Register (BSCR)

Figure A–39. Bank-Switching Control Register (BSCR) — C5402, C5409, and C5420

15–12	11	10–9	8	7–3	2	1	0
BNKCMP	PSDS	reserved	IPIRQ†	reserved	HBH†	BH	EXIO
R/W-1111b	R/W-1	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

† HBH and IPIRQ bits only on selected devices.

Note: R/W-x = Read/Write-Reset value

Table A–39. Bank-Switching Control Register (BSCR) Field Values — C5402, C5409, and C5420 (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.

Table A–39. Bank-Switching Control Register (BSCR) Field Values — C5402, C5409, and C5420 (EBUS_BSCR_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
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				**For C5402 and C5409:**
	HBH			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.
				For C5420:
	BH			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 C5402 and C5409:** 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			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–40. Bank-Switching Control Register (BSCR) — C5410, C5410A, and C5416

15	14–13	12	11–3	2	1	0
CONSEC	DIVFCT	IACKOFF	reserved	HBH†	BH†	reserved
R/W-1	R/W-11b	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

† BH and HBH bits only on selected devices.

Note: R/W-x = Read/Write-Reset value

Table A–40. Bank-Switching Control Register (BSCR) Field Values — C5410, C5410A, 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.

Table A–40. Bank-Switching Control Register (BSCR) Field Values — C5410, C5410A, and C5416 (EBUS_BSCR_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
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–41. Bank-Switching Control Register (BSCR) — C5440 and C5441

15–3	2	1	0
reserved	BHD	BHA	reserved
R/W-0	R/W-0	R/W-0	R/W-0

Note: R/W-x = Read/Write-Reset value

Table A–41. 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.8 General Purpose I/O Registers

A.8.1 General Purpose I/O Control Register (GPIOCR)

Figure A–42. General Purpose I/O Control Register (GPIOCR)

15	14–8	7	6	5	4	3	2	1	0
TOUT1†	reserved	DIR7	DIR6	DIR5	DIR4	DIR3	DIR2	DIR1	DIR0
R/W-0	R/W-0	R/W–0	R/W–0	R/W–0	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.

Note: R/W-x = Read/Write-Reset value

Table A–42. 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	DIR7			I/O pin 7 direction bit configures the HD7 pin as input or output.
			0	The HD7 pin is configured as an input.
		MASK	1	The HD7 pin is configured as an output. When the HPI-8 is enabled, this bit is forced to 0 and is not affected by writes.
6	DIR6			I/O pin 6 direction bit configures the HD6 pin as input or output.
			0	The HD6 pin is configured as an input.
		MASK	1	The HD6 pin is configured as an output. When the HPI-8 is enabled, this bit is forced to 0 and is not affected by writes.
5	DIR5			I/O pin 5 direction bit configures the HD5 pin as input or output.
			0	The HD5 pin is configured as an input.
		MASK	1	The HD5 pin is configured as an output. When the HPI-8 is enabled, this bit is forced to 0 and is not affected by writes.
4	DIR4			I/O pin 4 direction bit configures the HD4 pin as input or output.
			0	The HD4 pin is configured as an input.
		MASK	1	The HD4 pin is configured as an output. When the HPI-8 is enabled, this bit is forced to 0 and is not affected by writes.

Table A–42. General Purpose I/O Control Register (GPIOCR) Field Values (HPI_GPIOCR_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
3	DIR3			I/O pin 3 direction bit configures the HD3 pin as input or output.
			0	The HD3 pin is configured as an input.
		MASK	1	The HD3 pin is configured as an output. When the HPI-8 is enabled, this bit is forced to 0 and is not affected by writes.
2	DIR2			I/O pin 2 direction bit configures the HD2 pin as input or output.
			0	The HD2 pin is configured as an input.
		MASK	1	The HD2 pin is configured as an output. When the HPI-8 is enabled, this bit is forced to 0 and is not affected by writes.
1	DIR1			I/O pin 1 direction bit configures the HD1 pin as input or output.
			0	The HD1 pin is configured as an input.
		MASK	1	The HD1 pin is configured as an output. When the HPI-8 is enabled, this bit is forced to 0 and is not affected by writes.
0	DIR0			I/O pin 0 direction bit configures the HD0 pin as input or output.
			0	The HD0 pin is configured as an input.
		MASK	1	The HD0 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.8.2 General Purpose I/O Status Register (GPIOSR)

Figure A–43. General Purpose I/O Status Register (GPIOSR)

15–8	7	6	5	4	3	2	1	0
reserved	IO7	IO6	IO5	IO4	IO3	IO2	IO1	IO0
R/W-0	R/W–x	R/W–x	R/W–x	R/W–x	R/W–x	R/W–x	R/W–x	R/W–x

Note: R/W-x = Read/Write-Reset value

Table A–43. General Purpose I/O Status Register (GPIOSR) Field Values (HPI_GPIOSR_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	IO7			I/O pin 7 status bit reflects the logic level on the HD7 pin. When the HD7 pin is configured as an input (DIR7 = 0 in GPIOCR), the IO7 bit latches the logic value (1 or 0) of the HD7 pin. Writes to the IO7 bit have no effect when the HD7 pin is configured as an input. When the HD7 pin is configured as an output (DIR7 = 1 in GPIOCR), the HD7 pin is driven to the logic level (1 or 0) written in the IO7 bit.
			0	The HD7 input is externally driven low, or the HD7 output is internally driven low.
		MASK	1	The HD7 input is externally driven high, or the HD7 output is internally driven high.
6	IO6			I/O pin 6 status bit reflects the logic level on the HD6 pin. When the HD6 pin is configured as an input (DIR6 = 0 in GPIOCR), the IO6 bit latches the logic value (1 or 0) of the HD6 pin. Writes to the IO6 bit have no effect when the HD6 pin is configured as an input. When the HD6 pin is configured as an output (DIR6 = 1 in GPIOCR), the HD6 pin is driven to the logic level (1 or 0) written in the IO6 bit.
			0	The HD6 input is externally driven low, or the HD6 output is internally driven low.
		MASK	1	The HD6 input is externally driven high, or the HD6 output is internally driven high.

Table A–43. General Purpose I/O Status Register (GPIOSR) Field Values (HPI_GPIOSR_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
5	IO5			I/O pin 5 status bit reflects the logic level on the HD5 pin. When the HD5 pin is configured as an input (DIR5 = 0 in GPIOCR), the IO5 bit latches the logic value (1 or 0) of the HD5 pin. Writes to the IO5 bit have no effect when the HD5 pin is configured as an input. When the HD5 pin is configured as an output (DIR5 = 1 in GPIOCR), the HD5 pin is driven to the logic level (1 or 0) written in the IO5 bit.
			0	The HD5 input is externally driven low, or the HD5 output is internally driven low.
		MASK	1	The HD5 input is externally driven high, or the HD5 output is internally driven high.
4	IO4			I/O pin 4 status bit reflects the logic level on the HD4 pin. When the HD4 pin is configured as an input (DIR4 = 0 in GPIOCR), the IO4 bit latches the logic value (1 or 0) of the HD4 pin. Writes to the IO4 bit have no effect when the HD4 pin is configured as an input. When the HD4 pin is configured as an output (DIR4 = 1 in GPIOCR), the HD4 pin is driven to the logic level (1 or 0) written in the IO4 bit.
			0	The HD4 input is externally driven low, or the HD4 output is internally driven low.
		MASK	1	The HD4 input is externally driven high, or the HD4 output is internally driven high.
3	IO3			I/O pin 3 status bit reflects the logic level on the HD3 pin. When the HD3 pin is configured as an input (DIR3 = 0 in GPIOCR), the IO3 bit latches the logic value (1 or 0) of the HD3 pin. Writes to the IO3 bit have no effect when the HD3 pin is configured as an input. When the HD3 pin is configured as an output (DIR3 = 1 in GPIOCR), the HD3 pin is driven to the logic level (1 or 0) written in the IO3 bit.
			0	The HD3 input is externally driven low, or the HD3 output is internally driven low.
		MASK	1	The HD3 input is externally driven high, or the HD3 output is internally driven high.
2	IO2			I/O pin 2 status bit reflects the logic level on the HD2 pin. When the HD2 pin is configured as an input (DIR2 = 0 in GPIOCR), the IO2 bit latches the logic value (1 or 0) of the HD2 pin. Writes to the IO2 bit have no effect when the HD2 pin is configured as an input. When the HD2 pin is configured as an output (DIR2 = 1 in GPIOCR), the HD2 pin is driven to the logic level (1 or 0) written in the IO2 bit.
			0	The HD2 input is externally driven low, or the HD2 output is internally driven low.
		MASK	1	The HD2 input is externally driven high, or the HD2 output is internally driven high.

Table A–43. General Purpose I/O Status Register (GPIOSR) Field Values (HPI_GPIOSR_field_symval) (Continued)

Bit	*field*	*symval*	Value	Description
1	IO1			I/O pin 1 status bit reflects the logic level on the HD1 pin. When the HD1 pin is configured as an input (DIR1 = 0 in GPIOCR), the IO1 bit latches the logic value (1 or 0) of the HD1 pin. Writes to the IO1 bit have no effect when the HD1 pin is configured as an input. When the HD1 pin is configured as an output (DIR1 = 1 in GPIOCR), the HD1 pin is driven to the logic level (1 or 0) written in the IO1 bit.
			0	The HD1 input is externally driven low, or the HD1 output is internally driven low.
		MASK	1	The HD1 input is externally driven high, or the HD1 output is internally driven high.
0	IO0			I/O pin 0 status bit reflects the logic level on the HD0 pin. When the HD0 pin is configured as an input (DIR0 = 0 in GPIOCR), the IO0 bit latches the logic value (1 or 0) of the HD0 pin. Writes to the IO0 bit have no effect when the HD0 pin is configured as an input. When the HD0 pin is configured as an output (DIR0 = 1 in GPIOCR), the HD0 pin is driven to the logic level (1 or 0) written in the IO0 bit.
			0	The HD0 input is externally driven low, or the HD0 output is internally driven low.
		MASK	1	The HD0 input is externally driven high, or the HD0 output is internally driven high.

Index

D

R

S

T