TMS320C55x Chip Support Library API Reference Guide

SPRU433H
May 2004

PRINTED WITH SOY INK

TEXAS INSTRUMENTS

Printed on Recycled Paper

IMPORTANT NOTICE

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

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

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

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

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

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

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

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

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

Preface

Read This First

About This Manual

The TMS320C55x™ DSP Chip Support Library (CSL) provides C-program functions to configure and control on-chip peripherals, which makes it easier for algorithms to run in a real system. The CSL provides peripheral ease of use, shortened development time, portability, and hardware abstraction, along with some level of standardization and compatibility among devices. A version of the CSL is available for all TMS320C55x DSP devices.

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

How to Use This Manual

The contents of the TMS320C5000™ DSP Chip Support Library (CSL) are 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, and shows how to define Build options in the Code Composer Studio™ environment.

Chapters 3-21 provide basic examples, functions, and macros, for the individual CSL modules.

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.
- TMS320C55x™ DSP devices are referred to throughout this reference guide as C5501, C5502, etc.

Related Documentation From Texas Instruments

The following books describe the TMS320C55x™ 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.

TMS320C55x DSP Algebraic Instruction Set Reference Guide (literature number SPRU375) describes the algebraic instructions individually. Also includes a summary of the instruction set,a list of the instruction opcodes, and a cross-reference to the mnemonic instruction set.

TMS320C55x Assembly Language Tools User's Guide (literature number SPRU280) 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 TMS320C55x devices.

TMS320C55x Optimizing C Compiler User's Guide (literature number SPRU281) describes the C55x C Compiler. This C compiler accepts ANSI standard C source code and produces assembly language source code for TMS320C55x devices.

TMS320C55x DSP CPU Reference Guide (literature number SPRU371) describes the architecture, registers, and operation of the CPU for these digital signal processors (DSPs). This book also describes how to make individual portions of the DSP inactive to save power.

TMS320C55x DSP Mnemonic Instruction Set Reference Guide (literature number SPRU374) describes the mnemonic instructions individually. Also includes a summary of the instruction set, a list of the instruction opcodes, and a cross-reference to the algebraic instruction set.

TMS320C55x Programmer's Guide (literature number SPRU376) describes ways to optimize C and assembly code for the TMS320C55x DSPs and explains how to write code that uses special features and instructions of the DSP.

TMS320C55x Technical Overview (SPRU393). This overview is an introduction to the TMS320C55x digital signal processor (DSP). The TMS320C55x is the latest generation of fixed-point DSPs in the TMS320C5000 DSP platform. Like the previous generations, this processor is optimized for high performance and low-power operation. This book describes the CPU architecture, low-power enhancements, and embedded emulation features of the TMS320C55x.

Trademarks

The Texas Instruments logo and Texas Instruments are registered trademarks of Texas Instruments. Trademarks of Texas Instruments include: TI, Code Composer Studio, DSP/BIOS, and the TMS320C5000 family and devices.

All other brand or product names are trademarks or registered trademarks of their respective companies or organizations.

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 collection of functions, macros, and symbols used to configure and control on-chip peripherals. It is a fully scalable component of DSP/BIOS™ and does not require the use of other DSP/BIOS components to operate.

1.1.1 How the CSL Benefits You

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

Standard Protocol to Program Peripherals

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

Basic Resource Management

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

Symbol Peripheral Descriptions

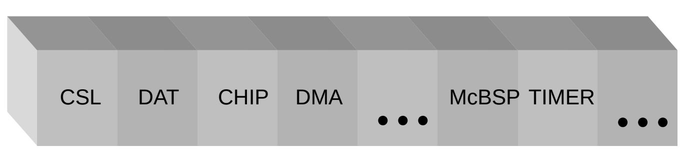
As a side benefit to the creation of the CSL, a complete symbolic description of all peripheral registers and register fields has been created. It is suggested you should use the higher level protocols described in the first two benefits, as these are less device-specific, thus making it easier to migrate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

1.1.2 CSL Architecture

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

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

Figure 1-1. CSL Modules



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

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

Table 1-1. CSL Modules and Include Files

Peripheral Module (PER)	Description	Include File	Module Support Symbol
ADC	Analog-to-digital Converter	csl_adc.h	_ADC_SUPPORT
CHIP	General device module	csl_chip.h	_CHIP_SUPPORT
DAT	A data copy/fill module based on the DMA C55x	csl_dat.h	_DAT_SUPPORT
DMA	DMA Peripheral	csl_dma.h	_DMA_SUPPORT
EMIF	External memory bus interface	csl_emif.h	_EMIF_SUPPORT
GPIO	Non-multiplexed general purpose I/O	csl_gpio.h	_GPIO_SUPPORT
I2C	I^2C peripheral	csl_i2c.h	_I2C_SUPPORT
ICACHE	Instruction Cache	csl_icache.h	_ICACHE_SUPPORT
IRQ	Interrupt controller	csl_irq.h	_IRQ_SUPPORT
McBSP	Multichannel buffered serial port	csl_mcbsp.h	_MCBSP_SUPPORT
MMC	Multimedia Card	csl_mmc.h	_MMC_SUPPORT
PLL	PLL	csl_pll.h	_PLL_SUPPORT
PWR	Power savings control	csl_pwr.h	_PWR_SUPPORT
RTC	Real-time clock	csl_rtc.h	_RTC_SUPPORT
TIMER	Timer peripheral	csl_timer.h	_TIMER_SUPPORT
WDTIM	Watchdog Timer	csl_wdtim.h	_WDT_SUPPORT
USB[†]	USB peripheral	csl_usb.h	_USB_SUPPORT
UART	Universal asynchronous receiver/ transmitter	csl_uart.h	_UART_SUPPORT
HPI	Host port interface	csl_hpi.h	_HPI_SUPPORT
GPT	64-bit General purpose timer	csl_gpt.h	_GPT_SUPPORT

[†] Information and instructions for the configuration of the USB module are found in the *TMS320C55x CSL USB Programmer's Reference Guide* (SPRU511).

Table 1-2 lists the C5000 devices that the CSL supports and the large and small-model libraries included in the CSL. The device support symbol must be used with the compiler (-d option), for the correct peripheral configuration to be used in your code.

Table 1-2. CSL Device Support

Device	Small-Model Library	Large-Model Library	Device Support Symbol
C5502	csl5502.lib	csl5502x.lib	CHIP_5502
C5509	csl5509.lib	csl5509x.lib	CHIP_5509
C5509A	csl5509a.lib	CSL5509ax.lig	CHIP_5509A
C5510PG1.0	csl5510PG1_0.lib	csl5510PG1_0x.lib	CHIP_5510PG1_0
C5510PG1.2	csl5510PG1_2.lib	csl5510PG1_2x.lib	CHIP_5510PG1_2
C5510PG2.0	csl5510PG2_0.lib	csl5510PG2_0x.lib	CHIP_5510PG2_0
C5510PG2.1	csl5510PG2_1.lib	csl5510PG2_1x.lib	CHIP_5510PG2_1
C5510PG2.2	csl5510PG2_2.lib	csl5510PG2_2x.lib	CHIP_5510PG2_2

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 uppercase letters.

Function names use all lowercase letters. Uppercase letters are used only if the function name consists of two separate words. For example, PER_getConfig().

Macro names use all uppercase letters; for example, DMA_DMPREC_RMK.

Data types start with an uppercase letter followed by lowercase letters, e.g., DMA_Handle.

1.3 CSL Data Types

The CSL provides its own set of data types that all begin with an uppercase 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**
CSLBool	unsigned short
*PER*_Handle	void *
Int16	short
Int32	long
Uchar	unsigned char
Uint16	unsigned short
Uint32	unsigned long
DMA_AdrPtr	void (*DMA_AdrPtr)() pointer to a void function

1.4 CSL Functions

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

Note:

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

The following conventions are used and are shown in Table 1-5:

Italics indicate variable names.

Brackets [...] indicate optional parameters.

- *[handle]* is required only for the handle-based peripherals: DAT, DMA, McBSP, and TIMER. See section 1.7.1.
- *[priority]* is required only for the DAT peripheral module.

CSL functions provide a way to program peripherals by:

Direct register initialization using the PER_config() function (see section 1.4.1).

Using functional parameters using the PER_setup() function and various module specific functions (see section 1.4.2). This method provides a higher level of abstraction compared with the direct register initialization method, but typically at the expense of a larger code size and higher cycle count.

Note:

These functions are not available for all CSL peripheral modules.

Table 1-5. Generic CSL Functions

Function	Description
handle = *PER*_**open**(*channelNumber,* *[priority,]* *flags*)	Opens a peripheral channel and then performs the operation indicated by *flags*; must be called before using a channel. The return value is a unique device handle 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. Initialize the configuration structure with: Integer constants Integer variables CSL symbolic constants, *PER_REG*_DEFAULT (See Section 1.6 on page 1-13, CSL *Symbolic Constant Values*) Merged field values created with the *PER_REG*_RMK macro
PER_setup(*[handle,]* *setupStructure)	Initializes the peripheral based on the functional parameters included in the initialization structure. Functional parameters are peripheral specific. This function may not be supported in all peripherals. Please consult the chapter that includes the module for specific details.
*PER*_**start**(*[handle,])* *[txrx,]* *[delay]* *)*	Starts the peripheral after using PER_config(). [txrx] and [delay] apply only to McBSP.
*PER*_**reset**(*[handle]* *)*	Resets the peripheral to its power-on default values.
*PER*_**close**(*handle*)	Closes a peripheral channel previously opened with *PER*_open(). The registers for the channel are set to their power-on defaults, and any pending interrupt is cleared.

1.4.1 Peripheral Initialization via Registers

The CSL provides a generic function, Per_config(), for initializing the registers of a peripheral (*PER* is the peripheral as listed in Table 1-1).

***PER*_config()** allows you to initialize a configuration structure with the appropriate register values and pass the address of that structure to the function, which then writes the values to the writable register. Example 1-1 shows an example of this method. The CSL also provides the PER_REG_RMK (make) macros, which form merged values from a list of field arguments. Macros are covered in section 1.5, *CSL Macros*.

Example 1-1. Using PER_config

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

1.4.2 Peripheral Initialization via Functional Parameters

The CSL also provides functions to initialize peripherals via functional parameters. This method provides a higher level of abstraction compared with the direct register initialization method, which produces larger code size and higher cycle count.

Even though each CSL module may offer different parameter-based functions, PER_setup() is the most commonly used. PER_setup() initializes the parameters in the peripheral that are typically initialized only once in your application. PER_setup() can then be followed by other module functions implementing other common run-time peripheral operations as shown in Example 1-2. Other parameter-based functions include module-specific functions such as the PLL_setFreq() or the ADC_setFreq() functions.

Example 1-2. Using PER_setup()

```
PER_setup mySetup = {param_1, .... param_n};


main() {
   ...
   PER_setup (&mySetup);
   ...
   }
```

Note:

In previous versions of CSL, PER_setup() is referred to as PER_init().

1.5 CSL 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 (with the exception of the DAT module).

REG indicates a register name (without the channel number).

REG# indicates, if applicable, a register with the channel number. (For example: DMAGCR, TCR0, ...)

FIELD indicates a field in a register.

regval indicates an integer constant, an integer variable, a symbolic constant (*PER_REG*_DEFAULT), or a merged field value created with the *PER*_REG_RMK() macro.

fieldval indicates an integer constant, integer variable, macro, or symbolic constant (*PER_REG_FIELD_SYMVAL*) as explained in section 1.6; all field values are right justified.

CSL also offers equivalent macros to those listed in Table 1-6, but instead of using REG# to identify which channel the register belongs to, it uses the Handle value. The Handle value is returned by the PER_open() function. These macros are shown Table 1-7. Please note that REG is the register name *without the channel/port 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.

Table 1-6. Generic CSL Macros (Continued)

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

1.6 CSL Symbolic Constant Values

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

PER indicates a peripheral module as listed in Table 1-1 (with the exception of the DAT module, which does not have its own registers).

REG indicates a peripheral register.

FIELD indicates a field in the register.

SYMVAL indicates the symbolic value of a register field.

Table 1-8. Generic CSL Symbolic Constants

(a) Constant Values for Registers

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

(b) Constant Values for Fields

Constant	Description
PER_REG_FIELD_SYMVAL	Symbolic constant to specify values for individual fields in the specified peripheral register.
*PER_REG_FIELD*_**DEFAULT**	Default value for a field; corresponds to the field value after a reset or to 0 if a reset has no effect.

1.7 Resource Management and the Use of CSL Handles

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

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

When given a specific channel/port number, the open function checks a global flag to determine its availability. If the port/channel is available, 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.

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

Note:

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

1.7.1 Using CSL Handles

CSL Handle objects are used to uniquely identify an opened peripheral channel/port or device. Handle objects must be declared in the C source, and initialized by a call to a PER_open function before calling any other API functions that require a handle object as argument. For example:

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

Once defined, the CSL Handle object is initialized by a call to PER_open:

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

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

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

Chapter 2

How to Use CSL

This chapter provides instructions on how to use the CSL to configure and program peripherals as well as how to compile and link the CSL using Code Composer Studio.

Topic **Page**

2.1 Overview

Peripherals are configured using the CSL by declaring/initializing objects and invoking the CSL functions inside your C source code.

2.2 Using the CSL

This section provides an example of using CSL APIs. There are two ways to program peripherals using the CSL:

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

Functional parameter-based configuration (PER_setup()): Configures peripherals via a set of parameters. Compared to register-based configurations, functional parameter-based configurations require more cycles and code size, but are more abstracted.

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

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

2.2.1 Using the DMA_config() function

The example and steps below use the DMA_config() function to initialize the registers. This example is written for the C5509 device.

Step 1: Include the csl.h and the header file of the module/peripheral you will use <csl_dma.h>. The different header files are shown in Table 1.1.

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

// Example-specific initialization
#define N 16        // block size to transfer
#pragma DATA_SECTION(src,"table1")
/* scr data table address */
```

```
Uint16 src[N] = {
      0xBEEFu,  0xBEEFu,  0xBEEFu,  0xBEEFu,
      0xBEEFu,  0xBEEFu,  0xBEEFu,  0xBEEFu,
      0xBEEFu,  0xBEEFu,  0xBEEFu,  0xBEEFu,
      0xBEEFu,  0xBEEFu,  0xBEEFu,  0xBEEFu
};
#pragma DATA_SECTION(dst, ”table2”)
/* dst data table address */
Uint16 dst[N];
```

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

```
DMA_Config myconfig = {  /* DMA configuration structure*/

     DMA_DMACSDP_RMK(
       DMA_DMACSDP_DSTBEN_NOBURST , /* Destination burst :-
                                 DMA_DMACSDP_DSTBEN_NOBURST
                                 DMA_DMACSDP_DSTBEN_BURST4
                                                           */

       DMA_DMACSDP_DSTPACK_OFF,     /* Destination packing :-
                                 DMA_DMACSDP_DSTPACK_ON
                                 DMA_DMACSDP_DSTPACK_OFF    */

       DMA_DMACSDP_DST_SARAM ,     /* Destination selection :-
                                 DMA_DMACSDP_DST_SARAM
                                 DMA_DMACSDP_DST_DARAM
                                 DMA_DMACSDP_DST_EMIF
                                 DMA_DMACSDP_DST_PERIPH     */

       DMA_DMACSDP_SRCBEN_NOBURST  , /* Source burst :-
                                 DMA_DMACSDP_SRCBEN_NOBURST
                                 DMA_DMACSDP_SRCBEN_BURST4  */

       DMA_DMACSDP_SRCPACK_OFF,     /* Source packing :-
                                 DMA_DMACSDP_SRCPACK_ON
                                 DMA_DMACSDP_SRCPACK_OFF    */

       DMA_DMACSDP_SRC_SARAM ,     /* Source selection :-
                                 DMA_DMACSDP_SRC_SARAM
                                 DMA_DMACSDP_SRC_DARAM
                                 DMA_DMACSDP_SRC_EMIF
                                 DMA_DMACSDP_SRC_PERIPH     */

      DMA_DMACSDP_DATATYPE_16BIT   /* Data type :-
                                 DMA_DMACSDP_DATATYPE_8BIT
                                 DMA_DMACSDP_DATATYPE_16BIT
                                 DMA_DMACSDP_DATATYPE_32BIT */

       ) /* DMACSDP */
```

```
DMA_DMACCR_RMK(
DMA_DMACCR_DSTAMODE_POSTINC, /* Destination address mode :-
                     DMA_DMACCR_DSTAMODE_CONST
                     DMA_DMACCR_DSTAMODE_POSTINC
                     DMA_DMACCR_DSTAMODE_SGLINDX
                     DMA_DMACCR_DSTAMODE_DBLINDX */

 DMA_DMACCR_SRCAMODE_POSTINC, /* Source address mode :-
                     DMA_DMACCR_SRCAMODE_CONST
                     DMA_DMACCR_SRCAMODE_POSTINC
                     DMA_DMACCR_SRCAMODE_SGLINDX
                     DMA_DMACCR_SRCAMODE_DBLINDX */

  DMA_DMACCR_ENDPROG_OFF, /* End of programmation bit :-
                     DMA_DMACCR_ENDPROG_ON
                     DMA_DMACCR_ENDPROG_OFF      */

  DMA_DMACCR_REPEAT_OFF,/*  Repeat condition :-
                     DMA_DMACCR_REPEAT_ON
                     DMA_DMACCR_REPEAT_ALWAYS
                     DMA_DMACCR_REPEAT_ENDPROG1
                     DMA_DMACCR_REPEAT_OFF       */

 DMA_DMACCR_AUTOINIT_OFF,/* Auto initialization bit :-
                     DMA_DMACCR_AUTOINIT_ON
                     DMA_DMACCR_AUTOINIT_OFF     */

 DMA_DMACCR_EN_STOP,/* Channel enable :-
                     DMA_DMACCR_EN_START
                     DMA_DMACCR_EN_STOP          */

 DMA_DMACCR_PRIO_LOW, /*  Channel priority :-
                      DMA_DMACCR_PRIO_HI
                       DMA_DMACCR_PRIO_LOW         */


 DMA_DMACCR_FS_ELEMENT, /*  Frame\Element Sync :-
                     DMA_DMACCR_FS_ENABLE
                     DMA_DMACCR_FS_DISABLE
                     DMA_DMACCR_FS_ELEMENT
                     DMA_DMACCR_FS_FRAME         */

 DMA_DMACCR_SYNC_NONE  /* Synchronization control :-
                     DMA_DMACCR_SYNC_NONE
                     DMA_DMACCR_SYNC_REVT0
                     DMA_DMACCR_SYNC_XEVT0
                     DMA_DMACCR_SYNC_REVTA0
                     DMA_DMACCR_SYNC_XEVTA0
                     DMA_DMACCR_SYNC_REVT1
                     DMA_DMACCR_SYNC_XEVT1
                     DMA_DMACCR_SYNC_REVTA1
                     DMA_DMACCR_SYNC_XEVTA1
                     DMA_DMACCR_SYNC_REVT2
```

```
                              DMA_DMACCR_SYNC_XEVT2
                              DMA_DMACCR_SYNC_REVTA2
                              DMA_DMACCR_SYNC_XEVTA2
                              DMA_DMACCR_SYNC_TIM1INT
                              DMA_DMACCR_SYNC_TIM2INT
                              DMA_DMACCR_SYNC_EXTINT0
                              DMA_DMACCR_SYNC_EXTINT1
                              DMA_DMACCR_SYNC_EXTINT2
                              DMA_DMACCR_SYNC_EXTINT3
                              DMA_DMACCR_SYNC_EXTINT4
                              DMA_DMACCR_SYNC_EXTINT5      */
     ) /* DMACCR */

    DMA_DMACICR_RMK(

    DMA_DMACICR_BLOCKIE_ON , /* Whole block interrupt enable :-
                              DMA_DMACICR_BLOCKIE_ON
                              DMA_DMACICR_BLOCKIE_OFF      */

    DMA_DMACICR_LASTIE_ON, /* Last frame Interrupt enable :-
                                DMA_DMACICR_LASTIE_ON
                                DMA_DMACICR_LASTIE_OFF       */

    DMA_DMACICR_FRAMEIE_ON, /* Whole frame interrupt enable :-
                                   DMA_DMACICR_FRAMEIE_ON
                                   DMA_DMACICR_FRAMEIE_OFF      */

   DMA_DMACICR_FIRSTHALFIE_ON, /* HAlf frame interrupt enable :-

                                   DMA_DMACICR_FIRSTHALFIE_ON
                                   DMA_DMACICR_FIRSTHALFIE_OFF  */

  DMA_DMACICR_DROPIE_ON, /* Sync. event drop interrupt enable :-
                              DMA_DMACICR_DROPIE_ON
                              DMA_DMACICR_DROPIE_OFF       */

      DMA_DMACICR_TIMEOUTIE_ON /* Time out inetrrupt enable  :-
                              DMA_DMACICR_TIMEOUTIE_ON

                              DMA_DMACICR_TIMEOUTIE_OFF    */
    ), /* DMACICR */

    (DMA_AdrPtr) &src, /* DMACSSAL */
     0, /* DMACSSAU */
     (DMA_AdrPtr)&dst, /* DMACDSAL */
     0, /* DMACDSAU */
     N, /* DMACEN */
     1, /* DMACFN */
     0, /* DMACFI */
     0  /* DMACEI */
};
```

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

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

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

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

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

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

By default, the TMS320C55xx compiler assigns all data symbols word addresses. The DMA however, expects all addresses to be byte addresses. Therefore, you must shift the address by 2 in order to change the word address to a byte address for the DMA transfer.

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

```
myconfig.dmacssal =
(DMA_AdrPtr)(((Uint16)(myconfig.dmacssal)<<1)&0xFFFF);
myconfig.dmacdsal =
(DMA_AdrPtr)(((Uint16)(myconfig.dmacdsal)<<1)&0xFFFF);
myconfig.dmacssau - (((Uint32) &src) >> 15) & 0xFFFF;
myconfig.dmacdsau - (((Uint32) &dst) >> 15) & 0xFFFF;
DMA_config(myhDma, &myConfig);   /* Configure Channel  */
```

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

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

Step 8: Wait for FRAME status bit in DMA status register to signal transfer is complete

```
while (!DMA_FGETH(myhDma, DMACSR, FRAME)) {
  ;
}
```

Step 9: Close DMA channel

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

2.3 Compiling and Linking with the CSL Using Code Composer Studio

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

Step 1: Specify the target device. (Refer to section 2.3.1)

Step 2: Determine whether or not you are using a small or large memory model and specify the CSL and RTS libraries you require. (Refer to section 2.3.1.1)

Step 3: Create the linker command file (with a special .csldata section) and add the file to the project. (Refer to section 2.3.1.2)

Step 4: Determine if you must enable inlining. (Refer to section 2.3.1.3)

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

Note:

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

Table 2-1. CSL Directory Structure

This CSL component...	Is located in this directory...
Libraries	<Install_Dir>\c5500\csl\lib
Source Library	<Install_Dir>\c5500\csl\lib
Include files	<Install_Dir>\c5500\csl\include
Examples	<Install_Dir>\examples\<target>\csl
Documentation	<Install_Dir>\docs

2.3.1 Specifying Your Target Device

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

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

Step 2: In the Build Options dialog box, select the Compiler tab (see Figure 2-1).

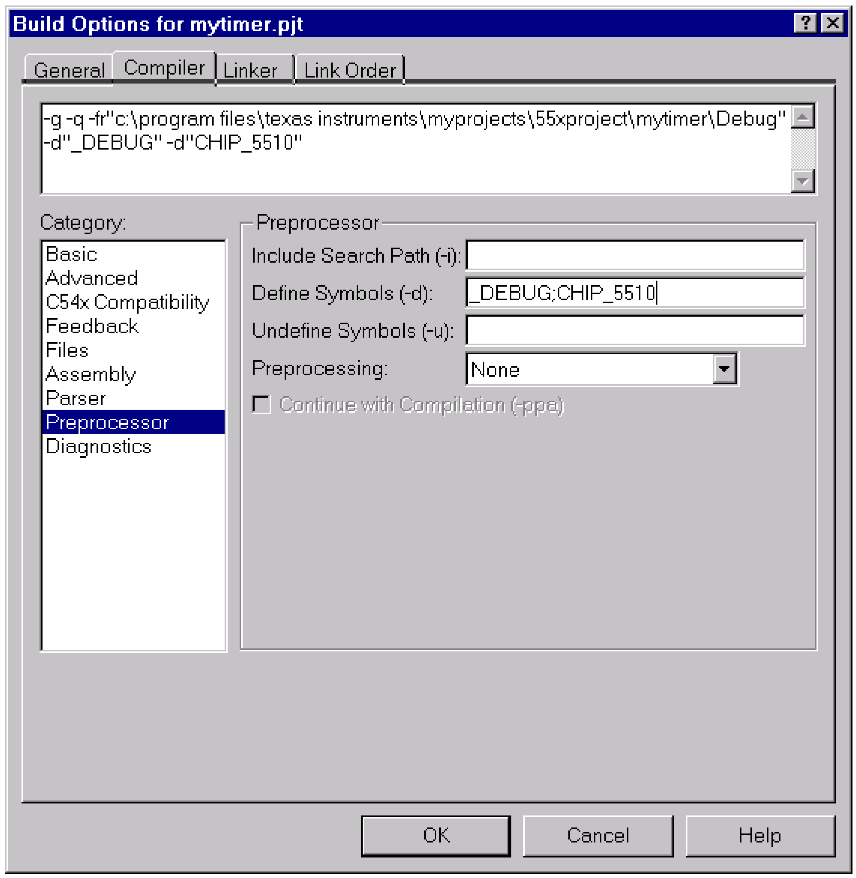
Step 3: In the Category list box, highlight Preprocessor.

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

For example, if you are using the 5510PG1.2 device, enter CHIP_5510PG1_2.

Step 5: Click OK.

Figure 2-1. Defining the Target Device in the Build Options Dialog



2.3.1.1 Large/Small Memory Model Selection

Use of CSL requires that all data resides in the base 64k (Page 0) of memory because of the way in which the small data memory model is implemented.

Page independence for the small data memory model is achieved in the compiler by setting all XAR registers to initially point to the area in memory where the .bss section is located. This is done when the C environment boot routine _c_int00 is executed. The compiler then uses ARx addressing for all data accesses, leaving the upper part of XARx untouched.

Because, CSL is written in C, it relies on the compiler to perform the data/peripheral memory access to read/write peripheral and CPU registers. So in the small data memory model, all peripheral/CPU registers are accessed via ARx addressing. Because the peripheral control registers and CPU status registers reside in the base 64K of I/O and data space respectively, this forces all data to be on page 0 of memory when compiling in small model and using the CSL.

Note that this is a problem only when using the small data memory model. This limitation does not exist when compiling with a large data memory model.

If you use any large memory model libraries, define the -ml option for the compiler and link with the large memory model runtime library (rts55x.lib) using the following steps:

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

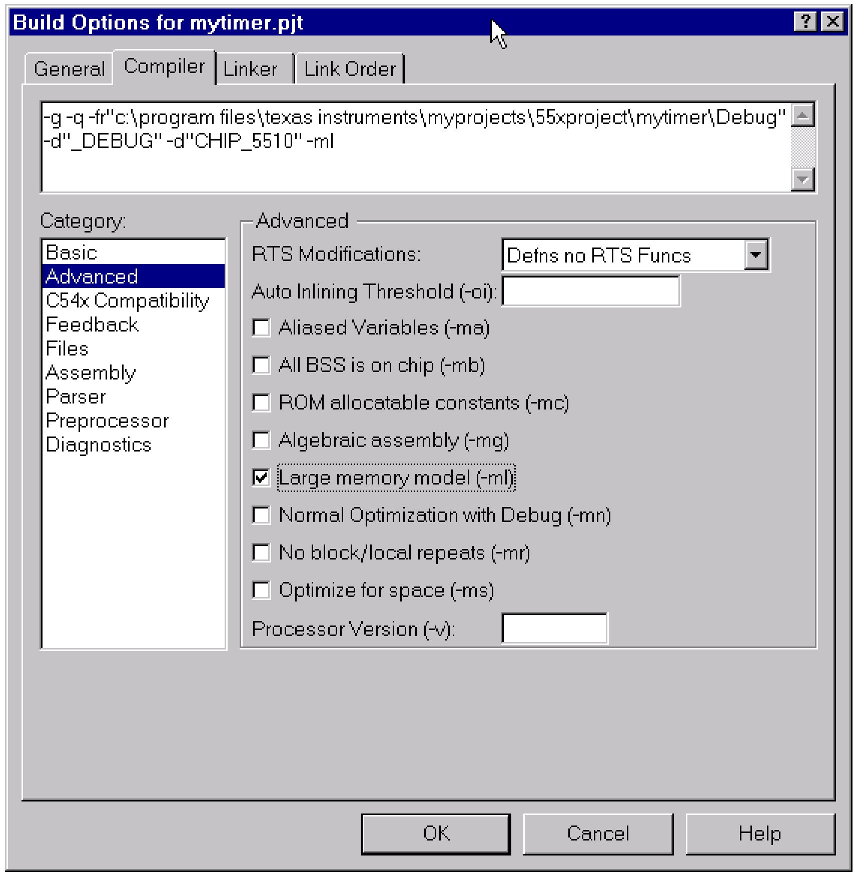
Step 2: In the Build Options dialog box, select the Compiler Tab (Figure 2-2).

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

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

Step 5: Click OK.

Figure 2-2. Defining Large Memory Model



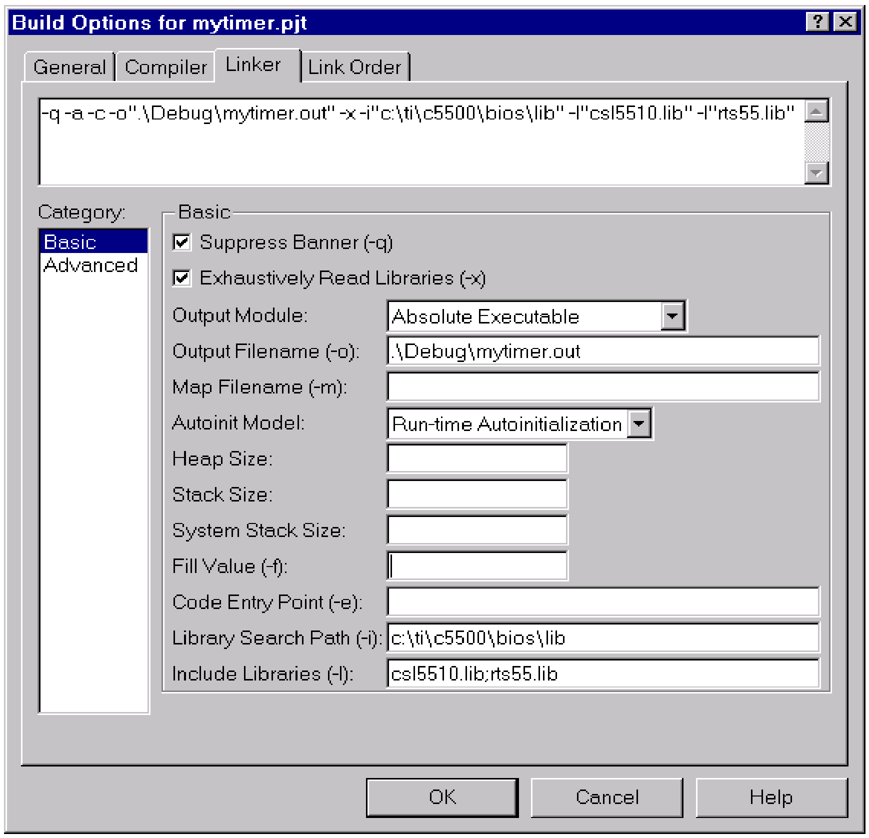
Then, you must specify which CSL and RTS libraries will be linked in your project.

- In Code Composer Studio, select Project → Options.
- In the Build Options dialog box, Select the Linker Tab (see Figure 2-3).
- In the Category list, highlight Basic.
- The Library search Path field (-l), should show: <Install_Dir>\c5500\csl\lib (automatically configured by Code Composer Studio)
- In the Include Libraries (-l) field, enter the correct library from Table 1-2, on page 1-5.

For example, if you are using the 5510 device, enter csl5510.lib for near mode or csl5510x.lib for far mode. In addition, you must include the corresponding rts55.lib or rts55x.lib compiler runtime support libraries.

Click OK.

Figure 2-3. Defining Library Paths



2.3.1.2 Creating a Linker Command File

The CSL has two requirements for the linker command file:

You must allocate the .csldata section.

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

You must reserve address 0x7b in scratch pad memory

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

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

Example 2-1. Using a Linker Command File

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

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

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

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

2.3.1.3 Using Function Inlining

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

Chapter 3

ADC Module

This chapter describes the ADC module, lists the API structure, functions, and macros within the module, and provides an ADC API reference section. The ADC module is not handle-based.

Topic **Page**

3.1 Overview

The configuration of the ADC can be performed by using one of the following methods:

Register-based configuration

A register-based configuration is performed by calling ADC_config() or any of the SET register/field macros.

Parameter-based configuration

A parameter-based configuration can be performed by calling ADC_setFreq(). Using ADC_setFreq() to initialize the ADC registers for the desired sampling frequency is the recommended approach. The sampled value can also be read using the ADC_read() function.

Compared to the register-based approach, this method provides a higher level of abstraction. The downside is larger code size and higher cycle counts.

Table 3-1 lists the configuration structure used to set up the ADC.

Table 3-2 lists the functions available for use with the ADC module.

Table 3-3 lists ADC registers and fields.

Table 3-1. ADC Configuration Structures

Syntax	Description	See page…
ADC_Config	ADC configuration structure used to set up the ADC (register based)	3-4

Table 3-2. ADC Functions

Syntax	Description	See page…
ADC_config()	Sets up the ADC using the configuration structure	3-5
ADC_getConfig()	Obtains the current configuration of all the ADC registers	3-5
ADC_read()	Performs conversion and reads sampled values from the data register	3-6
ADC_setFreq()	Sets up the ADC using parameters passed	3-6

Table 3-3. ADC Registers

Register	Field
ADCCTL	CHSELECT, ADCSTART
ADCDATA	ADCDATA(R), CHSELECT, ADCBUSY(R)
ADCCLKDIV	CONVRATEDIV, SAMPTIMEDIV
ADCCLKCTL	CPUCLKDIV, IDLEEN

Note: R = Read Only; W = Write; By default, most fields are Read/Write

3.2 Configuration Structures

The following is the configuration structure used to set up the ADC (register based).

ADC_Config *ADC configuration structure used to set up the ADC interface*

Structure ADC_Config

Members

Uint16 adcctl	Control Register
Uint16 adcclkdiv	Clock Divider Register
Uint16 adcclkctl	Clock Control Register

Description ADC configuration structure used to set up the ADC. You create and initialize this structure and then pass its address to the ADC_config() function. You can either use literal values or use ADC_RMK macros to create the structure member values.

Example

```
ADC_Config Config = {
   0xFFFF, /* ADCCTL    */
   0xFFFF, /* ADCCLKDIV */
   0xFFFF  /* ADCCLKCTL */

}
```

3.3 Functions

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

ADC_config

Writes the values to ADC registers using the configuration structure

Function void ADC_config(ADC_Config *Config);

Arguments Config — Pointer to an initialized configuration structure (see ADC_Config)

Return Value None

Description Writes a value to set up the ADC using the configuration structure. The values of the configuration structure are written to the port registers.

Example

```
ADC_Config Config = {
   0xFFFF,    /* ADCCTL  */
   0xFFFF,    /* ADCCLKDIV */
   0xFFFF    /* ADCCLKCTL */

};
```

ADC_getConfig

Writes values to ADC registers using the configuration structure

Function void ADC_getConfig(ADC_Config *Config);

Arguments Config — Pointer to a configuration structure (see ADC_Config)

Return Value None

Description Reads the current value of all ADC registers being used and places them into the corresponding configuration structure member.

Example

```
ADC_Config testConfig;
ADC_getConfig(&testConfig);
```

ADC_read — *Performs an ADC conversion and reads the digital data*

Function

void ADC_read(int channelnumber,
Uint16 date,
int length);

Arguments

int channelnumber	Analog Input Selector Value from 0-3
Uint16 *data	Data array to store digital data converted from analog signal
int length	number of samples to convert

Return Value

None

Description

Performs conversions by setting the ADC start bit (ADCCTL) and polling ADC busy (ADCDATA) until done. The sampled values are then read into the array.

Example

```
int i=7,j=15,k=1;
int channel=0,samplenumber=3;
Uint16 samplestorage[3]={0,0,0};

ADC_setFreq(i,j,k);
ADC_read(channel,samplestorage,samplenumber);
           /* performs 3 conversions from analog input 0 */
           /* and reads the digital data into the        */
           /* samplestorage array.                       */
```

ADC_setFreq — *Initializes the ADC for a desired sampling frequency*

Function

void ADC_setFreq(int cpuclkdiv,
int convratediv,
int sampletimediv);

Arguments

cpuclkdiv	CPU clock divider value (inside ADCCLKCTL register) Value from 0-255
convratediv	Conversion clock rate divider value (inside ADCCLKDIV) Value from 0-16
sampletimediv	Sample and hold time divider value (inside ADCCLKDIV) Value from 0-255

Return Value

None

Description

Initializes the ADC peripheral by setting the system clock divider, conversion clock rate divider, and sample and hold time divider values into the appropriate registers.

Refer to the *TMS320C55x Peripherals Reference Guide* (SPRU317A) for explanations on how to produce a desired ADC sampling frequency using these three parameters.

Example

```
int i=7,j=15,k=1;
ADC_setFreq(i,j,k);
/* This example sets the ADC sampling frequency    */
/* to 21.5 KHZ, given a 144 MHZ clockout frequency */
```

3.4 Macros

This section contains descriptions of the macros available in the ADC module. See the general macros description in section 1.5 on page 1-11. To use these macros, you must include “csl_adc.h.”

The ADC module defines macros that have been designed for the following purposes:

The RMK macros create individual control-register masks for the following purposes:

- To initialize a ADC_Config structure that can be passed to functions such as ADC_Config().
- To use as arguments for the appropriate RSET macro.

Other macros are available primarily to facilitate reading and writing individual bits and fields in the ADC control registers.

Table 3-4. ADC Macros

(a) Macros to read/write ADC register values

Macro	Syntax
ADC_RGET()	Uint16 ADC_RGET(*REG*)
ADC_RSET()	Void ADC_RSET(*REG*, Uint16 *regval*)

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

Macro	Syntax
ADC_FGET()	Uint16 ADC_FGET(*REG*, *FIELD*)
ADC_FSET()	Void ADC_FSET(REG,FIELD,Uint16 fieldval)

Notes: 1) *REG* indicates the registers, ADCCTL, ADCCLKDIV, ADCCLKCTL

2) *FIELD* indicates the register field name
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 3-4. ADC Macros (Continued)

(c) Macros to create values to ADC registers and fields (Applicable to registers with more than one field)

Macro	Syntax
ADC_REG_RMK()	Uint16 ADC_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
ADC_FMK()	Uint16 ADC_FMK(*REG*, *FIELD*, *fieldval*)

(d) Macros to read a register address

Macro	Syntax
ADC_ADDR()	Uint16 ADC_ADDR(REG)

Notes:
1) *REG* indicates the registers, ADCCTL, ADCCLKDIV, ADCCLKCTL
2) *FIELD* indicates the register field name
   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*).

3.5 Examples

ADC programming examples using CSL are provided in the:

\examples\<target>\CSL directory of Code Composer Studio

and in *Programming the C5509 ADC Peripheral Application Report* (SPRA785).

Chapter 4

CHIP Module

This chapter describes the CHIP module, lists the API functions and macros within the module, and provides a CHIP API reference section. The CSL CHIP module is not handle-based; it offers general CPU functions and macros for C55x register accesses.

Topic **Page**

4.1 Overview

The following sections contain all the information required to run the CHIP module. Table 4-1 lists the functions available, section 4.3 contains the macros, and Table 4-2 lists CHIP registers.

Table 4-1. CHIP Functions

Function	Description	See page ...
CHIP_getDieId_High32	Returns the high 32 bits of the DieID register.	4-3
CHIP_getDieId_Low32	Returns the low 32 bits of the DieID register.	4-3
CHIP_getRevId	Returns the value of the RevID register.	4-3

4.1.1 CHIP Registers

Table 4-2. CHIP Registers

Register	Field
ST0_55	ACOV0, ACOV1, ACOV2, ACOV3, TC1, TC2, CARRY, DP
ST1_55	BRAF, CPL, XF, HM, INTM, M40, SATD, SXMD, C16, FRCT, C54CM, ASM
ST2_55	ARMS, DBGM, EALLOW, RDM, CDPLC, AR7LC, AR6LC, AR5LC, AR4LC, AR3LC, AR2LC, AR1LC, AR0LC
ST3_55	CAFRZ, CAEN, CACLR, HINT, CBERR, MPNMC, SATA, AVIS, CLKOFF, SMUL, SST
IER0	DMAC5, DMAC4, XINT2, RINT2, INT3, DSPINT, DMAC1, XINT1, RINT1, RINT0, TINT0, INT2, INT0
IER1	INT5, TINT1, DMAC3, DMAC2, INT4, DMAC0, XINT0, INT1
IFR0	DMAC5, DMAC4, XINT2, RINT2, INT3, DSPINT, DMAC1, XINT1, RINT1, RINT0, TINT0, INT2, INT0
IFR1	INT5, TINT1, DMAC3, DMAC2, INT4, DMAC0, XINT0, INT1
IVPD	IVPD
IVPH	IVPH
PDP	PDP
SYSR	HPE, BH, HBH, BOOTM3(R), CLKDIV
XBSR	CLKOUT, OSCDIS, EMIFX2, SP2, SP1, PP

Note: R = Read Only; W = Write; By default, most fields are Read/Write

4.2 Functions

The following are functions available for use with theCHIP module.

CHIP_getDieId_High32 *Get the high 32 bits of the Die ID register*

Function Uint32 CHIP_getDieId_High32();

Arguments None

Return Value high 32 bits of Die ID

Description Returns high 32 bits of the Die ID register

Example

```
Uint32 DieId_32_High;
…
DieId_32_High = CHIP_getDieId_High32();
```

CHIP_getDieId_Low32 *Get the low 32 bits of the Die ID register*

Function Uint32 CHIP_getDieId_Low32();

Arguments None

Return Value low 32 bits of Die ID

Description Returns low 32 bits of the Die ID register

Example

```
Uint32 DieId_32_Low;
…
DieId_32_Low = CHIP_getDieId_Low32();
```

CHIP_getRevId *Gets the Rev ID Register*

Function Uint16 CHIP_getRevId();

Arguments None

Return Value Rev ID

Description This function returns the Rev Id register.

Example

```
Uint16 RevId;
...
RevId = CHIP_getRevId();
```

4.3 Macros

CSL offers a collection of macros to gain individual access to the CHIP peripheral registers and fields. Table 4-3 contains a list of macros available for the CHIP module. To use them, include “csl_chip.h.”

Table 4-3. CHIP Macros

(a) Macros to read/write CHIP register values

Macro	**Syntax**
CHIP_RGET()	Uint16 CHIP_RGET(REG)
CHIP_RSET()	void CHIP_RSET(REG, Uint16 regval)

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

Macro	**Syntax**
CHIP_FGET()	Uint16 CHIP_FGET(REG, FIELD)
CHIP_FSET()	void CHIP_FSET(REG,FIELD, Uint16 fieldval)

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

Macro	**Syntax**
CHIP_REG_RMK()	Uint16 CHIP_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
CHIP_FMK()	Uint16 CHIP_FMK(REG, FIELD, fieldval)

(d) Macros to read a register address

Macro	**Syntax**
CHIP_ADDR()	Uint16 CHIP_ADDR(REG)

Notes:
1) *REG* indicates the register XBSR
2) *FIELD* indicates the register field name
   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*).

Chapter 5

DAT Module

This chapter describes the DAT (data) module, lists the API functions within the module, and provides a DAT API reference section. The handle-based DAT module allows you to use DMA hardware to move data.

Topic **Page**

5.1 Overview

The handle-based DAT(data) module allows you to use DMA hardware to move data. This module works the same for all devices that support the DMA regardless of the type of the 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().

It should be noted that for 5509/5510/5509A targets, the source as well as destination data is in SARAM (since DMA internally is configured for this port) and for 5502, the data is in DARAM (since DMA internally is configured for DARAM PORT0).

Table 5-1 lists the functions for use with the DAT modules. The functions are listed in alphabetical order. Your application **must** call DAT_open() and DAT_close(); the other functions are used at your discretion.

Table 5-1. DAT Functions

Function	Purpose	See page ...
DAT_close()	Closes the DAT	5-3
DAT_copy()	Copies data of specific length from the source memory to the destination memory.	5-3
DAT_copy2D()	Copies 2D data of specific line length from the source memory to the destination memory.	5-4
DAT_fill()	Fills the destination memory with a data value	5-5
DAT_open()	Opens the DAT with a channel number and a channel priority	5-6
DAT_wait()	DAT wait function	5-7

5.2 Functions

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

DAT_close — *Closes a DAT device*

Function
void DAT_close(
    DAT_Handle hDat
);

Arguments
hDat

Return Value
None

Description
Closes a previously opened DAT device. Any pending requests are first allowed to complete.

Example

```
DAT_close(hDat);
```

DAT_copy — *Performs bytewise copy from source to destination memory*

Function
Uint16 DAT_copy(DAT_Handle hDat,
    (DMA_AdrPtr)Src,
    (DMA_AdrPtr)Dst,
    Uint16 ElemCnt
);

Arguments

hDat	Device Handler (see DAT_open)
Src	Pointer to source memory assumes byte addresses
Dst	Pointer to destination memory assumes byte addresses
ByteCnt	Number of bytes to transfer to *Dst

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 the memory values from the Src to the Dst memory locations.

Example

```
DAT_copy(hDat,         /* Device Handler */
   (DMA_AdrPtr)0xF000, /* src            */
   (DMA_AdrPtr)0xFF00, /* dst            */
   0x0010              /* ByteCnt        */
);
```

DAT_copy2D

Copies 2-dimensional data from source memory to destination memory

Function

```
Uint16 DAT_copy2D(DAT_Handle hDat,
    Uint16 Type,
    (DMA_AdrPtr)Src,
    (DMA_AdrPtr)Dst,
    Uint16 LineLen,
    Uint16 LineCnt,
    Uint16 LinePitch
);
```

Arguments

hDat	Device Handler (see DAT_open)
Type	Type of 2D DMA transfer, must be one of the following: DAT_1D2D : 1D to 2D transfer DAT_2D1D : 2D to 1D transfer DAT_2D2D : 2D to 2D transfer
Src	Pointer to source memory assumes byte addresses
Dst	Pointer to destination memory assumes byte addresses
LineLen	Number of 16-bit words in one line
LineCnt	Number of lines to copy
LinePitch	Number of bytes between start of one line to start of next line (always an even number since underlying DMA transfer assumes 16-bit elements)

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 the memory values from the Src to the Dst memory locations.

Example

```
DAT_copy2D(hDat,       /* Device Handler */
   DAT_2D2D,           /* Type           */
   (DMA_AdrPtr)0xFF00, /* src            */
   (DMA_AdrPtr)0xF000, /* dst            */
   0x0010,             /* linelen        */
   0x0004,             /* Line Cnt       */
   0x0110,             /* LinePitch      */
);
```

DAT_fill

Fills DAT destination memory with value

Function

```
Uint16 DAT_fill(DAT_Handle hDat,
   (DMA_AdrPtr)Dst,
   Uint16 ElemCnt,
   Uint16 *Value
 );
```

Arguments

hDat	Device Handler (DAT_open)
(DMA_AdrPtr)Dst	Pointer to destination memory location
ElemCnt	Number of 16-bit words to fill
*Value	Pointer to value that will fill the memory

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 the destination memory with a value for a specified byte count 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 value;
   DAT_fill(hDat,      /* Device Handler */
   (DMA_AdrPtr)0x00FF, /* dst            */
   0x0010,             /* ElemCnt        */
   &value              /* Value          */
   );
```

DAT_open — *Opens DAT for DAT calls*

Function

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

Arguments

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

- DAT_CHA_ANY (allocates Channel 2 or 3)
- DAT_CHA0
- DAT_CHA1
- DAT_CHA2
- DAT_CHA3
- DAT_CHA4
- DAT_CHA5

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

DAT_Handle hdat — Device Handler (see DAT_open). If the requested DMA channel is currently being used, an INV(-1) value is returned.

Description

Before a DAT channel can be used, it must first be opened by this function with an assigned priority. Once opened, it cannot be opened again until closed (see DAT_close).

Example

```
DAT_open(DAT_CHA0,DAT_PRI_LOW,0);
```

DAT_wait — *DAT wait function*

Function

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

Arguments

hDat   Device handler (see DAT_open).

Return Value

none

Description

This function polls the IFRx 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

```
DAT_wait(myhDat);
```

Chapter 6

DMA Module

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

Topic **Page**

6.1 Overview

Table 6-2 summarizes the primary API functions and macros.

Your application must call DMA_open() and DMA_close().

Your application can also call DMA_reset(hDma).

You can perform configuration by calling DMA_config() or any of the SET register macros.

Because DMA_config() initializes 11 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 use DMA_config() to initialize the DMA registers.

The CSL DMA module defines macros (see section 6.4) designed for these primary purposes:

The *RMK* macros create individual control-register masks for the following purposes:

- To initialize an DMA_Config structure that you then pass to functions such as DMA_config().
- To use as arguments for the appropriate SET macro.

Other macros are available primarily to facilitate reading and writing individual bits and fields in the DMA control registers.

Table 6-1. DMA Configuration Structure

Configuration Structure	Description	See page ...
DMA_Config	DMA configuration structure used to setup the DMA interface	6-5

Table 6-2. DMA Functions

Function	Description	See page ...
DMA_close()	Closes the DMA and its corresponding handler	6-6
DMA_config()	Sets up DMA using configuration structure (DMA_Config)	6-6
DMA_getConfig()	Reads the DMA configuration	6-7
DMA_getEventId()	Returns the IRQ Event ID for the DMA completion interrupt	6-7
DMA_open()	Opens the DMA and assigns a handler to it	6-8
DMA_pause()	Interrupts the transfer in the corresponding DMA channel	6-9
DMA_reset()	Resets the DMA registers with default values	6-9
DMA_start()	Enables transfers in the corresponding DMA channel	6-9
DMA_stop()	Disables the transfer in the corresponding DMA channel	6-10

Table 6-3. DMA Macros

Macro	Description	See page ...
DMA_ADDR()	Gets the address of a DMA register	6-11
DMA_ADDRH()	Gets the address of a DMA local register for channel used in hDma	6-11
DMA_FGET()	Gets the DMA register field value	6-12
DMA_FGETH()	Gets the DMA register field value	6-13
DMA_FMK()	Creates register value based on individual field values	6-14
DMA_FSET()	Sets the DMA register value to regval	6-15
DMA_FSETH()	Sets value of register field	6-16
DMA_REG_RMK()	Creates register value based on individual field values	6-17
DMA_RGET()	Gets value of a DMA register	6-18
DMA_RGETH()	Gets value of DMA register used in handle	6-19
DMA_RSET()	Sets the DMA register REG value to regval	6-19
DMA_RSETH()	Sets the DMA register LOCALREG for the channel associated with handle to the value regval	6-20

6.1.1 DMA Registers

Table 6-4. DMA Registers

Register	Field
DMAGCR	FREE, EHPIEXCL, EHPIPRIO
DMACSDP	DSTBEN, DSTPACK, DST, SRCBEN, SRCPACK, SRC, DATATYPE
DMACCR	DSTAMODE, SRCAMODE, ENDPROG, FIFOFLUSH, REPEAT, AUTOINIT, EN, PRIO, FS, SYNC
DMACICR	BLOCKIE, LASTIE, FRAMEIE, FIRSTHALFIE, DROPIE, TIMEOUTIE
DMACSR	(R)SYNC, (R)BLOCK, (R)LAST, (R)FRAME, (R)HALF, (R)DROP, (R)TIMEOUT
DMACSSAL	SSAL
DMACSSAU	SSAU
DMACDSAL	DSAL
DMACDSAU	DSAU
DMACEN	ELEMENTNUM
DMACFI	FRAMENDX
DMACEI	ELEMENTNDX
DMACSFI	FRAMENDX
DMACSEI	ELEMENTNDX
DMACDFI	FRAMENDX
DMACDEI	ELEMENTNDX
DMACSAC	DMACSAC
DMACDAC	DMACDAC
DMAGTCR	PTE, ETE, ITE1, ITE0
DMAGTCR	DTCE, STCE
DMAGSCR	COMPMODE

Note: R = Read Only; W = Write; By default, most fields are Read/Write

6.2 Configuration Structures

The following configuration structure is used to set up the DMA.

DMA_Config — *DMA configuration structure used to set up DMA interface*

Structure

DMA_Config

Members

Uint16 dmacsdp	DMA Channel Control Register
Uint16 dmaccr	DMA Channel Interrupt Register
Uint16 dmacicr	DMA Channel Status Register
(DMA_AdrPtr) dmacssal	DMA Channel Source Start Address (Lower Bits)
Uint16 dmacssau	DMA Channel Source Start Address (Upper Bits)
(DMA_AdrPtr) dmacdsal	DMA Channel Source Destination Address (Lower Bits)
Uint16 dmacdsau	DMA Channel Source Destination Address (Upper Bits)
Uint16 dmacen	DMA Channel Element Number Register
Uint16 dmacfn	DMA Channel Frame Number Register

For CHIP_5509, CHIP_5510PG1_x (x=0, 2)

Int16 dmacfi	DMA Channel Frame Index Register
Int16 dmacei	DMA Channel Element Index Register

For CHIP_5510PG2_x (x=0, 1, 2), 5509A, 5502

Int16 dmacsfi	DMA Channel Source Frame Index Register
Int16 dmacsei	DMA Channel Source Element Index Register
Int16 dmacdfi	DMA Channel Destination Frame Index Register
Int16 dmacdei	DMA Channel Destination Element Index

Description

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

Example

Refer to section 2.2.1, step 2 and step 6.

6.3 Functions

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

DMA_close — *Closes DMA*

Function

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

Arguments

hDma	Device Handle, see DMA_open();

Return Value None

Description Closes a previously opened DMA device. The DMA event is disabled and cleared. The DMA registers are set to their default values.

Example Refer to section 2.2.1, step 6.

DMA_config — *Writes value to up DMA using configuration structure*

Function

```
void DMA_config(DMA_Handle hDma,
    DMA_Config *Config
);
```

Arguments

hDma	DMA Device handle
Config	Pointer to an initialized configuration structure

Return Value None

Description Writes a value to the DMA using the configuration structure. The values of the structure are written to the port registers. See also DMA_Config.

Example Refer to section 2.2.1, step 2 and step 6.

DMA_getConfig *Reads the DMA configuration*

Function

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

Arguments

hDma	DMA device handle
Config	Pointer to an un-initialized configuration structure

Return Value None

Description Reads the DMA configuration into the Config structure (see DMA_Config).

Example

```
DMA_Config myConfig;
DMA_getConfig (hDma, &myConfig);
```

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);
```

DMA_open — *Opens DMA for DMA calls*

Function

DMA_Handle DMA_open(
    int ChaNum,
    Uint32 flags
);

Arguments

ChaNum	DMA Channel Number: DMA_CHA0, DMA_CHA1 DMA_CHA2, DMA_CHA3, DMA_CHA4, DMA_CHA5, DMA_CHA_ANY
flags	Event Flag Number: Logical open or DMA_OPEN_RESET

Return Value

DMA_Handle	Device handler

Description

Before a DMA device can be used, it must first be opened by this function. Once opened, it cannot be opened again until closed (see DMA_close). The return value is a unique device handle that is used in subsequent DMA API calls. If the function fails, INV is returned. If the DMA_OPEN_RESET is specified, then the power on defaults are set and any interrupts are disabled and cleared.

Example

```
DMA_Handle hDma;
...
hDma = DMA_open(DMA_CHA0,0);
```

DMA_pause

Interrupts the transfer in the corresponding DMA channel

Function

void DMA_pause(hDMA);

Arguments

hDma	Handle to DMA channel; see DMA_open().

Return Value

None

Description

If a DMA transfer is already active in the channel, DMA_pause will cause the DMA controller to stop the transfer and reset the channel.

Example

```
DMA_pause(hDma);
```

DMA_reset

Resets DMA

Function

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

Arguments

hDma	Device handle, see DMA_open();

Return Value

None

Description

Resets the DMA device. Disables and clears the interrupt event and sets the DMA registers to default values. If INV is specified, all DMA devices are reset.

Example

```
DMA_reset(hDma);
```

DMA_start

Enables transfers in the corresponding DMA channel

Function

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

Arguments

hDma	Handle to DMA channel; see DMA_open().

Return Value

None

Description

Enables the DMA channel indicated by hDma so it can be serviced by the DMA controller at the next available time slot.

Example

```
DMA_start(hDma);
```

DMA_stop

Disables the transfer in the corresponding DMA channel

Function

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

Arguments

hDma	Handle to DMA channel; see DMA_open().

Return Value

None

Description

The transfer in the DMA channel, indicated by hDma, is disabled. The channel can’t be serviced by the DMA controller.

Example

```
DMA_stop(hDma);
```

6.4 Macros

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

Because the DMA has several channels, the macros identify the channel used by either the channel number or the handle used.

DMA_ADDR — *Gets 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 Gets the address of a DMA register.

Example 1 For local registers:

```
myvar = DMA_ADDR (DMACSDP1);
```

Example 2 For global registers:

```
myvar = DMA_ADDR (DMAGCR);
```

DMA_ADDRH — *Gets address of given register*

Macro Uint16 DMA_ADDRH (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: DMACSDP (instead of DMACSDP#)

Return Value Address of register LOCALREG

Description Gets the address of a DMA local register for channel used in hDma

Example

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

DMA_FGET — *Gets value of register field*

Macro

Uint16 DMA_FGET (REG, FIELD)

Arguments

REG — Only writable registers containing more than one field are supported by this macro. Also notice that for local registers, the channel number is used as part of the register name.
For example:
DMAGCR
DMACSDP1

FIELD — Symbolic name for field of register REG Possible values: Field names as listed in the *TMS320C55x DSP Peripherals Reference Guide* (SPRU317C). Only writable fields are allowed.

Return Value

Value of register field

Description

Gets the DMA register field value

Example 1

For local registers:

```
Uint16 myregval;
...
myregval = DMA_FGET (DMACCR0, AUTOINIT);
```

Example 2

For global registers:

```
Uint16 myvar;
...
myregval = DMA_FGET (DMAGCR, EHPIEXCL);
```

DMA_FGETH *Gets value of register field*

Macro Uint16 DMA_FGETH (DMA_Handle hDma, LOCALREG, FIELD)

Arguments

hDma	Handle to DMA channel that identifies the specific DMA channel used.
LOCALREG	Same register as in DMA_RGET(), but without channel number (#). Example: DMACSDP (instead of DMACSDP#) Only registers containing more than one field are supported by this macro.
FIELD	Symbolic name for field of register REG. Possible values: Field names as listed in the *TMS320C55x DSP Peripherals Reference Guide* (SPRU317C). Only writable fields are allowed.

Return Value Value of register field given by FIELD.

Description Gets the DMA register field value

Example

```
DMA_Handle myHandle;
...
myHandle = DMA_open (DMA_CHA0, DMA_OPEN_RESET);
...
myVar = DMA_FGETH (myHandle, DMACCR, AUTOINIT);
```

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

Macro Uint16 DMA_FMK (REG, FIELD, fieldval)

Arguments

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

FIELD — Symbolic name for field of register REG Possible values: Field names as listed in the *TMS320C55x DSP Peripherals Reference Guide* (SPRU317C). Only writable fields are allowed.

fieldval — Field values to be assigned to the writable 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, 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 (DMAGCR, FREE, 1) | DMA_FMK (DMAGCR,
                                        EHPIEXCL, 1);
```

DMA_FSET *Sets value of register field*

Macro Void DMA_FSET (REG, FIELD, fieldval)

Arguments

REG — Only writable registers containing more than one field are supported by this macro. Also notice that for local registers, the channel number is used as part of the register name.
For example:
DMAGCR
DMACSDP1

FIELD — Symbolic name for field of register REG. Possible values: Field names as listed in the *TMS320C55x DSP Peripherals Reference Guide* (SPRU317C). Only writable fields are allowed.

fieldval — Field values to be assigned to the writable register fields.
Rules to follow:
Only writable fields are allowed
If fieldval value exceeds the number of bits allowed for field, fieldval is truncated accordingly.

Return Value None

Description Sets the DMA register field value to fieldval.

Example 1 For local registers:

```
DMA_FSET (DMACCR0, AUTOINIT, 1);
```

Example 2 For global registers:

```
DMA_FSET (DMAGCR, EHPIEXCL, 1);
```

DMA_FSETH — *Sets value of register field*

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

Arguments

hDma	Handle to DMA channel that identifies the specific DMA channel used.
LOCALREG	Same register as in DMA_RGET(), but without channel number (#). Example: DMACSDP (instead of DMACSDP#) 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 *TMS320C55x DSP Peripherals Reference Guide* (SPRU317C). Only writable fields are allowed.
fieldval	Field values to be assigned to the writable register fields. Rules to follow: Only writable fields are allowed Value should be a right-justified constant. If fieldval value exceeds the number of bits allowed for that field, fieldval is truncated accordingly.

Return Value None

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

Example

```
DMA_Handle myHandle;
...
myHandle = DMA_open (DMA_CHA0, DMA_OPEN_RESET);
...
DMA_FSETH (myHandle, DMACCR, AUTOINIT, 1);
```

DMA_REG_RMK *Creates register value based on individual field values*

Macro Uint16 DMA_REG_RMK (fieldval_n,...,fieldval_0)

Arguments

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

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 constant. If fieldval_n value exceeds the number of bits allowed for that field, fieldval_n is truncated accordingly.

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

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

Example

```
Uint16 myregval;
/* free, ehpiexcl, ehpi prio fields */
myregval = DMA_DMAGCR_RMK (0,0,1);
```

DMA_REG_RMK are typically used to initialize a DMA configuration structure used for the DMA_config() function (see section 6.2).

DMA_RGET *Gets value of a 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,

DMACSDP#
DMACCR#
DMACICR#
DMACSR#
DMACSSAL#
DMACSSAU#
DMACDSAL#
DMACDSAU#
DMACEN#
DMACFN#
DMACFI#
DMACEI#

For CHIP_5509 and CHIP_550PG2_0:
DMACSFI#
DMACSEI#
DMACDFI#
DMACDEI#

GLOBALREG Global register name
DMGCR
DMGSCR

Return Value value of register

Description Returns the DMA register value

Example 1 For local registers:

```
Uint16 myvar;
myVar = DMA_RGET(DMACSDP1); /*read DMACSDP for channel 1*/
```

Example 2 For global registers:

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

DMA_RGETH *Gets value of DMA register used in handle*

Macro Uint16 DMA_RGETH (DMA_Handle hDma, LOCALREG)

Arguments

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

Return Value Value of register

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

Example

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

DMA_RSET *Sets 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 Sets the DMA register REG value to regval

Example 1 For local registers:

```
/*DMACSDP for channel 1 = 0x8000 */
DMA_RSET(DMACSDP1, 0x8000);
```

Example 2 For global registers:

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

DMA_RSETH *Sets value of DMA register*

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

Arguments

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

Return Value None

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

Example

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

Chapter 7

EMIF Module

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

Topic **Page**

7.1 Overview

The EMIF configuration can be performed by calling either EMIF_config() or any of the SET register macros. Because EMIF_config() initializes 17 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 use EMIF_config() to initialize the EMIF registers.

The *RMK* macros create individual control-register masks for the following purposes:

- To initialize an EMIF_Config structure that is passed to EMIF_config().
- To use as arguments for the appropriate SET macros.
- Other macros are available primarily to facilitate reading and writing individual bits and fields in the control registers.

Section 7.4 includes a description of all EMIF macros.

Table 7-1 lists the configuration structure used to set up the EMIF.

Table 7-2 lists the functions available for use with the EMIF module.

Table 7-3 lists DMA registers and fields.

Table 7-1. EMIF Configuration Structure

Syntax	Description	See page ...
EMIF_Config	EMIF configuration structure used to setup the EMIF interface	7-6

Table 7-2. EMIF Functions

Syntax	Description	See page ...
EMIF_config()	Sets up EMIF using configuration structure (EMIF_Config)	7-8
EMIF_getConfig()	Reads the EMIF configuration structure	7-9
EMIF_enterselfRefresh (for 5509A only)	Places SDRAM in refresh mode	7-9
EMIF_exitselfRefresh (for 5509A only)	SDRAM exit refresh mode	7-10
EMIF_reset (for 5510xx, 5509, 5509A only)	Resets memory connected in EMIF CE Space	7-10

7.1.1 EMIF Registers

Table 7-3. Registers

(a) EMIF Registers

Register	Field
EGCR	MEMFREQ, WPE, MEMCEN, (R)ARDY, (R)HOLD, (R)HOLDA, NOHOLD
EMIRST	(W)EMIRST
EMIBE	(R)TIME, (R)CE3, (R)CE2, (R)CE1, (R)CE0, (R)DMA, (R)FBUS, (R)EBUS, (R)DBUS, (R)CBUS, (R)PBUS
CE01	MTYPE, RDSETUP, RDSTROBE, RDHOLD
CE11	MTYPE, RDSETUP, RDSTROBE, RDHOLD
CE21	MTYPE, RDSETUP, RDSTROBE, RDHOLD
CE31	MTYPE, RDSETUP, RDSTROBE, RDHOLD
CE02	RDEXHLD, WREXHLD, WRSETUP, WRSTROBE, WRHOLD
CE12	RDEXHLD, WREXHLD, WRSETUP, WRSTROBE, WRHOLD
CE22	RDEXHLD, WREXHLD, WRSETUP, WRSTROBE, WRHOLD
CE32	RDEXHLD, WREXHLD, WRSETUP, WRSTROBE, WRHOLD
CE03	TIMOUT
CE13	TIMOUT
CE23	TIMOUT
CE33	TIMOUT
SDC1	TRC, SDSIZE, SDWID, RFEN, TRCD, TRP
SDPER	PERIOD
SDCNT	(R)COUNTER
INIT	INIT
SDC2	TMRD, TRAS, TACTV2ACTV

Table 7-3. Registers (Continued)

(b) 5502 Registers

Register	Field
GBLCTL1	EK1EN,EK1HZ,NOHOLD,HOLDA,HOLD,ARDY
GBLCTL2	EK2EN,EK2HZ,EK2RATE
CE1CTL1	READ_HOLD,WRITE_HOLD,MTYPE,READ_STROBE,TA
CE1CTL2	READ_SETUP,WRITE_HOLD,WRITE_STROBE,WRITE_SETUP
CE0CTL1	READ_HOLD,WRITE_HOLD,MTYPE,READ_STROBE,TA
CE0CTL2	READ_SETUP,WRITE_HOLD,WRITE_STROBE,WRITE_SETUP
CE2CTL1	READ_HOLD,WRITE_HOLD,MTYPE,READ_STROBE,TA
CE2CTL2	READ_SETUP,WRITE_HOLD,WRITE_STROBE,WRITE_SETUP
CE3CTL1	READ_HOLD,WRITE_HOLD,MTYPE,READ_STROBE,TA
CE3CTL2	READ_SETUP,WRITE_HOLD,WRITE_STROBE,WRITE_SETUP
SDCTL1	SLFRFR,TRC
SDCTL2	TRP,TRCD,INIT,RFEN,SDWTH
SDRFR1	PERIOD,COUNTER
SDRFR2	COUNTER,EXTRA_REFRESHES
SDEXT1	TCL,TRAS,TRRD,TWR,THZP,RD2RD,RD2DEAC,RD2WR,R2WDQM
SDEXT2	R2WDQM,WR2WR,WR2DEAC,WR2RD
CE1SEC1	SYNCRL,SYNCWL,CEEXT,RENEN,SNCCLK
CE0SEC1	SYNCRL,SYNCWL,CEEXT,RENEN,SNCCLK
CE2SEC1	SYNCRL,SYNCWL,CEEXT,RENEN,SNCCLK
CE3SEC1	SYNCRL,SYNCWL,CEEXT,RENEN,SNCCLK
CESCR	CES

Note: R = Read Only; W = Write; By default, most fields are Read/Write

7.2 Configuration Structure

The following is the configuration structure used to set up the EMIF.

EMIF_Config *EMIF configuration structure used to set up EMIF interface*

Structure EMIF_Config

Members

Uint16 egcr	Global Control Register
Uint16 emirst	Global Reset Register
Uint16 ce01	EMIF CE0 Space Control Register 1
Uint16 ce02	EMIF CE0 Space Control Register 2
Uint16 ce03	EMIF CE0 Space Control Register 3
Uint16 ce11	EMIF CE1 Space Control Register 1
Uint16 ce12	EMIF CE1 Space Control Register 2
Uint16 ce13	EMIF CE1 Space Control Register 3
Uint16 ce21	EMIF CE2 Space Control Register 1
Uint16 ce22	EMIIF CE2 Space Control Register 2
Uint16 ce23	EMIF CE2 Space Control Register 3
Uint16 ce31	EMIF CE3 Space Control Register 1
Uint16 ce32	EMIF CE3 Space Control Register 2
Uint16 ce33	EMIF CE3 Space Control Register 3
Uint16 sdc1	EMIF SDRAM Control Register 1
Uint16 sdper	EMIF SDRAM Period Register
Uint16 init	EMIF SDRAM Initialization Register
Uint16 sdc2	EMIF SDRAM Control Register 2

Members 5502 only

Uint16 gblctl1	EMIF Global Control Register 1
Uint16 gblctl2	EMIF Global Control Register 2
Uint16 ce1ctl1	CE1 Space Control Register 1
Uint16 ce1ctl2	CE1 Space Control Register 2
Uint16 ce0ctl1	CE0 Space Control Register 1
Uint16 ce0ctl2	CE0 Space Control Register 2
Uint16 ce2ctl1	CE2 Space Control Register 1
Uint16 ce2ctl2	CE2 Space Control Register 2
Uint16 ce3ctl1	CE3 Space Control Register 1
Uint16 ce3ctl2	CE3 Space Control Register 2
Uint16 sdctl1	SDRAM Control Register 1
Uint16 sdctl2	SDRAM Control Register 2

Uint16 sdrfr1	SDRAM Refresh Control Register 1
Uint16 sdrfr2	SDRAM Refresh Control Register 2
Uint16 sdext1	SDRAM Extension Register 1
Uint16 sdext2	SDRAM Extension Register 2
Uint16 ce1sec1	CE1 Secondary Control Register 1
Uint16 ce0sec1	CE0 Secondary Control Register 1
Uint16 ce2sec1	CE2 Secondary Control Register 2
Uint16 ce3sec1	CE3 Secondary Control Register 1
Uint16 cescr	CE Size Control Register

Description

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

Example

```
EMIF_Config Config1 = {
   0x06CF, /* egcr   */
   0xFFFF, /* emirst */
   0x7FFF, /* ce01   */
   0xFFFF, /* ce02   */
   0x00FF, /* ce03   */
   0x7FFF, /* ce11   */
   0xFFFF, /* ce12   */
   0x00FF, /* ce13   */
   0x7FFF, /* ce21   */
   0xFFFF, /* ce22   */
   0x00FF, /* ce23   */
   0x7FFF, /* ce31   */
   0xFFFF, /* ce32   */
   0x00FF, /* ce33   */
   0x07FF, /* sdc1   */
   0x0FFF, /* sdper  */
   0x07FF, /* init   */
   0x03FF  /* sdc2   */
}
```

7.3 Functions

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

EMIF_config *Writes value to up EMIF using configuration structure*

Function

```
void EMIF_config(
    EMIF_Config *Config
);
```

Arguments

Config   Pointer to an initialized configuration structure

Return Value

None

Description

Writes a value to up the EMIF using the configuration structure. The values of the structure are written to the port registers.

Example

```
EMIF_Config MyConfig = {
   0x06CF, /* egcr   */
   0xFFFF, /* emirst */
   0x7FFF, /* ce01   */
   0xFFFF, /* ce02   */
   0x00FF, /* ce03   */
   0x7FFF, /* ce11   */
   0xFFFF, /* ce12   */
   0x00FF, /* ce13   */
   0x7FFF, /* ce21   */
   0xFFFF, /* ce22   */
   0x00FF, /* ce23   */
   0x7FFF, /* ce31   */
   0xFFFF, /* ce32   */
   0x00FF, /* ce33   */
   0x07FF, /* sdc1   */
   0x0FFF, /* sdper  */
   0x07FF, /* init   */
   0x03FF  /* sdc2   */ }

EMIF_config(&MyConfig);
```

EMIF_getConfig

Reads the EMIF configuration structure

Function

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

Arguments

Config   Pointer to an initialized configuration structure

Return Value

None

Description

Reads the EMIF configuration in a configuration structure.

Example

```
EMIF_Config myConfig;
EMIF_getConfig(&myConfig);
```

EMIF_enterself-Refresh

Performs self refresh for SDRAM connected to EMIF (5509A only)

Function

```
void EMIF_enterSelfRefresh(
    Uint16 ckePin,
    Uint16 tRasDelay
);
```

Arguments

ckePin — selects which pin to use for CKE
ckePin — 0 selects XF pin
ckePin — 1 selects GPIO.4
tRasDelay — number of CPU cycles to hold memory in refresh

Return Value

None

Description

Performs SDRAM self refresh, given GPIO pin to use toggle for refresh enable, and the minimum number of CPU cycles to hold the memory in refresh.

Example

```
EMIF_enterSelfRefresh(1,1000);
```

EMIF_exitselfRefresh

Exits self refresh for SDRAM connected to EMIF (5509A only)

Function

```
void EMIF_exitSelfRefresh(
    Uint16 tXsrDelay
);
```

Arguments tXsrDelay — number of CPU cycles to wait for refresh to complete before de-asserting refresh enable

Return Value None

Description Exits SDRAM self refresh after waiting tXsrDelay CPU cycles to allow current refresh to complete.

Example

```
EMIF_exitSelfRefresh(1000);
```

EMIF_reset

Resets memory connected in EMIF CE space (5510xx,5509,5509A)

Function

```
void EMIF_reset
    (void
);
```

Arguments None

Return Value None

Description Resets mememory in EMIF CE spaces. Has no effect on EMIF configuration registers. These register retain their current value.

Example

```
EMIF_reset();
```

7.4 Macros

The CSL offers a collection of macros to gain individual access to the EMIF peripheral registers and fields.

Table 7-4 contains a list of macros available for the EMIF module. To use them, include “csl_emif.h.”

Table 7-4. EMIF CSL Macros Using EMIF Port Number

(a) Macros to read/write EMIF register values

Macro	Syntax
EMIF_RGET()	Uint16 EMIF_RGET(*REG*)
EMIF_RSET()	Void EMIF_RSET(*REG*, Uint16 *regval*)

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

Macro	Syntax
EMIF_FGET()	Uint16 EMIF_FGET(*REG*, *FIELD*)
EMIF_FSET()	Void EMIF_FSET(*REG*, *FIELD*, Uint16 *fieldval*)

(c) Macros to create value to EMIF registers and fields (Applies only to registers with more than one field)

Macro	Syntax
EMIF_*REG*_RMK()	Uint16 EMIF_*REG*_RMK(*fieldval_n*,…*fieldval_0*) (see note 5) Note: *Start with field values with most significant field positions: field_n: MSB field field_0: LSB field *only writable fields allowed
EMIF_FMK()	Uint16 EMIF_FMK(*REG*, *FIELD*, *fieldval*) (see note 5)

(d) Macros to read a register address

Macro	Syntax
EMIF_ADDR()	Uint16 EMIF_ADDR(*REG*)

Notes:

1) *REG* indicates the register: EGCR, EMIRST, EMIBE, CE01, CE02, CE03, CE11, CE12, CE13, CE21, CE22, CE23, CE31, CE32, CE33, SDC1, SDPER, SDCNT, INIT, SDC2
2) *FIELD* indicates the register field name as specified in the *55x Peripheral User's Guide*.
   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*).
5) For the special case of the CEx0, CEx1, CEx2, and CEx3, EMIF_REG_RMK(), and EMIF_FMK() both use REG = CEx0, CEx1, CEx2, and CEx3, where x is the letter X

Chapter 8

GPIO Module

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

Topic **Page**

8.1 Overview

The GPIO module is designed to allow central control of the non-multiplexed and address GPIO pins available in the C55x devices. The following three tables list the functions, registers and macros used with this module.

Table 8-1. GPIO Functions

Syntax	Description	See page ...
GPIO_pinDirection	Sets the GPIO pins as either an input or output pin	8-8
GPIO_pinDisable	Disables a pin as a GPIO pin	8-13
GPIO_pinEnable	Enables a pin as a GPIO pin	8-13
GPIO_pinRead	Reads the GPIO pin value	8-14
GPIO_pinWrite	Writes a value to a GPIO pin	8-15
The following functions are supported by C5502 Only.		
GPIO_close	Frees one or more GPIO pins for use	8-5
GPIO_config	Configures GPIO pins	8-7
GPIO_open	Allocates one or more GPIO pins to the current process	8-5
GPIO_pinReadAll	Reads the value of one or more pins	8-14
GPIO_pinWriteAll	Writes the value to one or more pins	8-15
GPIO_pinReset	Resets the value of one or more pins	8-16

Table 8-2. GPIO Registers

Register	Field
IODIR	IO7DIR, IO6DIR, IO5DIR, IO4DIR, IO3DIR, IO2DIR, IO1DIR, IO0DIR
IODATA	IO7D, IO6D, IO5D, IO4D, IO3D, IO2D, IO1D, IO0D
The following registers are supported by c5509 and C5509A.	
AGPIOEN	IO13, IO12, IO11, IO10, IO9, IO8
AGPIODIR	IO13DIR, IO12DIR, IO11DIR, IO10DIR, IO9DIR, IO8DIR
AGPIODATA	IO13D, IO12D, IO11D, IO10D, IO9D, IO8D
The following registers are supported by C5502 Only.	
PGPIOEN0	IO15EN, IO14EN, IO13EN, IO12EN, IO11EN, IO10EN, IO9EN, IO8EN, IO7EN, IO6EN, IO5EN, IO4EN, IO3EN, IO2EN, IO1EN, IO0EN
PGPIODIR0	IO15DIR, IO14DIR, IO13DIR, IO12DIR, IO11DIR, IO10DIR, IO9DIR, IO8DIR, IO7DIR, IO6DIR, IO5DIR, IO4DIR, IO3DIR, IO2DIR, IO1DIR
PGPIODAT0	IO15DAT, IO14DAT, IO13DAT, IO12DAT, IO11DAT, IO10DAT, IO9DAT, IO8DAT, IO7DAT, IO6DAT, IO5DAT, IO4DAT, IO3DAT, IO2DAT, IO1DAT, IO0DAT
PGPIOEN1	IO31EN, IO30EN, IO29EN, IO28EN, IO27EN, IO26EN, IO25EN, IO24EN, IO23EN, IO22EN, IO21EN, IO20EN, IO19EN, IO18EN, IO17EN, IO16EN
PGPIODIR1	IO31DIR, IO30DIR, IO29DIR, IO28DIR, IO27DIR, IO26DIR, IO25DIR, IO24DIR, IO23DIR, IO22DIR, IO21DIR, IO20DIR, IO19DIR, IO18DIR, IO17DIR, IO16DIR
PGPIODAT1	IO31DAT, IO30DAT, IO29DAT, IO28DAT, IO27DAT, IO26DAT, IO25DAT, IO24DAT, IO23DAT, IO22DAT, IO20DAT, IO19DAT, IO18DAT, IO17DAT, IO16DAT
PGPIOEN2	IO45EN, IO44EN, IO43EN, IO42EN, IO41EN, IO40EN, IO39EN, IO38EN, IO37EN, IO36EN, IO35EN, IO34EN, IO33EN, IO32EN
PGPIODIR2	IO45DIR, IO44DIR, IO43DIR, IO42DIR, IO41DIR, IO40DIR, IO39DIR, IO38DIR, IO37DIR, IO36DIR, IO35DIR, IO34DIR, IO33DIR, IO32DIR
PGPIODAT2	IO45DAT, IO44DAT, IO43DAT, IO42DAT, IO41DAT, IO40DAT, IO39DAT, IO38DAT, IO37DAT, IO36DAT, IO35DAT, IO34DAT, IO33DAT, IO32DAT

Note: R = Read Only; W = Write; By default, most fields are Read/Write

8.2 Configuration Structure

The following is the configuration structure used to set up the GPIO.

GPIO_Config *Configuration structure for non-parallel GPIO pins*

Structure GPIO_Config

Members

Uint16 ioen	Pin Enable Register	IOEN
Uint16 iodir	Pin Direction Register	IODIR

Description The GPIO configuration structure is used to set up the non-parallel GPIO pins. You create and initialize this structure and then pass its address to the GPIO_config() function. You can use literal values or the GPIO_RMK macros to create the structure member values.

GPIO_ConfigAll *Configuration structure for both parallel and non-parallel GPIO pins*

Structure GPIO_ConfigAll

Description The GPIO configuration structure is used to set up both non-parallel and parallel GPIO pins. You create and initialize this structure and then pass its address to the GPIO_ConfigAll() function. You can use literal values or the GPIO_RMK macros to create the structure member values.

Members

Uint16 ioen	Non-parallel GPIO pin enable register	IOEN
Uint16 iodir	Non-parallel GPIO pin direction register	IODIR
Uint16 pgpioen	Parallel GPIO pin enable register 0	PGPIOEN0
Uint16 pgpiodir	Parallel GPIO pin direction register 0	PGPIODIR0
Uint16 pgpioen1	Parallel GPIO pin enable register 1	PGPIOEN1
Uint16 pgpiodir1	Parallel GPIO pin direction register 1	PGPIODIR1
Uint16 pgpioen2	Parallel GPIO pin enable register 2	PGPIOEN2
Uint16 pgpiodir2	Parallel GPIO pin direction register 2	PGPIODIR2

8.3 Functions

The following are functions available for the GPIO module. They are supported by C5502 only.

GPIO_close — *Frees GPIO pins previously reserved by call to GPIO_open()*

Function void GPIO_close(GPIO_Handle hGpio);

Arguments hGpio GPIO pin Handle (see GPIO_open()).

Return Value None

Description Frees GPIO pins previously reserved in call to GPIO_open().

Example

```
GPIO_close(hGpio);
```

GPIO_open — *Reserves GPIO pin for exclusive use*

Function GPIO_Handle GPIO_open(Uint32 allocMask, Uint32 flags);

Arguments

allocMask	GPIO pins to reserve. For list of pins, please see GPIO_pinDirection().
flags	Open flags , currently non defined.

Return Value GPIO_Handle Device handle

Description

Before a GPIO pin can be used, it must be reserved for use by the application. Once reserved, it cannot be requested again until, closed by GPIO_close(). The return value is a unique device handle that is used in subsequent GPIO API calls. If the function fails, INV (-1) is returned.

For C5502, there are four groups of GPIO pins. (See GPIO_pinDirection() for list of pins in each group).

GPIO_open() must be called to open one or more pins of only one group at a time. Calling the allocMask of pins in different groups will produce unknown results.

Example: The first parameter to GPIO_open() could be
(GPIO_GPIO_PIN4 | GPIO_GPIO_PIN2
as they are in the same group, but
(GPIO_GPIO_PIN4 | GPIO_PGPIO_PIN2)
will produce unknown results.

If GPIO_open() is called for one or more pins in a particular group, it cannot be called again to open other pins of the same group unless corresponding GPIO_close() is called. However, GPIO_open() can be called again to open one or more pins of another group.

Example: If GPIO_open() is called for the first time with GPIO_GPIO_PIN4 as the first parameter, it can not be called again with GPIO_GPIO_PIN2 parameter, as they belong to the same pin group. However, it can be called again with GPIO_PGPIO_PIN2 as the first parameter.

Example

```
GPIO_Handle hGPIO;
hGPio = GPIO_open(GPIO_PGPIO_PIN1,0);
```

GPIO_config *Writes value to non-parallel registers using GPIO_config*

Function void GPIO_config(GPIO_Handle hGpio,
GPIO_Config *cfg);

Arguments

hGpio	GPIO Device handle
cfg	Pointer to an initialized configuration structure

Return Value None

Description Writes values to the non-parallel GPIO control registers using the configuration structure.

Note: GPIO_Config structure is common for GPIO and PGPIO pins. The GPIO_config() function just discards the enable field in case of GPIO [0:7] pins.

Example

```
GPIO_Handle hGpio;
GPIO_Config myConfig = {GPIO_PIN1_OUTPUT |
                         GPIO_PIN3_OUTPUT
                         }
configuration for 5502

hGpio = GPIO_open(GPIO_GPIO_PIN1 | GPIO_GPIO_PIN3,0);
                  GPIO_config(hGpio &myConfig);
```

GPIO_configAll *Writes value to both non-parallel and parallel GPIO control registers*

Function void GPIO_config(GPIO_ConfigAll &gCfg);

Arguments

gCfg	Configuration structure for both power and non-power, non-muxedGPIO pins.

Return Value None

Description Writes values to both parallelowe and non-parallel GPIO control registers using the configuration structure. See also GPIO_ConfigAll.

Example

```
GPIO_ConfigAll gCfg = {
              GPIO_PIN1_OUTPUT | GPIO_PIN3_OUTPUT, /* IODIR */
              0, /* PGPIOEN0 */
              0, /* PGPIODIR0 */
              0, /* PGPIOEN1 */
              0, /* PGPIODIR1 */
              0, /* PGPIOEN2 */
              0  /* PGPIODIR2 */
          };
/* GPIO configuration for 5502 */
GPIO_configAll(&gCfg);
```

GPIO_pinDirection *Sets the GPIO pin as either an input oroutpit pin*

Function

For C5502 only:
```
void GPIO_pinDirection(GPIO_Handle hGpio,
                Uint32 pinMask,
                Uint16 direction);
```
For C5509/C5509A/C5510:
```
void GPIO_pinDirection(Uint32 pinMask,
                Uint16 direction);
```

Arguments

hGPIO	GPIO Handle returned from previous call to GPIO_open() (This argument is only for C5502 CSL)
pinMask	GPIO pins affected by direction

For 5502 pinMask may be any of the following:

GPIO Pin Group 0 (Non-Parallel GPIO Pins):

GPIO_GPIO_PIN0
GPIO_GPIO_PIN1
GPIO_GPIO_PIN2
GPIO_GPIO_PIN3
GPIO_GPIO_PIN4
GPIO_GPIO_PIN5
GPIO_GPIO_PIN6
GPIO_GPIO_PIN7

GPIO Pin Group 1 (Parallel GPIO Pins 0-15):

GPIO_PGPIO_PIN0
GPIO_PGPIO_PIN1
GPIO_PGPIO_PIN2
GPIO_PGPIO_PIN3
GPIO_PGPIO_PIN4
GPIO_PGPIO_PIN5
GPIO_PGPIO_PIN6
GPIO_PGPIO_PIN7
GPIO_PGPIO_PIN8
GPIO_PGPIO_PIN9
GPIO_PGPIO_PIN10
GPIO_PGPIO_PIN11
GPIO_PGPIO_PIN12
GPIO_PGPIO_PIN13

GPIO_PGPIO_PIN14
GPIO_PGPIO_PIN15

GPIO Pin Group 2 (Parallel GPIO Pins 16-31):
GPIO_PGPIO_PIN16
GPIO_PGPIO_PIN17
GPIO_PGPIO_PIN18
GPIO_PGPIO_PIN19
GPIO_PGPIO_PIN20
GPIO_PGPIO_PIN21
GPIO_PGPIO_PIN22
GPIO_PGPIO_PIN23
GPIO_PGPIO_PIN24
GPIO_PGPIO_PIN25
GPIO_PGPIO_PIN26
GPIO_PGPIO_PIN27
GPIO_PGPIO_PIN28
GPIO_PGPIO_PIN29
GPIO_PGPIO_PIN30
GPIO_PGPIO_PIN31

GPIO Pin Group 3 (Parellel GPIO Pins 32-45):
GPIO_PGPIO_PIN32
GPIO_PGPIO_PIN33
GPIO_PGPIO_PIN34
GPIO_PGPIO_PIN35
GPIO_PGPIO_PIN36
GPIO_PGPIO_PIN37
GPIO_PGPIO_PIN38
GPIO_PGPIO_PIN39
GPIO_PGPIO_PIN40
GPIO_PGPIO_PIN41
GPIO_PGPIO_PIN42
GPIO_PGPIO_PIN43
GPIO_PGPIO_PIN44
GPIO_PGPIO_PIN45

The pinMask may be formed by using a single pin Id listed above or you may combine pin IDs from pins within the same group (i.e., GPIO_PGPIO_PIN23 | GPIO_PGPIO_PIN30)

direction Mask used to set pin direction for pins selected in pinMask

GPIO Pin Group 0 (Non-Parallel GPIO Pins):

GPIO_GPIO_PIN0_OUTPUT
GPIO_GPIO_PIN1_OUTPUT
GPIO_GPIO_PIN2_OUTPUT
GPIO_GPIO_PIN3_OUTPUT
GPIO_GPIO_PIN4_OUTPUT
GPIO_GPIO_PIN5_OUTPUT
GPIO_GPIO_PIN6_OUTPUT
GPIO_GPIO_PIN7_OUTPUT

GPIO_GPIO_PIN0_INPUT
GPIO_GPIO_PIN1_INPUT
GPIO_GPIO_PIN2_INPUT
GPIO_GPIO_PIN3_INPUT
GPIO_GPIO_PIN4_INPUT
GPIO_GPIO_PIN5_INPUT
GPIO_GPIO_PIN6_INPUT
GPIO_GPIO_PIN7_INPUT

GPIO Pin Group 1 (Parallel GPIO Pins 0-15):

GPIO_PGPIO_PIN0_OUTPUT
GPIO_PGPIO_PIN1_OUTPUT
GPIO_PGPIO_PIN2_OUTPUT
GPIO_PGPIO_PIN3_OUTPUT
GPIO_PGPIO_PIN4_OUTPUT
GPIO_PGPIO_PIN5_OUTPUT
GPIO_PGPIO_PIN6_OUTPUT
GPIO_PGPIO_PIN7_OUTPUT
GPIO_PGPIO_PIN8_OUTPUT
GPIO_PGPIO_PIN9_OUTPUT
GPIO_PGPIO_PIN10_OUTPUT
GPIO_PGPIO_PIN11_OUTPUT
GPIO_PGPIO_PIN12_OUTPUT
GPIO_PGPIO_PIN13_OUTPUT
GPIO_PGPIO_PIN14_OUTPUT
GPIO_PGPIO_PIN15_OUTPUT

GPIO_PGPIO_PIN0_INPUT
GPIO_PGPIO_PIN1_INPUT
GPIO_PGPIO_PIN2_INPUT
GPIO_PGPIO_PIN3_INPUT

GPIO_PGPIO_PIN4_INPUT
GPIO_PGPIO_PIN5_INPUT
GPIO_PGPIO_PIN6_INPUT
GPIO_PGPIO_PIN7_INPUT
GPIO_PGPIO_PIN8_INPUT
GPIO_PGPIO_PIN9_INPUT
GPIO_PGPIO_PIN10_INPUT
GPIO_PGPIO_PIN11_INPUT
GPIO_PGPIO_PIN12_INPUT
GPIO_PGPIO_PIN13_INPUT
GPIO_PGPIO_PIN14_INPUT
GPIO_PGPIO_PIN15_INPUT

GPIO Pin Group 2 (Parallel GPIO Pins 16-31):
GPIO_PGPIO_PIN16_OUTPUT
GPIO_PGPIO_PIN17_OUTPUT
GPIO_PGPIO_PIN18_OUTPUT
GPIO_PGPIO_PIN19_OUTPUT
GPIO_PGPIO_PIN20_OUTPUT
GPIO_PGPIO_PIN21_OUTPUT
GPIO_PGPIO_PIN22_OUTPUT
GPIO_PGPIO_PIN23_OUTPUT
GPIO_PGPIO_PIN24_OUTPUT
GPIO_PGPIO_PIN25_OUTPUT
GPIO_PGPIO_PIN26_OUTPUT
GPIO_PGPIO_PIN27_OUTPUT
GPIO_PGPIO_PIN28_OUTPUT
GPIO_PGPIO_PIN29_OUTPUT
GPIO_PGPIO_PIN30_OUTPUT
GPIO_PGPIO_PIN31_OUTPUT

GPIO_PGPIO_PIN16_INPUT
GPIO_PGPIO_PIN17_INPUT
GPIO_PGPIO_PIN18_INPUT
GPIO_PGPIO_PIN19_INPUT
GPIO_PGPIO_PIN20_INPUT
GPIO_PGPIO_PIN21_INPUT
GPIO_PGPIO_PIN22_INPUT
GPIO_PGPIO_PIN23_INPUT
GPIO_PGPIO_PIN24_INPUT
GPIO_PGPIO_PIN25_INPUT
GPIO_PGPIO_PIN26_INPUT

GPIO_PGPIO_PIN27_INPUT
GPIO_PGPIO_PIN28_INPUT
GPIO_PGPIO_PIN29_INPUT
GPIO_PGPIO_PIN30_INPUT
GPIO_PGPIO_PIN31_INPUT

GPIO Pin Group 3 (Parellel GPIO Pins 32-45):

GPIO_PGPIO_PIN32_OUTPUT
GPIO_PGPIO_PIN33_OUTPUT
GPIO_PGPIO_PIN34_OUTPUT
GPIO_PGPIO_PIN35_OUTPUT
GPIO_PGPIO_PIN36_OUTPUT
GPIO_PGPIO_PIN37_OUTPUT
GPIO_PGPIO_PIN38_OUTPUT
GPIO_PGPIO_PIN39_OUTPUT
GPIO_PGPIO_PIN40_OUTPUT
GPIO_PGPIO_PIN41_OUTPUT
GPIO_PGPIO_PIN42_OUTPUT
GPIO_PGPIO_PIN43_OUTPUT
GPIO_PGPIO_PIN44_OUTPUT
GPIO_PGPIO_PIN45_OUTPUT

GPIO_PGPIO_PIN32_INPUT
GPIO_PGPIO_PIN33_INPUT
GPIO_PGPIO_PIN34_INPUT
GPIO_PGPIO_PIN35_INPUT
GPIO_PGPIO_PIN36_INPUT
GPIO_PGPIO_PIN37_INPUT
GPIO_PGPIO_PIN38_INPUT
GPIO_PGPIO_PIN39_INPUT
GPIO_PGPIO_PIN40_INPUT
GPIO_PGPIO_PIN41_INPUT
GPIO_PGPIO_PIN42_INPUT
GPIO_PGPIO_PIN43_INPUT
GPIO_PGPIO_PIN44_INPUT
GPIO_PGPIO_PIN45_INPUT

Direction may be set using any of the symbolic constant defined above. Direction for multiple pins within the same group may be set by OR'ing together several constants:

GPIO_PGPIO_PIN45_INPUT | GPIO_PGPIO_PIN40_OUTPUT

Return Value None

Description Sets the direction for oneor more General purpose I/O pins (input or output)

Example

```
/* sets the pin pgpio1 as an input */
GPIO_handle hGpio = GPIO_open(GPIO_PGPIO_PIN1|GPIO_PGPIO_PIN15);
GPIO_pinDirection(hGPio, GPIO_PGPIO_PIN1, GPIO_PGPIO_PIN1_INPUT);
```

GPIO_pinDisable *Disables a pin as a GPIO pin*

Function

For C5502 only:
void GPIO_pinDisable(GPIO_Handle hGpio, Uint32 pinId)
For C5509/C5509A/C5510:
void GPIO_pinDisable((Uint32 pinId)

Arguments

hGpio	GPIO handle returned from previous call to GPIO_open (This argument is only for C5502 CSL)
pinID	IDs of the pins to disable. Please see GPIO_pinDirection() for list of possible pin IDs.

Return Value None

Description Disables one or more pins as GPIO pins.

Example

```
/* disables pin pgpio1 as a GPIO pin */
GPIO_handle hGpio = GPIO_open(GPIO_PGPIO_PIN1|GPIO_PGPIO_PIN15);
GPIO_pinDisable (hGpio,GPIO_PGPIO_PIN1);
/* disables parallel pin IO1 as GPIO */
```

GPIO_pinEnable *Enables a pin as a GPIO pin*

Function

For C5502 only:
void GPIO_pinEnable(GPIO_Handle hGpio, Uint32 pinId)
For C5509/C5509A/C5510:
void GPIO_pinEnable(Uint32 pinId)

Arguments

hGpio	GPIO Handle returned from call to GPIO_open(). (This argument is only for C5502 CSL)
pinID	ID of the pin to enable. For valid pin IDs, please see GPIO_pinDirection().

Return Value None

Description Enables a pin as a general purpose I/O pin.

Example

```
GPIO_pinEnable (hGpio, GPIO_GPIO_PIN1);
/* enables pin IO1 as GPIO */
```

GPIO_pinRead *Reads a GPIO pin value*

Function

For C5502 only:
int GPIO_pinRead(GPIO_Handle hGpio, Uint32 pinId)
For C5509/C5509A/C5510
int GPIO_pinRead(Uint32 pinId)

Arguments

hGPio	GPIO Handle returned from previous call to GPIO_open(). (This argument is only for C5502 CSL)
pinId	IDs of the GPIO pins to read.

Return Value

Value	Value read in GPIO pin (1 or 0)

Description

Reads the value in a general purpose input pin.

Example

```
int val;
val = GPIO_pinRead (hGPio,GPIO_GPIO_PIN1);
/* reads IO1 pin value */
```

GPIO_pinReadAll *Reads a value of one or more GPIO pins*

Function

For C5502 only:
int GPIO_pinReadAll(GPIO_Handle hGpio, Uint32 pinMask)
For C5509/C5509A/C5510
int GPIO_pinReadAll(Uint32 pinMask)

Arguments

hGPio	GPIO Handle returned from previous call to GPIO_open(). (This argument is only for C5502 CSL)
pinMask	IDs of the GPIO pins to read. Please see GPIO_pinDirection() for list of pin IDs.

Return Value

Value	Value read in GPIO pin/s

Description

Reads in the value of the GPIO pins specified by pinMask. The function returns the value in place of the pins. It does not right-justify the value to return a raw result.

Example

```
int val;
/* reads IO0 and IO7 pin values */
val=GPIO_pinRead (hGPio,GPIO_GPIO_PIN0| GPIO_GPIO_PIN7);
```

GPIO_pinWrite *Writes a value to a GPIO pin*

Function

For C5502 only:
void GPIO_pinWrite(GPIO_Handle hGpio,
Uint32 pinMask,
Uint16 val)
For C5509/C5509A/C5510:
void GPIO_pinWrite(Uint32 pinMask Uint16 val)

Arguments

hGpio	GPIO Handle returned from previous call to GPIO_open(). (This argument is only for C5502 CSL)
pinMask	ID of one or more GPIO pins to write. Please see GPIO_pinDirection for a list of valid pin IDs.
val	Value (0 or 1) to write to selected GPIO pins.

Return Value

None

Description

Writes a value to a general purpose output pin.

Example

```
/* writes 1 to IO pin0 and IO pin 5 */
GPIO_pinWrite (hGpio, GPIO_GPIO_PIN0 | GPIO_GPIO_PIN5, 1);
```

GPIO_pinWriteAll *Writes a value to one or more GPIO pins*

Function

For C5502 only:
void GPIO_pinWriteAll(GPIO_Handle hGpio,
Uint32 pinMask,
Uint16 val)
For C5509/C5509A/C5510:
void GPIO_pinWriteAll(Uint32 pinMask,
Uint16 val)

Arguments

hGpio	GPIO Handle returned from previous call to GPIO_open(). (This argument is only for C5502 CSL)
pinMask	ID of one or more GPIO pins to write. Please see GPIO_pinDirection for a list of valid pin IDs.
val	Value mask to write to selected GPIO pins.

Return Value

None

Description

Writes a value to one or more general purpose output pins. Ths function assumes an in-place value mask for writing to the GPIO pins. It will not left-justify values.

Example

```
/* writes 1 to IO pin0 and IO pin 5 */
GPIO_pinWrite (hGpio,GPIO_GPIO_PIN0| GPIO_GPIO_PIN5,0x0021);
```

GPIO_pinReset *Resets GPIO pins to default values*

Function void GPIO_pinReset(GPIO_Handle hGpio, Uint32 pinMask)

Arguments

hGpio	GPIO Handle returned from previous call to GPIO_open().
pinMask	ID of one or more GPIO pins to write. Please see GPIO_pinDirection for list of valid pin IDs.

Return Value None

Description Restores selected GPIO pins to default value of 0.

Example

```
/* writes 1 to IO pin1 and IO pin 3 */
   GPIO_pinReset (hGpio, GPIO_GPIO_PIN1 | GPIO_GPIO_PIN3);
```

8.4 Macros

The CSL offers a collection of macros to gain individual access to the GPIO peripheral registers and fields.

Table 8-3 contains a list of macros available for the GPIO module. To use them, include “csl_gpio.h.”

Table 8-3. GPIO CSL Macros

(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 create value to GPIO registers and fields (Applies 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*)

Notes:
1) *REG* include the registers IODIR, IODATA, GPIODIR, GPIODATA, GPIOEN, AGPIODIR, AGPIODATA, and AGPIOEN.
2) *FIELD* indicates the register field name
   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*).

Chapter 9

HPI Module

This chapter describes the HPI module, lists the API structure, macros, functions, and provides an HPI API reference. The HPI module applies to the C5502 device only.

Topic **Page**

9.1 Overview

This module enables configuration of the 5502 HPI. The HPI module is not handle based. Configuration of the HPI is easily accomplished by calling HPI_config() or any of the SET register macros. Using HPI_config() is the preferred method for configuration.

Table 9-1 Lists the configuration structure for HPI modules

Table 9-2 Lists the function APIs

Table 9-3 Lists the register and bit field names

Lists the API macros

Table 9-1. HPI Module Configuration Structure

Syntax	Description	See page ...
HPI_Config	HPI module configuration structure	9-4

Table 9-2. HPI Functions

Syntax	Description	See page ...
HPI_config()	Sets up HPI using configuration structure (HPI_Config)	9-5
HPI_getConfig()	Returns current HPI control register values in a configuration structure (HPI_Config)	9-5

Table 9-3. HPI Registers and Bit Field Names

Register	Field
HGPIOEN	EN0, EN1, EN2, EN4, EN6, EN7, EN8, EN9, EN11, EN12
HGPIODIR	HDn(n=0-15)
HGPIODAT	HDn(n=0-15)
HPIC	HPIASEL, DUALHPIA, BOBSTAT, HPIRST, FETCH, HRDY, HINT, DSPINT, BOB
HPIAW	HPIAW
HPIAR	HPIAR
HPWREMU	FREE, SOFT

Table 9-4. HPI Macros

Syntax	Description	See page ...
HPI_ADDR	Get the address of a given register	9-6
HPI_FGET	Gets value of a register field	9-6
HPI_FMK	Creates register value based on individual field value	9-7
HPI_FSET	Sets value of register field	9-7
HPI_REG_RMK	Creates register value based on individual field values	9-8
HPI_RGET	Gets the value of an HPI register	9-9
HPI_RSET	Set the value of an HPI register	9-9

9.2 Configuration Structures

The following is the HPI configuration structure used to set up the HPI interface.

HPI_Config

HPI configuration structure used to set up HPI interface

Structure

HPI_Config

Members

Uint16	hpwremu	HPI power/emulation management register
Uint16	hgpioen	HPI GPIO pin enable register
Uint16	hgpiodir	HPI GPIO pin direction register
Uint16	hpic	HPI Control register

9.3 Functions

The following are functions available for the HPI module.

HPI_config

Writes to HPI registers using values in configuration structure

Function

```
void HPI_config(
    HPI_Config *myConfig
);
```

Arguments myConfig Pointer to an initialized configuration structure

Return Value None

Description Writes the values given in the initialized configuration structure to the corresponding HPI control register. See HPI_Config.

Example

```
HPI_Config myConfig = {0x3,     /* HPWREMU , Select FREE = SOFT = 1  */
                  0x0,     /* HGPIOEN , Disable all GPIO pins       */
                  0x0,     /* HGPIODIR, Default GPIO pins to output */
                  0x80     /* HPIC    , Reset HPI                   */
                  };
                  HPI_config(&myConfig);
```

HPI_getConfig

Reads current HPI configuration

Function

```
void HPI_getConfig(
    HPI_Config *myConfig
);
```

Arguments myConfig Pointer to an initialized configuration structure

Return Value None

Description Reads the curent values of the HPI control registers, returning those values in the given configuration structure. See HPI_config

Example

```
HPI_Config myConfig;
HPI_getConfig(&myConfig);
```

9.4 Macros

The following is a listing of HPI macros.

HPI_ADDR *Gets address of given register*

Macro HPI_ADDR(REG)

Function

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

Arguments REG register as listed in HPI_RGET()

Return Value Address of Register

Description Gets the address of an HPI register

Example

```
ioport Uint16 *hpi_ctl;
hpi_ctl = HPI_ADDR(HPIC);
```

HPI_FGET *Gets the value of register field*

Macro HPI_FGET(REG,FIELD)

Arguments

REG register as listed in HPI_RGET()
FIELD symbolic name for field of register REG.
Possible values: All field names are listed in the TMS320VC5501/5502 DSP Host Port Interface (HPI) Reference Guide (SPRU620A)

Return Value Value of register field

Description Gets current value of register field

Example

```
Uint16 bob = HPI_FGET(HPIC,BOB);
```

HPI_FMK

Creates register value based on individual field value

Macro

HPI_FMK(REG,FIELD,fieldval)

Arguments

REG register as listed in HPI_RGET()
FIELD symbolic name for field of register REG.
Possible values: All field names are listed in the TMS320VC5501/5502 DSP Host Port Interface (HPI) Reference Guide (SPRU620A)

Return Value

Shifted version of fieldval. Value is shifted to appropriate bit position 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 HPI_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

```
unt16 gpioenMask = HPI_FMK(HGPIOEN,EN2,1) |
                   HPI_FMK(HGPIOEN,EN8,1);
```

HPI_FSET

Sets the value of register field

Macro

Void HPI_FSET (REG, FIELD, fieldval)

Arguments

REG Only writable registers containing more than one field are supported by this macro.
FIELD symbolic name for field of register REG.
Possible values: All writeable field names are listed in the TMS320VC5501/5502 DSP Host Port Interface (HPI) Reference Guide (SPRU620A)

Return Value

None

Description

Sets the HPI register field value to fieldval.

Example

```
HPI_FSET(HGPIOEN,EN0,1);
```

HPI_REG_RMK *Creates register value based on individual field values*

Macro Uint16 HPI_REG_RMK (fieldval_n,...,fieldval_0)

Arguments REG Only writable registers containing more than one field are supported by this macro.

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 constant
- 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 HPI register value given specific field values. You can use constants or the CSL symbolic constants covered in Section 1.6.

Example

```
Uint16 myregval;
/* enable HA[0:7], HD[8:15], HD[0:7] for GPIO */
myregval = HPI_HGPIOEN_RMK (0,1,1,1,0,0,0,0,0);
```

HPI_REG_RMK are typically used to initialize a HPI configuration structure used for the HPI_config() function (see section 9.2).

HPI_RGET *Gets value of an HPI register*

Macro Uint16 HPI_RGET (REG)

Arguments REG where:
REG is one of the following

- HGPIOEN
- HGPIODIR
- HPIAR
- HPIAW
- HPWREMU
- HPIC

Return Value Value of register

Description Returns the HPI register value

Example

```
Uint16 myvar;
myVar = HPI_RGET(HPIC); /*read HPI control register */
```

HPI_RSET *Sets value of an HPI register*

Macro Void HPI_RSET (REG, Uint16 regval)

Arguments

REG	register, as listed in HPI_RGET() macro
regval	register value that wants to write to register REG

Return Value None

Description Sets the HPI register REG value to regval

Example

```
HPI_RSET(HPWREMU, 0x3); /* Set FREE and SOFT bits */
```

CSL offers a collection of macros to gain individual access to the GPIO peripheral registers and fields.

Table 8-3 contains a list of macros available for the GPIO module. To use them, include “csl_gpio.h.”

Chapter 10

I2C Module

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

Topic **Page**

10.1 Overview

The configuration of the I2C can be performed by using one of the following methods:

Register-based configuration

A register-based configuration can be performed by calling either I2C_config() or any of the SET register field macros.

Parameter-based configuration (Recommended)

A parameter-based configuration can be performed by calling I2C_setup(). Using I2C_setup() to initialize the I2C registers is the recommended approach.

Compared to the register-based approach, this method provides a higher level of abstraction. The downside is larger code size and higher cycle counts.

Table 10-3 lists DMA registers and fields.

Table 10-1. I2C Configuration Structure

Configuration Structure	Description	See page…
I2C_Config	I2C configuration structure used to set up the I2C (register-based)	10-5
I2C_Setup	Sets up the I2C using the initialization structure	10-6

Table 10-2. I2C Functions

Functions	Description	See page…
I2C_config()	Sets up the I2C using the configuration structure	10-7
I2C_eventDisable()	Disables the I2C interrupt specified.	10-8
I2C_eventEnable()	Enables the I2C interrupt specified.	10-8
I2C_getConfig()	Obtains the current configuration of all the I2C registers	10-8
I2C_getEventId()	Returns the I2C IRQ event ID	10-9
I2C_setup()	Sets up the I2C using the initialization structure	10-9
I2C_IsrAddr	I2C structure containing pointers to functions that will be executed when a specific I2C interrupt is enabled and received.	10-10

Table 10-2. I2C Functions (Continued)

Functions	Description	See page…
I2C_read()	Performs master/slave receiver functions	10-10
I2C_readByte()	Performs a read from the data receive register (I2CDRR).	10-11
I2C_reset()	Sets the IRS bit in the I2CMDR register to 1 (performs a reset).	10-12
I2C_rfull()	Reads the RSFULL bit in the I2CSTR register.	10-12
I2C_rrdy()	Reads the I2CRRDY bit in the I2CSTR register.	10-12
I2C_sendStop()	Sets the STP bit in the I2CMDR register (generates a stop).	10-13
I2C_setCallback()	Associates each callback function to one of the I2C interrupt events and installs the I2C dispatcher table.	10-13
I2C_start()	Sets the STT bit in the I2CMDR register (generates a start).	10-14
I2C_write()	Performs master/slave transmitter functions	10-14
I2C_writeByte()	Performs a write to the data transmit register (I2CDXR).	10-15
I2C_xempty()	Reads the XSMT bit in theI2CSTR register.	10-16
I2C_xrdy()	Reads the I2CXRDY bit in the I2CSTR register.	10-16

10.1.1 I2C Registers

Table 10-3. I2C Registers

Register	Field
I2COAR	OAR
I2CIER	AL , NACK , ARDY , RRDY , XRDY
I2CSTR	(R)AL, (R)NACK, (R)ARDY, RRDY, (R)XRDY, (R)AD0, (R)AAS, (R)XSMT, (R)RSFULL ,(R)BB
I2CCLKL	ICCL
I2CCLKH	ICCH
I2CCNT	ICDC
I2CDRR	(R)DATA
I2CSAR	SAR
I2CDXR	(R)DATA
I2CMDR	BC, FDF, STB, IRS, DLB, RM, XA, TRX, MST, STP, IDLEEN , STT, FREE
I2CISRC	(R)INTCODE, TESTMD
I2CGPIO	
I2CPSC	IPSC

Note: R = Read Only; W = Write; By default, most fields are Read/Write

10.2 Configuration Structures

The following are the configuration structures used to set up the I2C module.

I2C_Config *I2C Configuration Structure used to set up the I2C interface*

Structure I2C_Config

Members

Uint16 i2coar	Own address register
Uint16 i2cier	Interrupt mask/status register
Uint16 i2cstr	Interrupt status register
Uint16 i2cclkl	Clock Divider Low register
Uint16 i2cclkh	Clock Divider High register
Uint16 i2ccnt	Data Count register
Uint16 i2csar	Slave Address register
Uint16 i2cmdr	Mode register
Uint16 i2cisrc	Interrupt source vector register
Uint16 i2cpsc	Prescaler register

Description I2C configuration structure used to set up the I2C interface. You create and initialize this structure and then pass its address to the I2C_config() function. You can use either literal values, or I2C_RMK macros to create the structure member values.

Example

```
I2C_Config Config = {
0xFFFF,    /* I2COAR  */
0x0000,    /* I2CIER  */
0xFFFF,    /* I2CSTR  */
10,        /* I2CCLKL */
8,         /* I2CCLKH */
1,         /* I2CCNT  */
0xFFFA,    /* I2CSAR  */
0x0664,    /* I2CMDR  */
0xFFFF,    /* I2CISRC */
0x0000     /* I2CPSC  */
}
```

I2C_Setup *I2C Initialization Structure used to set up the I2C interface*

Structure I2C_Setup

Members

Uint16 addrmode	Address Mode: 0 = 7 bit 1 = 10 bit
Uint16 ownaddr	Own Address (I2COAR)
Uint16 sysinclock	System Clock Value (MHz)
Uint16 rate	Desired Transfer rate (10-400 kbps)
Uint16 bitbyte	Number of bits per byte to be received or transmitted:
Uint16 dlb	Data Loopback mode 0 = off, 1 = on
Uint16 free	emulator FREE mode 0 = off, 1 = on

bitbyte values:

Value	Bits/byte transmitted/received
0	8
1	1
2	2
3	3
4	4
5	5
6	6
7	7

Description I2C initialization structure used to set up the I2C interface. You create and initialize this structure and then pass its address to the I2C_setup() function.

Example

```
I2C_Setup Setup = {
0,          /* 7 or 10 bit address mode              */
0x0000,     /* own address - don't care if master    */
144,        /* clkout value (Mhz)                    */
400,        /* a number between 10 and 400           */
0,          /* number of bits/byte to be received or */
            /* transmitted (8 bits)                  */
0,          /* DLB mode                              */
1           /* FREE mode of operation                */
}
```

10.3 Functions

The following are functions available for use with the I2C module.

I2C_config — *Sets up the I2C using the configuration structure*

Function void I2C_config (I2C_Config *Config);

Arguments Config Pointer to an initialized configuration structure

Return Value none

Description Writes a value to set up the I2C using the configuration structure. The values of the configuration structure are written to the port registers.

If desired, you can configure all I2C registers with:

```
I2C_config(); [maintaining I2CMDR(STT)=0]
```

and later, use the I2C_start() function to start the I2C peripheral

Example

```
I2C_Config Config = {
0xFFFF,    /* I2COAR  */
0x0000,    /* I2CIER  */
0xFFFF,    /* I2CSTR  */
10,        /* I2CCLKL */
8,         /* I2CCLKH */
1,         /* I2CCNT  */
0xFFFA,    /* I2CSAR  */
0x0664,    /* I2CMDR  */
0xFFFF,    /* I2CSRC  */
0x0000     /* I2CPSC  */
};

I2C_config(&Config);
```

I2C_eventDisable *Disables the interrupt specified by the ierMask*

Function void I2C_eventDisable(Uint16 isrMask);

Arguments isrMask can be one or the logical OR any of the following:

I2C_EVT_AL	// Arbitration Lost Interrupt Enable
I2C_EVT_NACK	// No Acknowledgement Interrupt Enable
I2C_EVT_ARDY	// Register Access Ready Interrupt
I2C_EVT_RRDY	// Data Receive Ready Interrupt
I2C_EVT_XRDY	// Data Transmit Ready Interrupt

Description This function disables the interrupt specified by the ierMask.

Example

```
I2C_eventDisable(I2C_EVT_RRDY);
...
I2C_eventDisable (I2C_EVT_RRDY|I2C_EVT_XRDY);
```

I2C_eventEnable *Enables the I2C interrupt specified by the isrMask*

Function void I2C_eventEnable(Uint16 isrMask);

Arguments isrMask can be one or a logical OR of the following:

I2C_EVT_AL	// Arbitration Lost Interrupt Enable
I2C_EVT_NACK	// No Acknowledgement Interrupt Enable
I2C_EVT_ARDY	// Register Access Ready Interrupt
I2C_EVT_RRDY	// Data Receive Ready Interrupt
I2C_EVT_XRDY	// Data Transmit Ready Interrupt

Description This function enables the I2C interrupts specified by the isrMask.

Example

```
I2C_eventEnable(I2C_EVT_AL);
...
I2C_eventEnable (I2C_EVT_RRDY|I2C_EVT_XRDY);
```

I2C_getConfig *Writes values to I2C registers using the configuration strucucture*

Function void I2C_getConfig (I2C_Config *Config);

Arguments Config Pointer to a configuration structure

Return Value None

Description Reads the current value of all I2C registers being used and places them into the corresponding configuration structure member.

Example

```
I2C_Config *testConfig;
I2C_getConfig(testConfig);
```

I2C_getEventId — *Returns the I2C software interrupt value*

Function

int I2C_getEventId(
);

Arguments

None

Description

Returns the I2C software interrupt value.

Example

```
int evID;
evID = I2C_getEventId();
```

I2C_setup — *Initializes I2C registers using initialization structure*

Function

void I2C_setup (I2C_Setup *Setup);

Arguments

Setup Pointer to an initialized initialization structure

Return Value

None

Description

Sets the address mode (7 or 10 bit), the own address, the prescaler value (based on system clock), the transfer rate, the number of bits/byte to be received or transmitted, the data loopback mode, and the free mode. Refer to the I2C_Setup structure for structure members.

Example

```
I2C_Setup Setup = {
0,          /* 7 bit address mode                        */
0x0000,     /* own address                               */
144,        /* clkout value (Mhz)                        */
400,        /* a number between 10 and 400              */
0,          /* 8 bits/byte to be received or transmitted */
0,          /* DLB mode off                              */
1           /* FREE mode on                              */
};

I2C_setup(&Setup);
```

I2C_IsrAddr

I2C structure used to assign functions for each interrupt structure

Structure

I2C_IsrAddr

Members

void (*alAddr)(void);	pointer to function for AL interrupt
void (*nackAddr)(void);	pointer to function for NACK interrupt
void (*ardyAddr)(void);	pointer to function for ARDY interrupt
void (*rrdyAddr)(void);	pointer to function for RRDY interrupt
void (*xrdyAddr)(void);	pointer to function for XRDY interrupt

Description

I2C structure used to assign functions for each of the five I2C interrupts. The structure member values should be pointers to the functions that are executed when a particular interrupt occurs.

Example

```
I2C_IsrAddr addr = {
   myALIsr,
   myNACKIsr,
   myARDYIsr,
   myRRDYIsr,
   myXRDYIsr
};
```

I2C_read

Performs master/slave receiver functions

Function

int I2C_read (Uint16 *data, int length, int master, Uint16 slaveaddress,
   int transfermode, int timeout, int checkbus);

Arguments

Uint16 *data	Pointer to data array
int length	length of data to be received
int master	master mode: 0 = slave, 1 = master
Uint16 slaveaddress	Slave address to receive from
int transfermode	Transfer mode of operation (SADP, SAD, etc.) Value Transfer Mode 1 S-A-D..(n)..D-P 2 S-A-D..(n)..D (repeat n times) 3 S-A-D-D-D..... (continuous)
int timeout	Timeout for bus busy, no acknowledge, transmit ready
int checkbus	flag used to check if bus is busy. Typically, it must be set to 1, except under special I2C program conditions.)

Return Value int

Value returned	Description
0	No errors
1	Bus busy; not able to generate start condition
2	Timeout for transmit ready (first byte)
4	Timeout for transmit ready (within main loop)

Description Performs master/slave receiver functions. Inputs are the data array to be transferred, length of data, master mode, slaveaddress, timeout for errors, and a check for bus busy flag.

Example

```
Uint16 datareceive[6]={0,0,0,0,0,0};

int x;

I2C_Init Init = {
0,          /* 7 bit address mode                          */
0x0000,     /* own address                                 */
144,        /* clkout value (Mhz)                          */
400,        /* a number between 10 and 400                 */
0,          /* 8 bits/byte to be received or transmitted */
0,          /* DLB mode off                                */
1           /* FREE mode on                                */
};


I2C_init(&Init);

z=I2C_read(datareceive,6,1,0x50,3,30000,0);
                      /* receives 6 bytes of data  */
                      /* in master receiver        */
                      /* S-A-D..(n)..D-P mode      */
                      /* to from the 0x50 address  */
                      /* with a timeout of 30000   */
                      /* and check for bus busy on */
```

I2C_readByte *Performs a 16-bit data read*

Function Uint16 I2C_readByte(
);

Arguments None

Return Value Data read for an I2C receive port.

Description Performs a direct 16-bit read from the data receive register I2CDRR.

Example

```
Uint16 Data;
...
Data = I2C_readByte();
```

This function does not check to see if valid data has been received. For this purpose, use I2C_rrdy().

I2C_reset *Resets a given serial port*

Function void I2C_reset(
);

Arguments None

Return Value None

Description Sets the IRS bit in the I2CMDR register to 1 (performs a reset).

Example

```
I2C_reset();
```

I2C_rfull *Reads the RSFULL bit of I2CSTR Register*

Function Uint16 I2C_rfull(
);

Arguments None

Return Value RFULL Returns RSFULL status bit of I2CSTR register to 0 (receive buffer empty), or 1(receive buffer full).

Description Reads the RSFULL bit of the I2CSTR register.

Example

```
if (I2C_rfull()) {
 …
}
```

I2C_rrdy *Reads the ICRRDY status bit of I2CSTR*

Function Uint16 I2C_rrdy(
);

Arguments None

Return Value RRDY Returns RRDY status bit of SPCR1, 0 or 1

Description Reads the RRDY status bit of the I2CSTR register. A 1 indicates the receiver is ready with data to be read.

Example
```
if (I2C_rrdy()) {
  ...
}
```

I2C_sendStop *Sets the STP bit in the I2CMDR register (generates stop condition)*

Function void I2C_sendStop();

Arguments None

Return Value None

Description Sets the STP bit in the I2CMDR register (generates a stop condition).

Example
```
I2C_sendStop();
```

I2C_setCallback *Associates functions to interrupts and installs dispatcher routines*

Function void I2C_setCallback(I2C_IsrAddr *isrAddr);

Arguments isrAddr is a structure containing pointers to the five functions that will be executed when the corresponding interrupt is enabled and received. These five functions should not be declared using the “interrupt” keyword.

Description I2C_setCallback associates each function to one of the I2C interrupts and installs the I2C dispatcher routine address in the I2C interrupt vector. It then determines what I2C interrupt as been received (by reading the I2CIMR register) and calls the corresponding function from the structure.

Example
```
I2C_IsrAddr addr = {
      myalIsr,
      mynackIsr,
      myardyIsr
      myrrdyIsr,
      myxrdyIsr
};

I2C_setCallback(&addr);
```

I2C_start

Starts the transmit and/or receive operation for an I2C port

Function

void I2C_start(
);

Arguments

None

Return Value

None

Description

Sets the STT bit in the I2CMDR register (generates a start condition). The values of the configuration structure are written to the port registers.

If desired, you can configure all I2C registers with:

```
I2C_config() [maintaining I2CMDR(STT)=0]
```

and later, use the I2C_start() function to start the I2C peripheral

Example

```
I2C_start();
```

I2C_write

Performs master/slave transmitter functions

Function

int I2C_write (Uint16 *data, int length, int master, Uint16 slaveaddress,
int transfermode, int timeout);

Arguments

Uint16 *data	Pointer to data array
int length	length of data to be transmitted
int master	master mode: 0 = slave, 1 = master
Uint16 slaveaddress	Slave address to transmit to
int transfermode	Transfer mode of operation (SADP, SAD, etc.)
int timeout	Timeout for bus busy, no acknowledge, or transmit ready

Value	Transfer Mode
1	S-A-D..(n)..D-P
2	S-A-D..(n)..D (repeat n times)
3	S-A-D-D-D..... (continuous)

Return Value

int

Value returned	Description
0	No errors
1	Bus busy; not able to generate start condition
2	Timeout for transmit ready (first byte)
3	NACK (No-acknowledge) received
4	Timeout for transmit ready (within main loop)
5	NACK (No-acknowledge) received (last byte)

Description Performs master/slave transmitter functions. Inputs are the data array to be transferred, length of data, master mode, slaveaddress, and timeout for errors.

int timeout Timeout for bus busy, no acknowledge, or transmit ready

Example

```
Uint16 databyte[7]={0,0,10,11,12,13,14};

int x;

I2C_Init Init = {
0,          /* 7 bit address mode                          */
0x0000,     /* own address                                 */
144,        /* clkout value (Mhz)                          */
400,        /* a number between 10 and 400                 */
0,          /* 8 bits/byte to be received or transmitted */
0,          /* DLB mode off                                */
1           /* FREE mode on                                */
};


I2C_init(&Init);
x=I2C_write (databyte,7,1,0x50,1,30000);
                                 /* sends 7 bytes of data  */
                                 /* in master transmitter  */
                                 /* S-A-D..(n)..D-P mode   */
                                 /* to the 0x50 slave      */
                                 /* address with a timeout */
                                 /* of 30000.              */
```

I2C_writeByte *Writes a 16-bit data value for I2CDXR*

Function

void I2C_writeByte(
    Uint16 Val
);

Arguments Val 16-bit data value to be written to I2C transmit register.

Return Value None

Description Directly writes a 16-bit value to the serial port data transmit register; I2CDXR; before writing the value, **this function does not check if the transmitter is ready.** For this purpose, use I2C_xrdy().

Example

```
I2C_writeByte(0x34);
```

I2C_xempty *Reads an XMST bit from an I2CST register*

Function

Uint16 I2C_xempty(
);

Arguments

None

Return Value

XSMT Returns the XSMT bit of I2CSTR register:
0 (transmit buffer empty),
or
1 (transmit buffer full).

Description

Reads the XSMT bit from the I2CSTR register. A 0 indicates the transmit shift (XSR) is empty.

Example

```
if (I2C_xempty()) {
  …
}
```

I2C_xrdy *Reads the ICXRDY status bit of the I2CSTR register*

Function

Bool I2C_xrdy(
);

Arguments

None

Return Value

XRDY Returns the XRDY status bit of the I2CSTR register.

Description

Reads the XRDY status bit of the I2CSTR 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 (I2C_xrdy()) {
  ...
  I2C_writeByte (0x34);
  ...
}
```

10.4 Macros

This section contains descriptions of the macros available in the I2C module.

The I2C API defines macros that have been designed for the following purposes:

The RMK macros create individual control-register masks for the following purposes:

- To initialize a I2C_Config structure that you then pass to functions such as I2C_Config().
- To use as arguments for the appropriate RSET macros.

Other macros are available primarily to facilitate reading and writing individual bits and fields in the I2C control registers.

Table 10-4. I2C Macros

(a) Macros to read/write I2C register values

Macro	**Syntax**
I2C_RGET()	Uint16 I2C_RGET(*REG*)
I2C_RSET()	Void I2C_RSET(*REG*, Uint16 *regval*)

(b) Macros to read/write I2C register field values (Applicable to registers with more than one field)

Macro	**Syntax**
I2C_FGET()	Uint16 I2C_FGET(*REG*, *FIELD*)
I2C_FSET()	Void I2C_FSET(REG,FIELD,Uint16 fieldval)

(c) Macros to create values to I2C registers and fields (Applicable to registers with more than one field)

Macro	**Syntax**
I2C_REG_RMK()	Uint16 I2C_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
I2C_FMK()	Uint16 I2C_FMK(*REG*, *FIELD*, *fieldval*)

(d) Macros to read a register address

Macro	**Syntax**
I2C_ADDR()	Uint16 I2C_ADDR(REG)

Notes:
1) *REG* indicates the registers: I2COAR, I2CIMR, I2CSTR, I2CCLKL, I2CCLKH, I2CDRR, I2CCNT, I2CSAR, I2CDXR, I2CMDR, I2CSRC, I2CPSC.
2) *FIELD* indicates the register field name.
   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*).

10.5 Examples

I2C programming examples using CSL are provided in:

- The *Programming the C5509 I2C Peripheral Application Report* (SPRA785)
- In the CCS examples directory: examples\<target>\csl\

Chapter 11

ICACHE Module

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

Topic **Page**

11.1 Overview

Table 11-2 lists the configuration structures and functions used with the ICACHE module.

Section 11.4 lists the macros available for the ICACHE module.

Currently, there are no handles available for the Instruction Cache.

Table 11-1. ICACHE Configuration Structure

Structure	Purpose	See page ...
ICACHE_Config	ICACHE configuration structure used to setup the Instruction Cache	11-3
ICACHE_Setup	ICACHE Configuration structure used to enable the Instruction Cache.	11-4
ICACHE_TagSet	ICACHE structure used to set the tag registers.	11-4

Table 11-2. ICACHE Functions

Structure	Purpose	See page ...
ICACHE_config	Sets up the ICACHE register using the configuration structure	11-5
ICACHE_disable	Resets the Cache Enable bit in status register 3	11-5
ICACHE_enable	Sets the Cache Enable bit in status register 3	11-6
ICACHE_flush	Sets the Cache Flush bit in status register 3	11-6
ICACHE_freeze	Sets the Cache Freeze bit in status register 3	11-6
ICACHE_setup	Configures the ICACHE and enables it	11-7
ICACHE_tagset	Sets the values of the Ramset Tags	11-7
ICACHE_unfreeze	Resets the Cache Freeze bit in status register 3	11-7

11.2 Configuration Structures

The following are configuration structures used to set up the ICACHE module.

ICACHE_Config

ICACHE configuration structure used to setup the ICACHE

Structure

ICACHE_Config

Members

Members

Uint16 icgc	Global Control Register
Uint16 icwc	N-way Control Register (not supported on C5502)
Uint16 icrc1	Ramset 1 Control Register (not supported on C5502)
Uint16 icrtag1	Ramset 1 Tag Register (not supported on C5502)
Uint16 icrc2	Ramset 2 Control Register (not supported on C5502)
Uint16 icrtag2	Ramset 2 Tag Register (not supported on C5502)

Description

The ICACHE configuration structure is used to set up the cache. You create and initialize this structure, then pass its address to the ICACHE_config() function. You can use literal values or the ICACHE_RMK macros to create the structure member values.

Example

```
ICACHE_Config MyConfig = {
  0x0060, /* Global Control   */
  0x1000, /* N-way Control    */
  0x0000, /* Ramset 1 Control */
  0x1000, /* Ramset 1 Tag     */
  0x0000, /* Ramset 1 Control */
  0x1000 /* Ramset 1 Tag      */
  };
...
ICACHE_config(&MyConfig);
```

Example

For C5502

```
ICACHE_Config MyConfig = {
  0x0000, /* Global Control   */
  };
```

ICACHE_Setup
Structure used to configure and enable the ICACHE

Structure ICACHE_Setup

Members

Members	Uint16	rmode
Uint32	r1addr	
Uint32	r2addr	
rmode	Ramset Mode. Can take the following predefined values: ICACHE_ICGC_RMODE_0RAMSET ICACHE_ICGC_RMODE_1RAMSET ICACHE_ICGC_RMODE_2RAMSET	

Description ICACHE setup structure is used to configure and enable the ICACHE. The structure is created and initialized. Its address is passed to the ICACHE_setup() function.

Example

```
ICACHE_Setup Mysetup = {
ICACHE_ICGC_RMODE_1RAMSET,
0x50000,
0x0000};
...
ICACHE_setup(&Mysetup);
```

ICACHE_Tagset
Structure used to configure the ramset tag registers

Structure ICACHE_Tagset

Members

Members	Uint32	r1addr
Uint32	r2addr	

Description ICACHE tag set structure is used to configure the ramset tag registers of the ICACHE.

Example

```
ICACHE_Tagset Mytagset = {
0x50000,
0x0000};
...
ICACHE_tagset(&Mytagset);
```

11.3 Functions

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

ICACHE_config

Sets up ICACHE registers using configuration structure

Function

void ICACHE_config(
    ICACHE_Config *Config
);

Arguments

Config   Pointer to an initialized configuration structure

Return Value

None

Description

Sets up the ICACHE register using the configuration structure. The values of the structure are written to the registers ICGC, ICWC, ICRC1, ICRTAG1, ICRC2 and ICRTAG2 (see also ICACHE_Config).

Example

```
ICACHE_Config MyConfig = {
};
…
ICACHE_config(&MyConfig);
```

ICACHE_disable

Resets the ICACHE enable bit in the Status Register 3

Function

void ICACHE_disable();

Arguments

None

Return Value

None

Description

Function resets the ICACHE enable bit in the Status Register 3 and disables the ICACHE. After disabling the ICACHE the values in the ICACHE are preserved.

Example

```
ICACHE_disable();
```

ICACHE_enable

Sets the ICACHE enable bit in the Status Register 3

Function void ICACHE_enable();

Arguments None

Return Value None

Description Function sets the ICACHE enable bit in the Status Register 3 and then polls the enable flag in the Cache Status Register. This function is useful when the ICACHE was disabled using the ICACHE_disable() function. In order to initialize the ICACHE the use of the ICACHE_setParams is prefered since this function will also enable the ICACHE.

Example

```
ICACHE_enable();
```

ICACHE_flush

Sets the ICACHE flush bit in the Status Register 3

Function void ICACHE_flush();

Arguments None

Return Value None

Description Function sets the ICACHE flush bit in the Status Register 3 The content of the ICACHE is invalidated.

Example

```
ICACHE_flush();
```

ICACHE_freeze

Sets the ICACHE freeze bit in the Status Register 3

Function void ICACHE_freeze();

Arguments None

Return Value None

Description Function sets the ICACHE freeze bit in the Status Register 3 and freezes the content of the ICACHE.

Example

```
ICACHE_freeze();
```

ICACHE_setup *Configures the ICACHE and enables it*

Function void ICACHE_setup(ICACHE_Setup *setup);

Arguments setup Pointer to an initialized setup structure

Return Value None

Description Sets the Ramset Mode and enables the ICACHE

Example

```
ICACHE_Setup mySetup = {
};
...
ICACHE_setup (&mySetup);
```

ICACHE_tagset *Sets the address in the Ramset Tag registers*

Function void ICACHE_tagset(ICACHE_Tagset *params);

Arguments params Pointer to an initialized tagset structure

Return Value None

Description Function sets the addresses in the Ramset Tag registers. This function is useful when the user wants to change the Ramset addresses after the ICACHE had been flushed .

Example

```
ICACHE_Tagset mySetup = {
};
...
ICACHE_tagset(&mySetup);
```

ICACHE_unfreeze *Resets the ICACHE freeze bit in the Status Register 3*

Function void ICACHE_unfreeze();

Arguments None

Return Value None

Description Function resets the ICACHE freeze bit in the Status Register 3 the content of the ICACHE is unfrozen.

Example

```
ICACHE_unfreeze();
```

11.4 Macros

The CSL offers a collection of macros to access CPU control registers and fields.

Table 11-3 lists the ICACHE macros available. To use them include “csl_icache.h.”

Table 11-3. ICACHE CSL Macros

(a) Macros to read/write ICACHE register values

Macro	Syntax
ICACHE_RGET()	Uint16 ICACHE_RGET(*REG*)
ICACHE_RSET()	void ICACHE_RSET(*REG*, Uint16 *regval*)

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

Macro	Syntax
ICACHE_FGET()	Uint16 ICACHE_FGET(*REG*, *FIELD*)
ICACHE_FSET()	void ICACHE_FSET(*REG*, *FIELD*, Uint16 *fieldval*)

(c) Macros to create value to write to ICACHE registers and fields (Applicable only to registers with more than one field)

Macro	Syntax
ICACHE_*REG*_RMK()	Uint16 ICACHE_*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
ICACHE_FMK()	Uint16 ICACHE_FMK(*REG*, *FIELD*, *fieldva*l)

(d) Macros to read a register address

Macro	Syntax
ICACHE_ADDR()	Uint16 ICACHE_ADDR(*REG*)

Notes:
1) *REG* indicates the registers:ICGC, ICWC, ICST, ICRC1&2 or ICRTAG1&2.
2) *FIELD* indicates the register field name.
   - 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*)

Chapter 12

IRQ Module

This chapter describes the IRQ module, lists the API structure and functions within the module, and provides an IRQ API reference section. The IRQ module provides an easy to use interface for enabling/disabling and managing interrupts.

Topic **Page**

12.1 Overview

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

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

  The DSP BIOS dispatcher is responsible for dynamically handling interrupts and maintains a table of ISRs to be executed for specific interrupts. The IRQ module has a set of APIs that update the dispatch table. Table 12-2 lists the IRQ APIs.

The IRQ functions can be used with or without DSP/BIOS; however, if DSP/BIOS is present, do not disable interrupts for long periods of time because this could disrupt the DSP/BIOS environment.

IRQ_plug() is 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.

The API functions that enable DSP/BIOS dispatcher communication are noted in the table. These functions should be used only when DSP/BIOS is present **and** the DSP/BIOS dispatcher is enabled.

Table 12-3 lists all IRQ logical interrupt events for this module.

Table 12-1. IRQ Configuration Structure

Syntax	Description	See page ...
IRQ_Config	IRQ structure that contains all local registers required to set up a specific IRQ channel.	12-8

Table 12-2. IRQ Functions

Syntax	Description	See page ...
IRQ_clear()	Clears the interrupt flag in the IFR0/1 registers for the specified event.	12-9
IRQ_config()†	Updates the DSP/BIOS dispatch table with a new configuration for the specified event.	12-9
IRQ_disable()	Disables the specified event in the IMR0/1 registers.	12-10
IRQ_enable()	Enables the specified event in the IMR0/1 register flags.	12-10
IRQ_getArg()†	Returns value of the argument to the interrupt service routine that the DSP/BIOS dispatcher passes when the interrupt occurs.	12-10
IRQ_getConfig()†	Returns current DSP/BIOS dispatch table entries for the specified event.	12-11
IRQ_globalDisable()	Globally disables all maskable interrupts. (INTM = 1)	12-11
IRQ_globalEnable()	Globally enables all maskable interrupts. (INTM = 0)	12-12
IRQ_globalRestore()	Restores the status of global interrupt enable/disable (INTM).	12-12
IRQ_map()†	Maps a logical event to its physical interrupt.	12-13
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 if the DSP/BIOS HWI module or the DSP/BIOS dispatcher are in use.	12-13
IRQ_restore()	Restores the status of the specified event in the IMR0/1 register.	12-14
IRQ_setArg()†	Sets the value of the argument for DSP/BIOS dispatch to pass to the interrupt service routine for the specified event.	12-14
IRQ_setVecs()	Sets the base address of the interrupt vector table.	12-15
IRQ_test()	Polls the interrupt flag in IFR register the specified event.	12-15

12.1.1 The Event ID Concept

The IRQ module assigns an event ID to each of the possible physical interrupts. Because there are more events possible than events that 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 (see Table 12-3). All of the IRQ APIs operate on logical events.

Table 12-3. IRQ_EVT_NNNN Events List

Constant	Purpose
IRQ_EVT_RS	Reset
IRQ_EVT_SINTR	Software Interrupt
IRQ_EVT_NMI	Non-Maskable Interrupt (NMI)
IRQ_EVT_SINT16	Software Interrupt #16
IRQ_EVT_SINT17	Software Interrupt #17
IRQ_EVT_SINT18	Software Interrupt #18
IRQ_EVT_SINT19	Software Interrupt #19
IRQ_EVT_SINT20	Software Interrupt #20
IRQ_EVT_SINT21	Software Interrupt #21
IRQ_EVT_SINT22	Software Interrupt #22
IRQ_EVT_SINT23	Software Interrupt #23
IRQ_EVT_SINT24	Software Interrupt #24
IRQ_EVT_SINT25	Software Interrupt #25
IRQ_EVT_SINT26	Software Interrupt #26
IRQ_EVT_SINT27	Software Interrupt #27
IRQ_EVT_SINT28	Software Interrupt #28
IRQ_EVT_SINT29	Software Interrupt #29
IRQ_EVT_SINT30	Software Interrupt #30
IRQ_EVT_SINT0	Software Interrupt #0
IRQ_EVT_SINT1	Software Interrupt #1
IRQ_EVT_SINT2	Software Interrupt #2
IRQ_EVT_SINT3	Software Interrupt #3
IRQ_EVT_SINT4	Software Interrupt #4
IRQ_EVT_SINT5	Software Interrupt #5

Table 12-3. IRQ_EVT_NNNN Events List (Continued)

Constant	Purpose1
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
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)

Table 12-3. IRQ_EVT_NNNN Events List (Continued)

Constant	**Purpose1**
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_RTC	RTC Interrupt
IRQ_EVT_I2C	I2C Interrupt
IRQ_EVT_WDTINT	Watchdog Timer Interrupt

12.2 Using Interrupts with CSL

Interrupts can be managed using any of the following methods:

- You can use DSP/BIOS HWIs: Refer to DSP/BIOS Users Guide.
- You can use the DSP/BIOS Dispatcher
- You can use CSL IRQ routines: Example 12-1 illustrates how to initialize and manage interrupts outside the DSP/BIOS environment.

Example 12-1. Manual Interrupt Setting Outside DSP/BIOS HWIs

```
extern Uint32 myVec;

; ...
interrupt void myIsr();

; ...
main (){

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

IRQ_setVecs((Uint32)(&myvec) << 1));
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((Uint32)(&myvec) << 1));
eventId = TIMER_getEventId (hTimer);
IRQ_plug (eventId,&myIsr);
IRQ_enable (eventId);
IRQ_globalEnable();
; ...

}

interrupt void myIsr(void)

{

//...

}
```

12.3 Configuration Structures

The following is the configuration structure used to set up the IRQ module.

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 cachectrl	Currently, this member has no function and has been reserved for future expansion.
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, /* cachectrl */
  0x0000, /* funcArg   */
};
```

12.4 Functions

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

IRQ_clear — *Clears event flag from IFR register*

Function

void IRQ_clear(
    Uint16 EventId
);

Arguments

EventId	Event ID, see IRQ_EVT_NNNN (Table 12-3) for a complete list of events. Or, use the PER_getEventId() function to get the Event ID.

Return Value None

Description Clears the event flag from the IFR register

Example

```
IRQ_clear(IRQ_EVT_TINT0);
```

IRQ_config — *Updates an entry in the DSP/BIOS Dispatch Table*

Function

void IRQ_config(
    Uint16 EventId,
    IRQ_Config *Config
);

Arguments

EventID	Event ID, see IRQ_EVT_NNNN for a complete list of events.
Config	Pointer to an initialized configuration structure

Return Value None

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

Example

```
IRQ_config myConfig =  {
              0X0000,
              0X0300,
              0X0000,
              0X0000
           };
IRQ_config (IRQ_EVT_TINT0, &myConfig);
```

IRQ_disable — *Disables specified event*

Function

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

Arguments

EventId — Event ID, see IRQ_EVT_NNNN (Table 12-3) for a complete list of events. Or, use the PER_getEventId() function to get the EventID.

Return Value

int — Old value of the event

Description

Disables the specified event, by modifying the IMR register.

Example

```
Uint32 oldint;
oldint = 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 12-3) for a complete list of events. Or, use the PER_getEventId() function to get the Event ID.

Return Value

None

Description

Enables the specified event.

Example

```
Uint32 oldint;
oldint = 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 12-3) for a complete list of events. Or, use the PER_getEventId() function to get the EventID.

Return Value Value of argument

Description Returns value for specified event.

Example

```
Uint32 evVal;
evVal = 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 12-3) for a complete list of events. Or, use the PER_getEventId() function to get the EventID.
Config	Pointer to configuration structure

Return Value None

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

Example

```
IRQ_Config myConfig;
IRQ_getConfig(IRQ_EVT_SINT3, &myConfig);
```

IRQ_globalDisable *Globally disables interrupts*

Function

int IRQ_globalDisable(
);

Arguments None

Return Value intm Returns the old INTM value

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

Example

```
int 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

```
int 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 for a complete list of events.

Return Value

None

Description

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

Example

```
IRQ_map(IRQ_EVT_TINT0);
```

IRQ_plug — *Initializes an interrupt vector table vector*

Function

void IRQ_plug(
    Uint16 EventId,
    IRQ_IsrPtr funcAddr
);

Arguments

EventId   Event ID, see IRQ_EVT_NNNN (Table 12-3) for a complete list of events. Or, use the PER_getEventId() function to get the EventID.

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

Return Value

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

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

IRQ_restore — *Restores the state of a specified event*

Function

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

Arguments

EventId	Event ID, see IRQ_EVT_NNNN (Table 12-3) for a complete list of events. Or, use the PER_getEventId() function to get the EventID.
Old_flag	Value used to restore an event (0 = enable, 1 = disable)

Return Value None

Description This function restores the event's state to the value that was originally passed to it.

Example

```
int oldint;
oldint = IRQ_disable(IRQ_EVT_TINT0);
.
.
.
IRQ_restore(IRQ_EVT_TINT0, oldint);
```

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 12-3) for a complete list of events. Or, use the PER_getEventId() function to get the EventID.

Return Value None

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

Example

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

IRQ_setVecs *Sets the base address of the interrupt vectors*

Function

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

Arguments

IVPD	IVPD pointer to the DSP interrupt vector table

Return Value Old IVPD register value

Description Use this function to set the base address of the interrupt vector table in the IVPD and IVPH registers (both registers are set to the same value).

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

Example

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

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

Function

```
Bool IRQ_test(
    Uint16 EventId
);
```

Arguments

EventId	Event ID, see IRQ_EVT_NNNN (Table 12-3) for a complete list of events. Or, use the PER_getEventId() function to get the EventID.

Return Value Event flag, 0 or 1

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

Example

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

Chapter 13

McBSP Module

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

Topic **Page**

13.1 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 13-2 lists the structure and functions for use with the McBSP modules.

Table 13-1 lists the configuration structure used to set up the McBSP.

Table 13-2 lists the functions available for use with the McBSP module

Table 13-3 lists McBSP registers and fields.

Table 13-1. McBSP Configuration Structure

Syntax	Description	See page ...
MCBSP_Config	McBSP configuration structure used to setup a McBSP port.	13-6

Table 13-2. McBSP Functions

Syntax	Description	See page ...
MCBSP_channelDisable()	Disables one or several McBSP channels	13-8
MCBSP_channelEnable()	Enables one or several McBSP channels of the selected register	13-9
MCBSP_channelStatus()	Returns the channel status	13-11
MCBSP_close()	Closes the McBSP and its corresponding handle	13-12
MCBSP_config()	Sets up McBSP using configuration structure (MCBSP_Config)	13-12
MCBSP_getConfig()	Get MCBSP channel configuration	13-14
MCBSP_getRcvEventId()	Retrieves the receive event ID for the given port	13-15
MCBSP_getXmtEventId()	Retrieves the transmit event ID for the given port	13-15
MCBSP_getPort()	Get MCBSP Port number used in given handle	13-14
MCBSP_open()	Opens the McBSP and assigns a handle to it	13-16
MCBSP_read16()	Performs a direct 16-bit read from the data receive register DRR1	13-17
MCBSP_read32()	Performs two direct 16-bit reads: data receive register 2 DRR2 (MSB) and data receive register 1 DRR1 (LSB)	13-17
MCBSP_reset()	Resets the McBSP registers with default values	13-18

Syntax	Description	See page ...
MCBSP_rfull()	Reads the RFULL bit SPCR1 register	13-18
MCBSP_rrdy()	Reads the RRDY status bit of the SPCR1 register	13-19
MCBSP_start()	Starts a McBSP receive/transmit based on start flags	13-19
MCBSP_write16()	Writes a 16-bit value to the serial port data transmit register, DXR1	13-21
MCBSP_write32()	Writes two 16-bit values to the two serial port data transmit registers, DXR2 (16-bit MSB) and DXR1 (16-bit LSB)	13-21
MCBSP_xempty()	Reads the XEMPTY bit from the SPCR2 register	13-22
MCBSP_xrdy()	Reads the XRDY status bit of the SPCR2 register	13-22

13.1.1 MCBSP Registers

Table 13-3. MCBSP Registers

Register	Field
SPCR1	DLB, RJUST, CLKSTP, DXENA, ABIS, RINTM, RSYNCERR, (R)RFULL, (R)RRDY, RRST
SPCR2	FREE, SOFT, FRST, GRST, XINTM, XSYNCERR, (R)XEMPTY, (R)XRDY, XRST
PCR	SCLKME, (R)CLKSSTAT, DXSTAT, (R)DRSTAT, FSXP, FSRP, CLKXP, CLKRP, IDLEEN, XIOEN, RIOEN, FSXM, FSRM, CLKXM, CLKRM
RCR1	RFRLEN1, RWDLEN1
RCR2	RPHASE, RFRLEN2, RWDLEN2, RCOMPAND, RFIG, RDATDLY
XCR1	XFRLEN1, XWDLEN1
XCR2	XPHASE, XFRLEN2, XWDLEN2, XCOMPAND, XFIG, XDATDLY
SRGR1	FWID, CLKGDV
SRGR2	GSYNC, CLKSP, CLKSM, FSGM, FPER
MCR1	RMCME, RPBBLK, RPABLK, (R)RCBLK, RMCM
MCR2	XMCME, XPBBLK, XPABLK, (R)XCBLK, XMCM
XCERA	XCEY15, XCEY14, XCEY13, XCEY12, XCEY11, XCEY10, XCEY9, XCEY8, XCEY7, XCEY6, XCEY5, XCEY4, XCEY3, XCEY2, XCEY1, XCEY0

Table 13-3. MCBSP Registers(Continued)

Register	Field
XCERB	XCEY15, XCEY14, XCEY13, XCEY12, XCEY11, XCEY10, XCEY9, XCEY8, XCEY7, XCEY6, XCEY5, XCEY4, XCEY3, XCEY2, XCEY1, XCEY0
XCERC	XCEY15, XCEY14, XCEY13, XCEY12, XCEY11, XCEY10, XCEY9, XCEY8, XCEY7, XCEY6, XCEY5, XCEY4, XCEY3, XCEY2, XCEY1, XCEY0
XCERD	XCEY15, XCEY14, XCEY13, XCEY12, XCEY11, XCEY10, XCEY9, XCEY8, XCEY7, XCEY6, XCEY5, XCEY4, XCEY3, XCEY2, XCEY1, XCEY0
XCERE	XCEY15, XCEY14, XCEY13, XCEY12, XCEY11, XCEY10, XCEY9, XCEY8, XCEY7, XCEY6, XCEY5, XCEY4, XCEY3, XCEY2, XCEY1, XCEY0
XCERF	XCEY15, XCEY14, XCEY13, XCEY12, XCEY11, XCEY10, XCEY9, XCEY8, XCEY7, XCEY6, XCEY5, XCEY4, XCEY3, XCEY2, XCEY1, XCEY0
XCERG	XCEY15, XCEY14, XCEY13, XCEY12, XCEY11, XCEY10, XCEY9, XCEY8, XCEY7, XCEY6, XCEY5, XCEY4, XCEY3, XCEY2, XCEY1, XCEY0
XCERH	XCEY15, XCEY14, XCEY13, XCEY12, XCEY11, XCEY10, XCEY9, XCEY8, XCEY7, XCEY6, XCEY5, XCEY4, XCEY3, XCEY2, XCEY1, XCEY0
RCERA	RCEY15, RCEY14, RCEY13, RCEY12, RCEY11, RCEY10, RCEY9, RCEY8, RCEY7, RCEY6, RCEY5, RCEY4, RCEY3, RCEY2, RCEY1, RCEY0
RCERB	RCEY15, RCEY14, RCEY13, RCEY12, RCEY11, RCEY10, RCEY9, RCEY8, RCEY7, RCEY6, RCEY5, RCEY4, RCEY3, RCEY2, RCEY1, RCEY0
RCERC	RCEY15, RCEY14, RCEY13, RCEY12, RCEY11, RCEY10, RCEY9, RCEY8, RCEY7, RCEY6, RCEY5, RCEY4, RCEY3, RCEY2, RCEY1, RCEY0
RCERD	RCEY15, RCEY14, RCEY13, RCEY12, RCEY11, RCEY10, RCEY9, RCEY8, RCEY7, RCEY6, RCEY5, RCEY4, RCEY3, RCEY2, RCEY1, RCEY0
RCERE	RCEY15, RCEY14, RCEY13, RCEY12, RCEY11, RCEY10, RCEY9, RCEY8, RCEY7, RCEY6, RCEY5, RCEY4, RCEY3, RCEY2, RCEY1, RCEY0
RCERF	RCEY15, RCEY14, RCEY13, RCEY12, RCEY11, RCEY10, RCEY9, RCEY8, RCEY7, RCEY6, RCEY5, RCEY4, RCEY3, RCEY2, RCEY1, RCEY0
RCERG	RCEY15, RCEY14, RCEY13, RCEY12, RCEY11, RCEY10, RCEY9, RCEY8, RCEY7, RCEY6, RCEY5, RCEY4, RCEY3, RCEY2, RCEY1, RCEY0
RCERH	RCEY15, RCEY14, RCEY13, RCEY12, RCEY11, RCEY10, RCEY9, RCEY8, RCEY7, RCEY6, RCEY5, RCEY4, RCEY3, RCEY2, RCEY1, RCEY0
DRR1	RCEY15, RCEY14, RCEY13, RCEY12, RCEY11, RCEY10, RCEY9, RCEY8, RCEY7, RCEY6, RCEY5, RCEY4, RCEY3, RCEY2, RCEY1, RCEY0
DRR2	RCEY15, RCEY14, RCEY13, RCEY12, RCEY11, RCEY10, RCEY9, RCEY8, RCEY7, RCEY6, RCEY5, RCEY4, RCEY3, RCEY2, RCEY1, RCEY0

Table 13-3. MCBSP Registers(Continued)

Register	Field
DXR1	RCEY15, RCEY14, RCEY13, RCEY12, RCEY11, RCEY10, RCEY9, RCEY8, RCEY7, RCEY6, RCEY5, RCEY4, RCEY3, RCEY2, RCEY1, RCEY0
DXR2	RCEY15, RCEY14, RCEY13, RCEY12, RCEY11, RCEY10, RCEY9, RCEY8, RCEY7, RCEY6, RCEY5, RCEY4, RCEY3, RCEY2, RCEY1, RCEY0

Note: R = Read Only; W = Write; By default, most fields are Read/Write

13.2 Configuration Structures

The following is the configuration structure used to set up the McBSP.

MCBSP_Config — *McBSP configuration structure used to set up a 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
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 The McBSP configuration structure is used to set up a McBSP port. You create and initialize this structure and then pass its address to the MCBSP_config() function. You can use literal values or the *MCBSP_RMK* macros to create the structure member values.

13.3

Example

```
MCBSP_Config config1 = {
   0xFFFF, /* spcr1  */
   0x03FF, /* spcr2  */
   0x7FE0, /* rcr1   */
   0xFFFF, /* rcr2   */
   0x7FE0, /* xcr1   */
   0xFFFF, /* xcr2   */
   0xFFFF, /* srgr1  */
   0xFFFF, /* srgr2  */
   0x03FF, /* mcr1   */
   0x03FF, /* mcr2   */
   0xFFFF, /* pcr    */
   0xFFFF, /* rcera  */
   0xFFFF, /* rcerb  */
   0xFFFF, /* rcerc  */
   0xFFFF, /* rcerd  */
   0xFFFF, /* rcere  */
   0xFFFF, /* rcerf  */
   0xFFFF, /* rcerg  */
   0xFFFF, /* rcerh  */
   0xFFFF, /* xcera  */
   0xFFFF, /* xcerb  */
   0xFFFF, /* xcerc  */
   0xFFFF, /* xcerd  */
   0xFFFF, /* xcere  */
   0xFFFF, /* xcerf  */
   0xFFFF, /* xcerg  */
   0xFFFF  /* xcerh  */
}
...
hMcbsp = MCBSP_open(MCBSP_PORT0, MCBSP_OPEN_RESET)
...
MCBSP_config(hMcbsp, &config1);
```

13.4 Functions

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

MCBSP_channelDisable *Disables one or several McBSP channels*

Function

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

Arguments

hMcbsp Handle to McBSP port obtained by MCBSP_open()

RegName Receive and Transmit Channel Enable Registers:

RCERA
RCERB
XCERA
XCERB
RCERC
RCERD
RCERE
RCERF
RCERG
RCERH
XCERC
XCERD
XCERE
XCERF
XCERG
XCERH

Channels Available values for the specific RegName are:

MCBSP_CHAN0
MCBSP_CHAN1
MCBSP_CHAN2
MCBSP_CHAN3
MCBSP_CHAN4
MCBSP_CHAN5
MCBSP_CHAN6
MCBSP_CHAN7
MCBSP_CHAN8
MCBSP_CHAN9
MCBSP_CHAN10
MCBSP_CHAN11
MCBSP_CHAN12

MCBSP_CHAN13
MCBSP_CHAN14
MCBSP_CHAN15

Return Value None

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

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

Example

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

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

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

Function

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

Arguments

hMcbsp Handle to McBSP port obtained by MCBSP_open()

RegName Receive and Transmit Channel Enable Registers:
RCERA
RCERB
XCERA
XCERB
RCERC
RCERD
RCERE
RCERF
RCERG
RCERH
XCERC
XCERD
XCERE

XCERF
XCERG
XCERH

Channels   Available values for the specificReg Addr are:

MCBSP_CHAN0
MCBSP_CHAN1
MCBSP_CHAN2
MCBSP_CHAN3
MCBSP_CHAN4
MCBSP_CHAN5
MCBSP_CHAN6
MCBSP_CHAN7
MCBSP_CHAN8
MCBSP_CHAN9
MCBSP_CHAN10
MCBSP_CHAN11
MCBSP_CHAN12
MCBSP_CHAN13
MCBSP_CHAN14
MCBSP_CHAN15

Return Value

None

Description

Enables one or several McBSP channels of the selected register.

To enable several channels at the same time, the sign “|” OR has to be added in between.

Example

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

MCBSP_channelStatus *Returns channel status*

Function

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

Arguments

hMcbsp Handle to McBSP port obtained by MCBSP_open()

RegName Receive and Transmit Channel Enable Registers:

RCERA
RCERB
XCERA
XCERB
RCERC
RCERD
RCERE
RCERF
RCERG
RCERH
XCERC
XCERD
XCERE
XCERF
XCERG
XCERH

Channel Selectable Channels for the specific RegName are:

MCBSP_CHAN0
MCBSP_CHAN1
MCBSP_CHAN2
MCBSP_CHAN3
MCBSP_CHAN4
MCBSP_CHAN5
MCBSP_CHAN6
MCBSP_CHAN7
MCBSP_CHAN8
MCBSP_CHAN9
MCBSP_CHAN10
MCBSP_CHAN11
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 (RegName). Only one channel can be observed.

Example

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

MCBSP_close *Closes a McBSP Port*

Function

void MCBSP_close(
MCBSP_Handle hMcbsp
);

Arguments hMcbsp Device Handle (see MCBSP_open()).

Return Value None

Description Closes a previously opened McBSP port. The McBSP registers are set to their default values and any associated interrupts are disabled and cleard.

Example

```
MCBSP_close(hMcbsp);
```

MCBSP_config *Sets up a McBSP port using a 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 directly to the Mcbsp port registers.

Note:

If you want to configure all McBSP registers without starting the McBSP port, use MCBSP_config() without setting the SPCR2 (XRST, RRST, GRST, and FRST) fields. Then, after you write the first data valid to the DXR registers, call MCBSP_start() when ready to start the McBSP port. This guarantees that the correct value is transmitted/received.

Example

```
MCBSP_Config MyConfig = {
   0xFFFF, /* spcr1  */
   0x03FF, /* spcr2  */
   0x7FE0, /* rcr1   */
   0xFFFF, /* rcr2   */
   0x7FE0, /* xcr1   */
   0xFFFF, /* xcr2   */
   0xFFFF, /* srgr1  */
   0xFFFF, /* srgr2  */
   0x03FF, /* mcr1   */
   0x03FF, /* mcr2   */
   0xFFFF, /* pcr    */
   0xFFFF, /* rcera  */
   0xFFFF, /* rcerb  */
   0xFFFF, /* rcerc  */
   0xFFFF, /* rcerd  */
   0xFFFF, /* rcere  */
   0xFFFF, /* rcerf  */
   0xFFFF, /* rcerg  */
   0xFFFF, /* rcerh  */
   0xFFFF, /* xcera  */
   0xFFFF, /* xcerb  */
   0xFFFF, /* xcerc  */
   0xFFFF, /* xcerd  */
   0xFFFF, /* xcere  */
   0xFFFF, /* xcerf  */
   0xFFFF, /* xcerg  */
   0xFFFF  /* xcerh  */
};
...
   MCBSP_config(myhMcbsp, &MyConfig);
```

MCBSP_getConfig *Reads the MCBSP configuration in the configuration structure*

Function

```
void MCBSP_getConfig(
    MCBSP_Handle hMcbsp,
    MCBSP_Config *Config
);
```

Arguments

hMcbsp	McBSP Device Handle obtained by MCBSP_open()
Config	Pointer to a McBSP configuration structure

Return Value None

Description Reads the McBSP configuration into the configuration structure. See also McBSP_Config.

Example

```
MCBSP_Config myConfig;
...
hMcbsp = MCBSP_open(MCBSP_PORT0, 0);
MCBSP_getConfig(hMcbsp, &myConfig);
```

MCBSP_getPort *Get McBSP port number used in given handle*

Function Uint16 MCBSP_getPort (MCBSP_Handle hMcbsp)

Arguments

hMcbsp	Handle to McBSP port given by MCBSP_open()

Return Value Port number

Description Get Port number used by specific handle

Example

```
Uint16  PortNum;
  ...
    PortNum = MCBSP_getPort (hMcbsp));
```

MCBSP_getRcvEventId *Retrieves the receive event ID for a given McBSP 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 a given port. Use this ID to manage the event using the IRQ module.

Example

```
Uint16 RecvEventId;
...
RecvEventId = MCBSP_getRcvEventId(hMcbsp);
IRQ_enable(RecvEventId);
```

MCBSP_getXmt EventID *Retrieves the transmit event ID for a given MCBSP port*

Function

Uint16 MCBSP_getXmtEventId(
    MCBSP_Handle hMcbsp
);

Arguments

hMcbsp   Handle to McBSP port obtained by MCBSP_open()

Return Value

Transmitter event ID

Description

Retrieves the IRQ transmit event ID for the given port. Use this ID to manage the event using the IRQ module.

Example

```
Uint16 XmtEventId;
...
XmtEventId = MCBSP_getXmtEventId(hMcbsp);
IRQ_enable(XmtEventId);
```

MCBSP_open — *Opens a McBSP port*

Function

```
MCBSP_Handle MCBSP_open(
    int devnum,
    Uint32 flags
);
```

Arguments

devNum — McBSP device (port) number:
- MCBSP_PORT0
- MCBSP_PORT1
- MCBSP_PORT2
- MCBSP_PORT_ANY

flags — Open flags, may be logical OR of any of the following:
- MCBSP_OPEN_RESET

Return Value

MCBSP_Handle — Device handle

Description

Before a McBSP device 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 is used in subsequent McBSP API calls. If the function fails, INV (-1) is returned.

If the MCBSP_OPEN_RESET is specified, then the power on defaults are set and any interupts are disabled and cleared.

Example

```
MCBSP_Handle hMcbsp;
...
hMcbsp = MCBSP_open(MCBSP_PORT0,MCBSP_OPEN_RESET);
```

MCBSP_read16 *Reads a 16-bit value*

Function

```
Uint16 MCBSP_read16(
    MCBSP_Handle hMcbsp
);
```

Arguments hMcbsp McBSP Device Handle obtained by MCBSP_open()

Return Value 16-bit value

Description Directly reads a 16-bit value from the McBSP data receive register DRR1.

Depending on the receive word data length you have selected in the RCR1/RCR2 registers, the actual data could be 8, 12, or 16 bits long.

This function does not verify that new valid data has been received. Use MCBSP_rrdy() (prior to calling MCBSP_read16()) for this purpose.

Example

```
Uint16 val16;
val16 = MCBSP_read16(hMcbsp);
```

MCBSP_read32 *Reads a 32-bit value*

Function

```
Uint32 MCBSP_read32(
    MCBSP_Handle hMcbsp
);
```

Arguments hMcbsp McBSP Device Handle (see MCBSP_open())

Return Value 32-bit value (MSW-LSW ordering)

Description A 32-bit read. First, the 16-bit MSW (Most significant word) is read from register DRR2. Then, the 16-bit LSW (least significant word) is read from register DRR1.

Depending on the receive word data length you have selected in the RCR1/RCR2 register, the actual data could be 20, 24, or 32 bits.

This function does not check to verify that new valid data has been received. Use MCBSP_rrdy() (prior to calling MCBSP_read32()) for this purpose.

Example

```
Uint32 val32;
val32 = MCBSP_read32(hMcbsp);
```

MCBSP_reset *Resets a McBSP port*

Function

void MCBSP_reset(
    MCBSP_Handle hMcbsp
);

Arguments

hMcbsp   Device handle, see MCBSP_open();

Return Value

None

Description

Resets the McBSP device. Disables and clears the interrupt event and sets the McBSP registers to default values. If INV is specified, all McBSP devices are reset.

Actions Taken:

All serial port registers are set to their power-on defaults.

All associated interrupts are disabled and cleared.

Example

```
MCBSP_reset(hMcbsp);
MCBSP_reset(INV);
```

MCBSP_rfull *Reads RFULL bit of serial port control register 1*

Function

CSLBool MCBSP_rfull(
    MCBSP_Handle hMcbsp
);

Arguments

hMcbsp   Handle to McBSP port obtained by MCBSP_open()

Return Value

RFULL   Returns RFULL status bit of SPCR1 register
0 - receive buffer empty
1 - receive buffer full

Description

Reads the RFULL bit of the serial port control register 1. (Both RBR and RSR are full. A receive overrun error could have occured.)

Example

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

MCBSP_rrdy

Reads RRDY status bit of SPCR1 register

Function

CSLBool MCBSP_rrdy(
    MCBSP_Handle hMcbsp
);

Arguments

hMcbsp   Handle to McBSP port obtained by MCBSP_open()

Return Value

RRDY   Returns RRDY status bit of SPCR1
0 - no new data to be received
1 - new data has been received

Description

Reads the RRDY status bit of the SPCR1 register. A 1 indicates the receiver is ready with data to be read.

Example

```
if (MCBSP_rrdy(hMcbsp)) {
val = MCBSP_read16 (hMcbsp);
}
```

MCBSP_start

Starts a transmit and/or receive operation for a MCBSP port

Function

void MCBSP_start(
    MCBSP_Handle hMcbsp,
    Uint16 startMask,
    Uint16 SampleRateGenDelay
    );

Arguments

hMcbsp   Handle to McBSP port obtained by MCBSP_open()

startMask   Start mask. It could be any of the following values (or their logical OR):
MCBSP_XMIT_START: start transmit (XRST field)
MCBSP_RCV_START: start receive (RRST field)
MCBSP_SRGR_START: start sample rate generator (GRST field)
MCBSP_SRGR_FRAMESYNC: start framesync generation (FRST field)

SampleRateGenDelay   Sample rate generates delay. MCBSP logic requires two sample_rate generator clock_periods after enabling the sample rate generator for its logic to stabilize. Use this parameter to provide the appropriate delay before starting the MCBSP.
A conservative value should be equal to:

$$SampleRateGenDelay = \frac{2 \times Sample\_Rate\_Generator\_Clock\_period}{4 \times C55x\_Instruction\_Cycle}$$

A default value of:

MCBSP_SRGR_DEFAULT_DELAY (0xFFFF value) can be used (maximum value).

Return Value None

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

Note:

If you want to configure all McBSP registers without starting the McBSP port, use MCBSP_config() without setting the SPCR2 (XRST, RRST, GRST, and FRST) fields. Then, after you write the first data valid to the DXR registers, call MCBSP_start() when ready to start the McBSP port. This guarantees that the correct value is transmitted/received.

Example 1

```
MCBSP_start(hMcbsp, MCBSP_XMIT_START, 0x3000);
...
MCBSP_start(hMcbsp, MCBSP_XMIT_START|MCBSP_SRGR_START|
             MCBSP_SRGR_FRAMESYNC, 0x1000);
```

Example 2

```
MCBSP_start(hMcbsp,
      MCBSP_SRGR_START|MCBSP_RCV_START,
      0x200
      );
```

MCBSP_write16 *Writes a 16-bit value*

Function

```
void MCBSP_write16(
    MCBSP_Handle hMcbsp,
    Uint16 Val
);
```

Arguments

hMcbsp	McBSP Device Handle obtained by MCBSP_open()
Val	16-bit value to be written

Return Value None

Description Directly writes a 16-bit value to the serial port data transmit register: DXR1.

Depending on the receive word data length you have selected in the XCR1/XCR2 registers, the actual data could be 8, 12, or 16 bits long.

This function does not verify that the transmitter is ready to transmit a new word. Use MCBSP_xrdy() (prior to calling MCBSP_write16()) for this purpose.

Example

```
Uint16 val16;
MCBSP_write16(hMcbsp, val16);
```

MCBSP_write32 *Writes a 32-bit value*

Function

```
void MCBSP_write32(
    MCBSP_Handle hMcbsp,
    Uint32 Val
);
```

Arguments

hMcbsp	McBSP Device Handle obtained by MCBSP_open()
Val	32-bit value to be written

Return Value None

Description Writes a 32-bit value. Depending on the transmit word data length you have selected in the XCR1|XCR2 registers, the actual data could be 20, 24, or 32 bits long.

This function does not check to verify that the transmitter is ready to transmit a new word. Use MCBSP_xrdy() (prior to calling MCBSP_write32()) for this purpose.

Example

```
Uint32 val32;
MCBSP_write32(hMcbsp, val32);
```

MCBSP_xempty — *Reads XEMPTY bit from SPCR2 register*

Function

```
CSLBool MCBSP_xempty(
    MCBSP_Handle hMcbsp
);
```

Arguments

hMcbsp Handle to McBSP port obtained by MCBSP_open()

Return Value

XEMPTY Returns XEMPTY bit of SPCR2 register
0 - transmit buffer empty)
1 - transmit buffer full

Description

Reads the XEMPTY bit from the SPCR2 register. A 0 indicates the transmit shift (XSR) is empty.

Example

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

MCBSP_xrdy — *Reads XRDY status bit of SPCR2 register*

Function

```
CSLBool MCBSP_xrdy(
    MCBSP_Handle hMcbsp
);
```

Arguments

hMcbsp Handle to McBSP port obtained by MCBSP_open()

Return Value

XRDY Returns XRDY status bit of SPCR2
0 - not ready to transmit new data
1 - ready to transmit new data

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);
  ...
}
```

13.5 Macros

The CSL offers a collection of macros to gain individual access to the McBSP peripheral registers and fields.

Table 13-4 lists macros available for the McBSP module using McBSP port number. Table 13-5 lists macros available for the McBSP module using handle.

Table 13-4. McBSP Macros Using McBSP Port Number

(a) Macros to read/write McBSP register values

Macro	Syntax
MCBSP_RGET()	Uint16 MCBSP_RGET(*REG#*)
MCBSP_RSET()	Void MCBSP_RSET(*REG#*, Uint16 *regval*)

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

Macro	Syntax
MCBSP_FGET()	Uint16 MCBSP_FGET(*REG#*, *FIELD*)
MCBSP_FSET()	Void MCBSP_FSET(*REG#*, *FIELD*, Uint16 *fieldval*)

(c) Macros to create a value for the McBSP registers and fields (Applies only to registers with more than one field)

Macro	Syntax
MCBSP_REG_RMK()	Uint16 MCBSP_*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
MCBSP_FMK()	Uint16 MCBSP_FMK(*REG*, *FIELD*, *fieldval*)

Table 13-4. McBSP Macros Using McBSP Port Number (Continued)

(d) Macros to read a register address

Macro	Syntax
MCBSP_ADDR()	Uint16 MCBSP_ADDR(*REG#*)

Notes:
1) REG# indicates, if applicable, a register name with the channel number (example: DMACCR0)
2) REG indicates the registers: SPCR1, SPCR2, RCR1, RCR2, XCR1, XCR2, SRGR1, SRGR2, MCR1, MCR2, RCERA, RCERB, RCERC, RCERD, RCERE, RCERF, RCERG, RCERH, XCERA, XCERB, XCERC, XCERD, XCERE, XCERF, XCERG, XCERH, PCR
3) *FIELD* indicates the register field name as specified in the *55x DSP Peripherals Reference Guide*.
   For *REG*_FSET and *REG*_FMK, *FIELD* must be a writable field.
   For *REG*_FGET, the field must be a readable field.
4) *regval* indicates the value to write in the register (*REG*).
5) *fieldval* indicates the value to write in the field (*FIELD*).

Table 13-5. McBSP CSL Macros Using Handle

(a) Macros to read/write McBSP register values

Macro	Syntax
MCBSP_RGETH()	Uint16 MCBSP_RGETH(MCBSP_Handle hMCBSP, *REG*)
MCBSP_RSETH()	Void MCBSP_RSETH(MCBSP_Handle hMCBSP, *REG*, Uint16 *regval*)

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

Macro	Syntax
MCBSP_FGETH()	Uint16 MCBSP_FGETH(MCBSP_Handle hMCBSP, *REG*, *FIELD*)
MCBSP_FSETH()	Void MCBSP_FSETH(MCBSP_Handle hMCBSP, *REG*, *FIELD*, *Uint16 fieldval*)

Table 13-5. McBSP CSL Macros Using Handle (Continued)

(c) Macros to read a register address

Macro	Syntax
MCBSP_ADDRH()	Uint16 MCBSP_ADDRH(MCBSP_Handle hMCBSP, *REG*)

Notes: 1) *REG* indicates the registers: SPCR1, SPCR2, RCR1, RCR2, XCR1, XCR2, SRGR1, SRGR2, MCR1, MCR2, RCERA, RCERB, RCERC, RCERD, RCERE, RCERF, RCERG, RCERH, XCERA, XCERB, XCERC, XCERD, XCERE, XCERF, XCERG, XCERH, PCR

2) *FIELD* indicates the register field name as specified in the *55x DSP Peripherals Reference Guide*.
For *REG*_FSETH, *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*).

13.6 Examples

Examples for the McBSP module are found in the CCS examples\<target>\csl directory.

Example 13-1 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.

Example 13-1. McBSP Port Initialization Using MCBSP_config()

```
#include <csl.h>
#include <csl_mcbsp.h>
#define N    10

/* Step 0: This is your MCBSP register configuration */

static MCBSP_Config ConfigLoopBack32= {
   ....
};


void main(void) {

  MCBSP_Handle mhMcbsp;
  Uint32 xmt[N], rcv[N];

  ....

/* Step 1: Initialize CSL */

   CSL_init();

/* Step 2: Open and configure the MCBSP port */

   mhMcbsp = MCBSP_open(MCBSP_PORT0, MCBSP_OPEN_RESET);

   MCBSP_config(mhMcbsp, &ConfigLoopBack32);

/* Step 3: Write the first data value and start */
/* the sample rate genteration in the MCBSP     */

       MCBSP_write32(mhMcbsp, xmt[0]);
       MCBSP_start(mhMcbsp,MCBSP_XMIT_START|MCBSP_RCV_START|
                           MCBSP_SRGR_START|MCBSP_SRGR_FRAMESYNC,
                           0x300u);

......

   while (!MCBSP_rrdy(mhMcbsp));
   rcv[0] = MCBSP_read32(mhMcbsp);
```

Example 13-1. McBSP Port Initialization Using MCBSP_config() (Continued)

```
/* Begin the data transfer loop of the remaining (N-1) values. */
for (i=1; i<N-1;i++)
   {

/* Wait for XRDY signal before writing data to DXR */
   while (!MCBSP_xrdy(mhMcbsp));

/* Write 32 bit data value to DXR */
   MCBSP_write32(mhMcbsp,xmt[i]);

/* Wait for RRDY signal to read data from DRR */
   while (!MCBSP_rrdy(mhMcbsp));

/* Read 32 bit value from DRR */
   rcv[i] = MCBSP_read32(mhMcbsp);
}
   MCBSP_close(mhMcbsp);

} /* main */
```

Chapter 14

MMC Module

This chapter contains descriptions of the configuration structures, data strucutres, and functions available in the multimedia card (MMC) module.

Topic **Page**

14.1 Overview

Table 14-1. MMC Configuration Structures

Config Structure	Description	See Page
MMC_Config	MMC configuration structure	14-5

Table 14-2. MMC Data Structures

Data Structure	Description	See Page
MMC_CallBackObj	Structure used to assign functions for each interrupt	14-6
MMC_CardCsdObj	Contains card specific data (CSD)	14-7
MMC_CardIdObj	Contains card identification (CID)	14-8
MMC_CardObj	Contains information about memory cards including CID and CSD	14-8
MMC_CardXCsdObj	Extended card specific data (XCSD)	14-9
MMC_Cmdobj	Structure to store commands	14-9
MMC_MmcRegObj	Structure to store values of all MMC regs	14-10
MMC_SetupNative	Native mode Initialization Structure	14-11
MMC_RspRegObj	Structure to store values of MMC response regs	14-11
MMC_SetupSpi	SPI mode Initialization Structure	14-12

Table 14-3. MMC Functions

Function	Description	See Page
MMC_clearResponse	Clears the MMC response registers	14-13
MMC_close	Frees MMC controller reserved by call to MMC_open	14-13
MMC_config	Writes the values of the configuration structure into the control registers for the specified MMC controller.	14-14
MMC_deselectCard	Deselects the given card. This function is valid in SPI mode only.	14-14
MMC_dispatch0	ISR dispatch function to service MMC0 (port0) isrs.	14-14
MMC_dispatch1	ISR dispatch function to service MMC1 (port1) isrs.	14-15

Table 14-3. MMC Functions (Continued)

Function	Description	See Page
MMC_drrdy	Returns the contents of the DRRDY status bit in the MMCST0 register.	14-15
MMC_dxrdy	Returns the contents of the DXRDY status bit in the MMCST0 register	14-15
MMC_getCardCsd	Reads the Card Specific Data from response registers.	14-16
MMC_getCardId	Reads card ID from the MMC response registers.	14-16
MMC_getConfig	Returns the current contents of the MMC control registers. This excludes the MMC response registers.	14-17
MMC_getNumberOfCards	Returns the number of cards found when MMC_open is called with MMC_OPEN_SENDALLCID option.	14-17
MMC_getSpiCid	Sends a request to card to submit its Card Identification Structure when operating in SPI mode.	14-18
MMC_getStatus	Returns the status of the specified field in the MMCST0 register.	14-18
MMC_SetupNative	Initializes the controller when in Native mode	14-11
MMC_SetupSpi	Initializes the controller when in SPI mode.	14-12
MMC_open	Reserves the MMC device specified by, device.	14-19
MMC_read	Sends commands to read blocks of data. This is a blocking function in that it does not return until all data has been transferred.	14-19
MMC_responseDone	Checks the status of a register for a response complete condition.	14-20
MMC_saveStatus	Saves current contents of MMCST0 register in MMC Handle.	14-20
MMC_selectCard	Selects card with specified relative address for communication.	14-21
MMC_sendAllCID	Sends broadcast command to all cards to identify themselves.	14-21
MMC_sendCmd	Sends a command to selected memory card/s. Optionally waits for a response.	14-22
MMC_sendCSD	Sends a request to card to submit its Card Specific Data or CSD Structure.	14-22
MMC_sendGoIdle	Sends a broadcast GO_IDLE command.	14-23
MMC_setCardPtr	Sets the card pointer in the MMC global status table.	14-23

Table 14-3. MMC Functions (Continued)

Function	Description	See Page
MMC_sendOpCond	Sets the operating voltage “window” while in Native mode; Initializes the card in SPI mode.	14-24
MMC_setCallBack	Associated functions to interrupts and installs dispatcher routines.	14-25
MMC_setChipSelect	Associates the GPIO pin with the card Chip Select. It may also open/configure the appropriate peripheral to gain control/access of the specified GPIO pin.	14-26
MMC_setRca	Set the relative card address of an attached memory card.	14-26
MMC_stop	Halts a current data transfer.	14-27
MMC_waitForFlag	Waits for a particular field in the MMCST0 register to be set.	14-27
MMC_write	Writes a block of data. This is a blocking function in that it does not return until all data has been transferred.	14-28

14.2 Configuration Structures

The section contains the configuration structures available for the MMC module.

MMC_Config *MMC Configuration Structure*

Structure void MMC_Config

Members

```
Uint16 mmcctl       /* MMC Control Register                   */
Uint16 mmcfclkctl   /* MMC Functinal Clock Control Register   */
Uint16 mmcclk       /* MMC Memory Clock Control Register      */
Uint16 mmcim        /* MMC Interrupt Enable Register          */
Uint16 mmctor       /* MMC Timeout Response Register          */
Uint16 mmctod       /* MMC Timeout Read Data Register         */
Uint16 mmcblen      /* MMC Block Length Register              */
Uint16 mmcnblk      /* MMC Number of Block Register           */
```

Description MMC Configuration Strucutre used to set up the MMC interface. You create and initialize this structure and then pass its address to the MMC_config() function.

Example

```
MMC_Config Config = {

0x000F,    /* MMCCTL  */
0x0F00,    /* MMCFCLKCTL */
0x0001,    /* MMCCLK  */
0x0FA0,    /* MMCIm   */
0x0500,    /* MMCTOR  */
0x0500,    /* MMCTOD  */
0x0200,    /* MMCBLEN */
0x0001     /* MMCNBLK */
};
```

14.3 Data Structures

This section contains the data structures available for use with the MMC module.

MMC_CallBack-Obj

Configures pointers to functions

Structure MMC_CallBackObj

Members

MMC_CallBackPtr mmcDatdneCallBack	Pointer to function for DATDNE interrupt
MMC_CallBackPtr mmcBsydneCallBack	Pointer to function for BSYDNE interrupt
MMC_CallBackPtr mmcRspdneCallBack	Pointer to function for RSPDNE interrupt
MMC_CallBackPtr mmcToutrdCallBack	Pointer to function for TOUTRD interrupt
MMC_CallBackPtr mmcToutrsCallBack	Pointer to function for TOUTRS interrupt
MMC_CallBackPtr mmcCrcwrCallBack	Pointer to function for CRCWR interrupt
MMC_CallBackPtr mmcCrcrdCallBack	Pointer to function for CRCRD interrupt
MMC_CallBackPtr mmcCrcrsCallBack	Pointer to function for CRCRS interrupt
MMC_CallBackPtr mmcDxrdyCallBack	Pointer to function for DXRDY interrupt
MMC_CallBackPtr mmcDrrdyCallBack	Pointer to function for DRRDY interrupt
MMC_CallBackPtr mmcDategCallBack	Pointer to function for DATEG interrupt

Description Configures pointers to functions.

Example None

MMC_CardCsdobj

Contains Card Specific Data (CSD)

Structure MMC_CardCsdObj

Members

Uint16 csdStructType	2-bit structure type field
Uint16 mmcProtocolVersion	2-bit MMC protocol
Uint16 dataReadAccessTime1	8-bit TAAC
Uint16 dataReadAccessTime2	8-bit NSAC
Uint16 maxDataXfrRate	8-bit max data transmission speed
Uint16 cardCommandClass	12-bit card command classes
Uint16 maxReadBlockLen	4-bit maximum Read Block Length
Uint16 allowPartialReadBlocks	1-bit indicates if partial blocks allowed
Uint16 writeBlockMisalign	1-bit flag indicates write block misalignment
Uint16 readBlockMisalign	1-bit flag indicates read block misalignment
Uint16 dsrImplemented	1-bit flag indicates whether card has DSR reg
Uint16 deviceSize	12-bit device size
Uint16 maxReadCurrVddMin	3-bit Max. Read Current @ Vdd Min
Uint16 maxReadCurrVddMax	3-bit Max. Read Current @ Vdd Max
Uint16 maxWriteCurrVddMin	3-bit Max. Write Current @ Vdd Min
Uint16 maxWriteCurrVddMax	3-bit Max. Write Current @ Vdd Max
Uint16 deviceSizeMul	3-bit device size multiplier
Uint16 eraseSectorSize	5-bit erase sector size
Uint16 eraseGroupSize	5-bit erase group size
Uint16 writeProtectGroupSize	5-bit write protect group size
Uint16 writeProtectGroupEnable	1-bit write protect enable flag
Uint16 mfgDefualtEcc	2-bit Manufacturer Default ECC
Uint16 streamWriteSpeedFac	3-bit stream write factor
Uint16 maxWriteBlockLen	4-bit maximum write block length
Uint16 allowPartialWriteBlocks	1-bit indicates if partial write blocks allowed
Uint16 copyFlag	1-bit copy flag
Uint16 permWriteProtect	1-bit to disable/enable permanent write-write protection
Uint16 tempWriteProtect	1-bit to disable/enable temporary write-write protection

Uint16 eccCode	2-bit ECC code
Uint16 crc	7-bit r/w/e redundancy check

Description Contains card specific data (CSD)

Example None

MMC_CardIdObj *Contains Card Identification (CID)*

Structure MMC_CardIdObj

Members

Uint32 mfgId	24-bit Manufacturer's ID
Char productName[9]	8-character Product Name
Uint16 hwRev	4-bit Hardware Revision Number
Uint16 fwRev	4-bit Firmware Revision Number
Uint32 serialNumber	24-bit Serial Number
Uint16 monthCode	4-bit Manufacturing Date (Month)
Uint16 yearCode	bit Manufacturing Date (Year)
Uint16 checksum	7-bit crc

Description Contains card identification.

Example None

MMC_CardObj *Contains information about Memory Cards, including CID and CSD*

Structure MMC_CardObj

Members

Uint16 rca	User assigned relative card address (RCA)
MMC mode or GPIO pin mapping associated with Chip Select in SPI mode	
Uint16 cardType	MMC or SD
Uint32 maxXfrRate	Maximum transfer rate
Uint32 readAccessTime	TAAC exp * mantissa
Uint32 cardCapacity	Total memory available on card

Uint32 lastAddrRead	Last Address Read from memory card
Uint32 lastAddrWritten	Last Address written to on memory card
MMC_CardIdObj cardId	Manufacturers Card ID
MMC_CardCsdObj csd	Card specific data
MMC_CardXCsdObj	xcsdExtended CSD

Description Contains information about Memory Cards, including CID and CSD.

Example None

MMC_CardXCs-dobj *Extended Card Specific Data (XCSD)*

Structure MMC_CardXCsdObj

Members

Uint16 securitySysId	Security System ID
Uint16 securitySysVers	Security System Version
Uint16 maxLicenses	Maximum number of storable licenses
Uint32 xStatus	Extended status bits

Description Extended card specific data.

Example None

MMC_Cmdobj *Stores an MMC Command*

Structure MMC_Cmdobj

Members

Uint16 argh	High part of command argument
Uint16 argl	Low part of command argument
Uint16 cmd	MMC command

Description Stores an MMC command.

Example None

MMC_MmcRegObj

Structure to store values of all MMC regs

Structure MMC_MmcRegObj

Members

Uint16 mmcfclkctl	MMCFCLKCTL register
Uint16 mmcctl	MMCCTL register
Uint16 mmcst0	MMCST0 register
Uint16 mmcst1	MMCST1 register
Uint16 mmcim	MMCIM register
Uint16 mmctor	MMCTOR register
Uint16 mmctod	MMCTOD register
Uint16 mmcblen	MMCBLEN register
Uint16 mmcnblk	MMCNBLK register
Uint16 mmcdrr	MMCDRR register
Uint16 mmcdxr	MMCDXR register
Uint16 mmccmd	MMCCMD register
Uint16 mmcargl	MMCARGL register
Uint16 mmcarh	MMCARGH register
MMC_RspRegObj mmcrsp	MMCRSP registers
Uint16 mmcdrsp	MMCDRSP register
Uint16 mmcetok	MMCETOK register
Uint16 mmccidx	MMCCIDX register

Description Structure to store values of all MMC regs

Example None

MMC_SetupNative

Native mode Initialization Structure

Structure MMC_SetupNative

Members

Uint16 dmaEnable	Enable/disable DMA for data read/write
Uint16 dat3EdgeDetection	Set level of edge detection for DAT3 pin
Uint16 goIdle	Determines if MMC goes IDLE during IDLE instr
Uint16 enableClkPin	Memory clk reflected on CLK Pin
Uint32 fdiv	CPU CLK to MMC function clk divide down
Uint32 cdiv	MMC func clk to memory clk divide down
Uint16 rspTimeout	Number of memory clks to wait before response timeout
Uint16 dataTimeout	Number of memory clks to wait before data timeout
uint16 blockLen	Block length must be same as CSD

Description Initialization structure for Native mode.

Example None

MMC_RspRegObj

Structure to store values of all MMC response regs

Structure MMC_RspRegObj

Members
Uint16 rsp0
Uint16 rsp1
Uint16 rsp2
Uint16 rsp3
Uint16 rsp4
Uint16 rsp5
Uint16 rsp6
Uint16 rsp7

Description Structure to store values of all MMC response regs

Example None

MMC_SetupSpi *SPI mode Initialization Structure*

Structure MMC_SpiInitObj

Members

Uint16 dmaEnable	Enable/disable DMA for data read/write
Uint16 dat3EdgeDetection	Set level of edge detection for DAT3 pin
Uint16 goIdle	Determines if MMC goes IDLE during IDLE instr
Uint16 enableClkPin	Memory clk reflected on CLK Pin
Uint32 fdiv	CPU CLK to MMC function clk divide down
Uint32 cdiv	MMC func clk to memory clk divide down
Uint16 rspTimeout	Number of memory clks to wait before response timeout
Uint16 dataTimeout	Number of memory clks to wait before data timeout
uint16 spiCrc	Enable/disable CRC checking

Description Initialization structure for SPI Mode.

Example None

14.4 Functions

MMC_clrResponse *Clears the contents of the MMC response registers*

Function

Void MMC_clearResponse(
MMC_Handle mmc
);

Arguments

mmc MMC Handle returned by call to MMC_open

Description

Clears the contents of the MMC response registers.

Example

```
MMC_Handle myMmc;
Uint16 rca = 2;
Uint16 waitForRsp = TRUE;
MyMmc = MMC_open(MMC_DEV1);
.
.
.
MMC_clrResponse(myMmc);
MMC_sendCmd(MyMmc,  MMC_SEND_CID, waitForRsp, rca);
```

MMC_close *Closes/frees the MMC device*

Function

void MMC_close(
MMC_Handle mmc
);

Arguments

mmc MMC Handle returned by call to MMC_open

Description

Closes/frees the MMC device reserved by previous call to MMC_open.

Example

```
MMC_Handle myMmc;
MyMmc = MMC_open(MMC_DEV0);
.
.
.
MMC_close(myMmc);
```

MMC_config

Writes the values of configuration structures for MMC controllers

Function

```
void MMC_config(
MMC_Handle mmc,
MMC_Config *mmcCfg
);
```

Arguments

mmc	MMC handle returned call to MMC_open.
mmcCfg	Pointer to user defined MMC configuration structure which contains the values to set the MMC control registers.

Description

Configures the MMC controller by writing the specified values to the MMC control registers. Calls to this function are unnecessary if you have called the MMC_open function using any of the MMC_OPEN_INIT_XXX flags and have set the needed configuration parameters in the MMC_InitObj structure.

Example

```
MMC_config(myMMC, &myMMCCfg);
```

MMC_deselect-Card

Deselects the given card

Function

```
void MMC_deselectCard(
MMC_Handle mmc
MMC_cardObj *card
);
```

Arguments

mmc	MMC Handle returned by call to MMC_open
card	Pointer to card object

Description

Deselects the given card. This function is valid in SPI mode only.

Example

```
MMC_Handle myMmc;
MMC_cardObj *card;

myMmc = MMC_open(MMC_DEV1);
```

MMC_dispatch0

ISR dispatch function to service the MMC0 isrs

Function

```
void MMC_dispatch0(
);
```

Arguments

None

Description

Interrupt service routine dispatch function to service interrupts that occur on MMC port 0.

Example

```
MMC_dispatch0();
```

MMC_dispatch1

ISR dispatch function to service the MMC1 isrs

Function

void MMC_dispatch1(
);

Arguments

None

Description

Interrupt service routine dispatch function to service interrupts that occur on MMC port 1.

Example

```
MMC_dispatch1();
```

MMC_drrdy

Returns the DRRDY status bit

Function

int MMC_drrdy(
MMC_Handle myMmc
);

Arguments

mmc   MMC Handle returned by call to MMC_open

Description

Returns the value of the DRRDY field in the MMCST0 register.

Example

```
MMC_Handle myMmc;
int i;
.
.
.
i = MMC_drrdy(myMmc);
```

MMC_dxrdy

Returns the DXRDY status bit

Function

int MMC_dxrdy(
MMC_Handle mmc
);

Arguments

mmc   MMC Handle returned by call to MMC_open

Description

Returns the value of the DXRDY field in the MMCST0 register.

Example

```
MMC_Handle myMmc;
int i;
.
.
.
i = MMC_dxrdy(myMmc);
```

MMC_getCardCSD

Reads card specific data from response registers

Function

void MMC_getCardCSD(
MMC_Handle mmc,
MMC_CardCSD Obj *csd);

Arguments

mmc	MMC Handle returned by call to MMC_open
csd	Pointer to Card Specific Data object

Description

Parses CSD data from response registers. MMC_getCardCSD verifies that the SEND_CSD command has been issued and the response is complete.

Example

```
MMC_Handle myMmc;
MMC_CardCsd Obj *csd;
.
.
.
MMC_sendCSD(myMmc);
MMC_getCardCSD(myMmc, csd);
```

MMC_getCardId

Reads card ID from the MMC response registers

Function

Void MMC_getCardId(
MMC_Handle mmc,
MMC_CardIdObj *cardId
)

Arguments

mmc	MMC Handle returned by call to MMC_open
cardId	Pointer to user defined memory card ID object.

Description

Parses memory card ID from contents of the MMC controller response registers and returns the card identity in the given card ID object.

Example

```
MMC_Handle myMmc;
MMC_CardIdObj myCardId;
myMmc = MMC_open(MMC_DEV1);
.
.
MMC_getCardId(myMmc,&myCardId);
```

MMC_getConfig

Returns the current contents of the MMC conrtrol registers

Function

Void MMC_getConfig(
MMC_Handle mmc,
MMC_Config *mmcCfg
);

Arguments

mmc	MMC_Handle returned from a call to MMC_open.
mmcCfg	Pointer to a user defined MMC configuration structure where current values of the MMC control registers will be returned.

Description

Returns the values of the MMC control registers in the specified MMC configuration structure.

Example

```
MMC_getConfig(myMMC, &myMMcCfg);
```

MMC_getNumberOfCards

Returns the number of cards found when MMC_Open is called

Function

Uint16 MMC_getNumberOfCards(
MMC_Handle mmc,
Uint16 *active,
Uint16 *inactive
);

Arguments

mmc	MMC Handle returned by call to MMC_open.
active	Pointer to where to return number of active cards.
inactive	Pointer to where to return number of inactive cards.

Description

Returns the number of cards found when MMC_open is called with the MMC_OPEN_SENDALLCID option.

Example

```
MMC_Handle myMmc;
MMC_InitObj myMmcInit;
Uint16 n;
Uint16 active[i] = {0};
Uint16 inactive[i] = {0};

MyMmc = MMC_open(MMC_DEV1);

n = MMC_getNumberOfCards(myMmc, active, inactive);
```

MMC_getSpiCid

Retrieves the Card Indentification Structure in SPI mode

Function

int MMC_getSpiCid(
MMC_Handle mmc,
MMC_cardIdObj *cid
);

Arguments

mmc	MMC_Handle returned from a call to MMC_open
cid	Pointer to card identification object

Description

Sends a request to card in the identification process to submit its Card Identification Structure when operating in SPI mode.

Example

```
MMC_Handle myMmc;
MMC_cardIdObj *cid;
.
.
.
MMC_getSpiCid(myMmc, cid);
```

MMC_getStatus

Returns the status of a specified field in the status register

Function

int MMC_getStatus(
MMC_Handle mm,
Uint32 lmask
);

Arguments

mmc	MMC Handle returned by call to MMC_open
lmask	Mask of the status flags to check

Description

Returns the contents of status registers

Example

```
MMC_Handle myMmc;
Uint16 ready;

read = MMC_getStatus(myMmc, MMC_ST0_DXRDY);
```

MMC_open

Reserves the MMC device as specified by a device

Function

MMC_Handle MMC_open(
    Uint16 device,
);

Arguments

device	Device (port) number. It can be one of the following:
	MMC_DEV0 (also called MMC_PORT1)
	MMC_DEV1 (also called MMC_PORT2)

Description

MMC_open performs the following tasks:

1. Reserves the specified MMC controller and corresponding MMC port.
2. Enables controller access by setting appropriate bits in the External Bus Selection register.

Example

```
MMC_Handle myMmC;

myMmc = MMC_open(MMC_DEV0);
```

MMC_read

Reads a block of data from a pre-selected memory card

Function

void MMC_read(
MMC_Handle mmc,
Uint32 cardAddr,
Void *buffer,
Uint16 buflen
);

Arguments

mmc	MMC Handle returned by call to MMC_open.
cardAddr	Address on card where read begins.
buffer	Pointer to buffer where received data should be stored.
buflen	number of elements to store in buffer.

Description

Reads a block of data from the pre-selected memory card (see MMC_selectCard) and stores the information in the specified buffer.

Example

```
MMC_Handle myMmc;
Uint16 mybuf[512];

MyMmc = MMC_open(MMC_DEV1);
.
.
.
MMC_read(myMmc, 0, mybuf, 512);
```

MMC_responseDone

Checks status register for Response Done condition

Function

int MMC_responseDone(
MMC)Handle mmc
);

Arguments

mmc   MMC Handle returned by call to MMC_open

Description

Checks the status of register MMCST0 for response done (RSPDONE) condition. If a timeout occurs before the response done flag is set, the function returns an error condition of 0xFFFF = MMC_RESPONSE_TIMEOUT.

Example

```
MMC_Handle myMmc;
.
.
./* wait for response done */
while  ((sfd = MMC_responseDone (myMmc))==0){
}
      if(sfd == MMC_RESPONSE_TIMEOUT)
         return 0;
```

MMC_saveStatus

Saves the current status of MMC

Function

int MMC_saveStatus(
MMC_Handle mmc
);

Arguments

mmc   MMC Handle returned by call to MMC_open

Description

Saves the current contents of the MMCST0 register in the MMC Handle.

Example

```
MMC_Handle myMmc;
.
.
.
MMC_saveStatus(myMmc);
```

MMC_selectCard

Selects card with specified relative address for communication

Function

Int MMC_selectCard(
    MMC_Handle mmc;
    MMC_CardObj *card
)

Arguments

mmc	MMC Handle returned from MMC_open
card	Pointer to card object

Description

Selects card with specified relative address for communication.

Example

```
MMC_InitObj myMmcInit;
MMC_Handle myMmc;
MMC_CardObj card;
Uint16 rca = 2;

myMmc = MMC_open(MMC_DEV1,);

MMC_selectCard(myMmc, &card);
```

MMC_sendAllCID

Sends a broadcast command to all cards to identify themselves

Function

void MMC_sendAllCID(
MMC_Handle mmc,
MMC_CardId Obj *cid
);

Arguments

mmc	MMC Handle returned by call to MMC_open
cid	Pointer to card ID object

Description

This function sends the MMC_SEND_ALL_CID command to initiate identification of all memory cards attached to the controller. If a response is sent from a card, it returns the information about that card in the specified cardId object.

Example

```
MMC_Handle myMmc;
MMC_CardIdObj myCardId;
myMmc = MMC_open(MMC_DEV1, MMC_OPEN_ONLY);
.
.
MMC_SendAllCID(myMmc, &myCardID);
```

MMC_sendCmd

Sends commands to selected memory cards.

Function

void MMC_sendCmd(
MMC_Handle mmc,
Uint16 cmd,
Uint16 argh,
Uint16 argh,
Uint16 waitForRsp,
);

Arguments

mmc	MMC Handle returned from call to MMC_open
cmd	Command to send to memory card.
argh	Upper 16 bits of argument
argl	Lower 16 bits of argument
waitForRsp	Boolean. TRUE, if function should wait for response from card,FALSE otherwise.
	variable length set of arguments for specified command

Description

Function sends the specified command to the memory card associated with the given relative card address. Optionally, the function will wait for a response from the card before returning.

Example

```
MMC_Handle myMmc;
myMmc = MMC_open(MMC_DEV0);
.
.
.
MMC_SendCmd(myMmc, MMC_GO_IDLE_STATE, 0, 0, 1);
```

MMC_sendCSD

Sends a request to card to submit its CSD structure

Function

int MMC_sendCSD(
MMC_Handle mmc
);

Arguments

mmc MMC_Handle returned from a call to MMC_open

Description

Sends a request to card in the identification process to submit its Card Specific Data Structures.

Example

```
MMC_Handle myMmc;
.
.
.
MMC_sendCSD(myMmc);
```

MMC_sendGoIdle

Sends a broadcast GO_IDLE command

Function

void MMC_sendGoIdle(
MMC_Handle mmc
);

Arguments

mmc MMC_Handle returned from a call to MMC_open

Description

Sends a broadcast GO_IDLE command

Example

```
MMC_Handle myMmc;
.
.
.
MMC_sendGoIdle(myMmc);
```

MMC_setCardPtr

Sets the card pointer in the MMC global status table

Function

void MMC_setCardPtr(
MMC_Handle mmc,
MMC_cardObj *card
);

Arguments

mmc MMC_Handle returned from a call to MMC_open
card Pointer to card objects

Description

Sets the card pointer in the MMC global status table. This function must be used if the application performs a system/card initialization outside of the MMC_initCard function.

Example

```
MMC_Handle myMmc;
MMC_cardObj *card;

MMC_setCardPtr(myMmc, &card);
```

MMC_sendOpCond

Sends the SEND_OP_COND command to a card

Function

```
int MMC_sendOpCond(
MMC_Handle mmc,
Uint32 hvddMask
);
```

Arguments

mmc	MMC Handle returned by call to MMC_open
hvddMask	Mask used to set operating voltage conditions in native mode

Description

Sets the operating condition in native mode and initializes the card in SPI mode.

hvddMask is not affected if the SPI mode is enabled.

Table 14-4. OCR Register Definitions

OCR Bit	VDD Voltage Window
0-7	Reserved
8	2.0-2.1
9	2.1-2.2
10	2.2-2.3
11	2.3-2.4
12	2.4-2.5
13	2.5-2.6
14	2.6-2.7
15	2.7-2.8
16	2.8-2.9
17	2.9-3.0
18	3.0-3.1
19	3.1-3.2
20	3.2-3.3
21	3.3-3.4
22	3.4-3.5
23	3.5-3.6
24-30	reserved
31	Card power-up status bit (busy)

Example

```
MMC_Handle myMmc;
.
.
.
/* enables 3.2-3.3V of operating voltage by setting bit 20 */
MMC_sendOpCond(myMmc, 0x00100000)
```

MMC_setCallBack

Associates functions to interrupts and installs dispatcher routines

Function

void MMC_setCallBack(
MMC_Handle mmc,
Uint16 enableMask,
MMC_callBackObj *callbackfuncs
);

Arguments

mmc	MMC_Handle returned from a call to MMC_open
enableMask	mask to enable interrupts in the MMCIE register
callbackfuncs	Pointer to MMC_callBackObj containing a predefined set of functions to call to service flagged MMC interrupts.

Description

MMC_setCallBack associates each function to one of the MMC interrupts and installs the MMC dispatcher routine address in the MMC interrupt vector.

Example

```
MMC_Handle myMmc;
MMC_callBackObj *callback;
.
.
.
MMC_setCallBack(myMmc, 0x1000, &callback);
```

MMC_setChipSelect *Associates the GPIO pin with the card Chip select*

Function

Void MMC_setChipSelect(
MMC_Handle mmc,
Uint16 gpioPin;
MMC_CardObj *card;
);

Arguments

mmc	MMC Handle returned by call to MMC_open
gpioPin	GPIO pin to associated with Chip Select for this card.
card	Pointer to card object.

Description

Sets the rca field in the card object associating the given GPIO pin with the card. It may also open and initialize any devices that may be associated with the given GPIO pin.

Example

```
MMC_Handle myMmc;
MMC_CardObj card0;

myMmc = MMC_open(MMC_DEV1);
.
.
.
MMC_setChipSelect(myMmc,MMC_GPIO0, &card0);
```

MMC_setRca *Sets the relative card address of an attatched memory card*

Function

void MMC_setRca(
MMC_Handle mmc,
MMC_CardObj *card,
Uint16 rca
);

Arguments

mmc	MMC Handle returned by call to MMC_open
card	Pointer to card object
Rca	Relative card address

Description

Sends command to set a card’s relative card address.

Example

```
MMC_Handle myMmc;
MMC_CardObj *card;
myMmc = MMC_open(MMC_DEV0);
.
```

```
.
.
MMC_sendAllCid(myMmc, &cardid);
.
.
.
MMC_setRca(myMmc, card, 2);
```

MMC_stop

Halts a current data transfer

Function

int MMC_stop(
MMC_Handle mmc
);

Arguments

mmc	MMC_Handle returned from a call to MMC_open

Description

Halts a current data transfer by issuing the MMC_STOP_TRANSMISSION command.

Example

```
MMC_Handle myMmc;
.
.
.
MMC_stop(myMmc);
```

MMC_waitForFlag

Waits for specified flags to be set in the status register

Function

int MMC_waitForFlag(
MMC_Handle mmc,
Uint16 mask
);

Arguments

mmc	MMC Handle returned by call to MMC_open
mask	Mask of the status flags wait for (ST0)

Description

Waits for specified flags to be set in the status register

Example

```
MMC_Handle myMmc;
.
.
.
MMC_waitForFlag(myMmc, 0x0100);
```

MMC_write

Writes a block of data to a pre-selected memory card

Function

void MMC_write(
MMC_Handle mmc,
Uint32 cardAddr,
Void *buffer,
Uint16 buflen
);

Arguments

mmc	MMC Handle returned by call to MMC_open
cardAddr	Address on card where read begins.
buffer	Pointer to buffer where received data should be stored.
buflen	number of elements to store in buffer.

Description

Writes a block of data to the pre-selected memory card.

Example

```
MMC_Handle myMmc;
Uint16 mybuf[512];

myMmc = MMC_open(MMC_DEV1);
.
.
.
MMC_write(myMmc, 0, mybuf, 512);
```

Chapter 15

PLL Module

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

Topic **Page**

15.1 Overview

The CSL PLL module offers functions and macros to control the Phase Locked Loop of the C55xx.

The PLL module is not handle-based.

Table 15-1 lists the configuration structure used to set up the PLL module.

Table 15-2 lists the functions available for use with the PLL module.

Table 15-3 lists PLL registers and fields.

Section 15.4 includes a description of available PLL macros.

Table 15-1. PLL Configuration Structure

Syntax	Description	See page ...
PLL_Config	PLL configuration structure used to set up the PLL interface	15-4

Table 15-2. PLL Functions

Syntax	Description	See page ...
PLL_config()	Sets up PLL using configuration structure (PLL_Config)	15-5
PLL_setFreq()	Initializes the PLL to produce the desired CPU (core)/Fast peripherals/Slow peripherals/EMIF output frequency	15-6

Table 15-3. PLL Registers

Register	Field
CLKMD	PLLENABLE, PLLDIV, PLLMULT, VCOONOFF
For C5502 Only	
PLLCSR	PLLEN, PLLPWRDN, OSCPWRDN, PLLRST, LOCK, STABLE
PLLM	PLLM
PLLDIV0	PLLDIV0, D0EN
PLLDIV1	PLLDIV1, D1EN
PLLDIV2	PLLDIV2, D2EN
PLLDIV3	PLLDIV3, D3EN
OSCDIV1	OSCDIV1, OD1EN
WAKEUP	WKEN0, WKEN1, WKEN2, WKEN3
CLKMD	CLKMD0
CLKOUTSR	CLKOUTDIS, CLKOSEL

Note: R = Read Only; W = Write; By default, most fields are Read/Write

15.2 Configuration Structures

The following is the configuration structure used to set up the PLL.

PLL_Config *PLL configuration structure used to set up PLL interface*

Structure PLL_Config

Members For devices having a digital PLL:

Uint16 iai	Initialize After Idle
Uint16 iob	Initialize On Break
Uint16 pllmult	PLL Multiply value
Uint16 div	PLL Divide value

For devices having an analog PLL (5510PG1_2 only):

Uint16 vcoonoff	APLL Voltage-controlled oscillator control
Uint16 pllmult	APLL Multiply value
Uint16 div	APLL Divide value

For 5502 device only:

Uint16 pllcsr	// PLL Control Register
Uint16 pllm	// Clock 0 Multiplier Register
Uint16 plldiv0	// Clock 0 Divide Down Register
Uint16 plldiv1	// Sysclk 1 Divide Down Register
Uint16 plldiv2	// Sysclk 1 Divide Down Register
Uint16 plldiv3	// Sysclk3 Divide Down Register
Uint16 oscdiv1	// Oscillator divide down register
Uint16 wken	// Oscillator Wakeup Control Register
Uint16 clkmd	// Clock Mode Control Register
Uint16 clkoutsr	// CLKOUT Select Register

Description The PLL configuration structure is 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_RMK* macros to create the structure member values.

Example

```
PLL_Config Config1 = {
   1,      /* iai     */
   1,      /* iob     */
   31,     /* pllmult */
   3       /* div     */
}
```

15.3 Functions

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

PLL_config *Writes value to 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_Config).

Example

1. /* Using PLL_config function and PLL_Config structure for Digital PLL*/

```
PLL_Config MyConfig = {
1, /* iai */
1, /* iab */
31, /* pllmult */
3 /* div */
    };
```

2. /* Using PLL_config function and PLL_Config structure for 5502 PLL*/

```
      PLL_Config MyConfig = {
            0x0,    /* PLLCSR  */
            0xA,    /* PLLM    */
            0x8001, /* PLLDIV0 */
            0x8003, /* PLLDIV1 */
            0x8003, /* PLLDIV2 */
            0x8003, /* PLLDIV3 */
            0x0,    /* OSCDIV1 */
            0x0,    /* WAKEUP  */
            0x0,    /* CLKMD   */
            0x2     /* CLKOUTSR */
              };
PLL_config(&MyConfig);
```

PLL_setFreq

Initializes the PLL to produce the desired CPU output frequency

Function

void PLL_setFreq (Uint16 mul, Uint16 div);

(For C5502 device Only):
void PLL_setFreq (Uint16 mode, Uint16 mul, Uint16 div0, Uint16 div1,
Uint16 div2,Uint16 div3, Uint16 oscdiv);

Arguments

```
Uint16 mode     // PLL mode
                //PLL_PLLCSR_PLLEN_BYPASS_MODE
                //PLL_PLLCSR_PLLEN_PLL_MODE
Uint16 mul      // Multiply factor,  Valid values are (multiply by) 2 to 15.
Uint16 div0     // Sysclk 0 Divide Down, Valid values are 0, (divide by 1)
                //to 31 (divide by 32)
Uint16 div1     // Sysclk1 Divider, Valid values are 0, 1, and 3 corresponding
                //to divide by 1, 2, and 4 respectively
Uint16 div2     // Sysclk2 Divider, Valid values are 0, 1, and 3
                //corresponding to divide by 1, 2, and 4 respectively
Uint16 div3     // Sysclk3 Divider, Valid values are 0, 1 and 3
                //corresponding to divide by 1, 2 and 4 respectively
Uint16 oscdiv   // CLKOUT3(DSP core clock) divider,Valid values are 0
                //(divide by 1) to 31 (divide by 32)
```

Return Value

None

Description

Initializes the PLL to produce the desired CPU output frequency (clkout)

Example

1.
```
/* Using PLL_setFreq for devices other than 5502 */
PLL_setFreq (1, 2); // set clkout = 1/2 clkin
```

2.
```
/* Using PLL_setFreq for 5502 device  */
/*
   mode = 1 means PLL enabled (non-bypass mode)
   mul  = 5 means multiply by 5
   div0 = 0 means Divider0 divides by 1
   div1 = 3 means Divider1 divides by 4
   div2 = 3 means Divider2 divides by 4
   div3 = 3 means Divider3 divides by 4
   oscdiv = 1 means Oscillator Divider1 divides by 2
*/
PLL_setFreq(1, 5, 0, 3, 3, 3, 1);
```

15.4 Macros

The CSL offers a collection of macros to gain individual access to the PLL peripheral registers and fields.

Table 15-4 contains a list of macros available for the PLL module. To use them, include “csl_pll.h.”

Table 15-4. PLL CSL Macros Using PLL 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*)

Notes:
1) *REG* indicates the register, CLKMD.
2) *FIELD* indicates the register field name.
   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*).

Chapter 16

PWR Module

This chapter describes the PWR module, lists the API functions and macros within the module, and provides a PWR API reference section. The CSL PWR module offers functions to select which section in the device will power-down during an IDLE execution.

Topic **Page**

16.1 Overview

The CSL PWR module offers functions to control the power consumption of different sections in the C55x device. The PWR module is not handle-based.

Table 16-1 lists the functions for use with the PWR modules that order specific parts of the C55x to power down.

Table 16-2 lists DMA registers and fields.

Table 16-1. PWR Functions

Functions	Purpose	See page ...
PWR_powerDown (only for C5509 and C5510)	Forces the DSP to enter a power-down (IDLE) state	16-3

16.1.1 PWR Registers

Table 16-2. PWR Registers

Register	Field
Only for C5509 and C5510	
ICR	EMIFI, CLKGENI, PERI, CACHEI, DMAI, CPUI
ISTR	EMIFIS, CLKGENIS, PERIS, CACHEIS, DMAIS, CPUIS
Only for C5502	
ICR	IPORTI,MPORTI,XPORTI,EMIFI,CLKI,PERI,ICACHEI,MPI,CPUI
ISTR	IPORTIS,MPORTIS,XPORTIS,EMIFIS,CLKIS,PERIS,ICACHEIS,MPIS,CPUIS
PICR	MISC,EMIF,BIOST,WDT,PIO,URT,I2C,ID,IO,SP2,SP1,SP0,TIM1,TIM0
PISTR	MISC,EMIF,BIOST,WDT,PIO,URT,I2C,ID,IO,SP2,SP1,SP0,TIM1,TIM0
MICR	HPI,DMA

Note: R = Read Only; W = Write; By default, most fields are Read/Write

16.2 Functions

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

PWR_powerDown *Forces DSP to enter power-down state (On C5509 and C5510 only)*

Function

void PWR_powerDown (Uint16 wakeUpMode)

Arguments

wakeupMode

PWR_WAKEUP_MI wakes up with an unmasked interrupt and jump to execute the ISRs executed.

PWR_WAKEUP_NMI wakes up with an unmasked interrupt and executes the next following instruction (interrupt is not taken).

Return Value

None

Description

This function will Power-down the device in different power-down and wake-up modes by setting the C55x ICR register and invoking the IDLE instruction.

Example

```
/* This function will power-down the McBSP2 */
 /*and wake-up with an unmasked interrupt    */
PWR_FSET(ICR, PERI, 1);
MCBSP_FSET(PCR2, IDLEEN, 1);
PWR_powerDown(PWR_WAKEUP_MI);
```

16.3 Macros

The CSL offers a collection of macros to gain individual access to the PWR peripheral registers and fields..

Table 16-3 contains a list of macros available for the PWR module. To use them, include “csl_pwr.h.”

Table 16-3. PWR CSL Macros

(a) Macros to read/write PWR register values

Macro	Syntax
PWR_RGET()	Uint16 PWR_RGET(*REG*)
PWR_RSET()	Void PWR_RSET(*REG*, Uint16 *regval*)

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

Macro	Syntax
PWR_FGET()	Uint16 PWR_FGET(*REG*, *FIELD*)
PWR_FSET()	Void PWR_FSET(*REG*, *FIELD*, Uint16 *fieldval*)

(c) Macros to create value to PWR registers and fields (Applies only to registers with more than one field)

Macro	Syntax
PWR_*REG*_RMK()	Uint16 PWR_*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
PWR_FMK()	Uint16 PWR_FMK(*REG*, *FIELD*, *fieldval*)

(d) Macros to read a register address

Macro	Syntax
PWR_ADDR()	Uint16 PWR_ADDR(*REG*)

Notes:
1) *REG* indicates the register, ICR, ISTR
2) *FIELD* indicates the register field name as specified in the *55x DSP Peripherals Reference Guide*.
   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*).

Chapter 17

RTC Module

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

Topic **Page**

17.1 Overview

The real-time clock (RTC) provides the following features:

- 100-year calendar up to year 2099
- Counts seconds, minutes, hours, day of the week, date, month, and year with leap year compensation
- Binary-coded-decimal (BCD) representation of time, calendar, and alarm
- 12-hour (with AM and PM in 12-hour mode) or 24-hour clock modes. CSL supports only 24-hour mode.
- Second, minute, hour, or day alarm interrupts
- Update Cycle interrupt and periodic interrupts

The RTC has a separate clock domain and power supply. The clock is derived from the external 32 KHz crystal.

The configuration of the RTC can be performed by using one of the following methods:

- Register-based configuration

  A register-based configuration can be performed by calling either RTC_config(), or any of the SET register/field macros.

- Parameter-based configuration

  A parameter based configuration can be performed by calling the functions listed in Table 17-1, such as RTC_setTime(), RTC_setAlarm().

  Compared to the register-based approach, this method provides a higher level of abstraction. The downside is larger code size and higher cycle counts.

- ANSI C-Style Time Configuration

  Time functions are provided for the RTC module, which performs the same functions as the ANSI C-style standard time functions. The time is obtained, however, from the RTC. Table 17-3 contains the a list and descriptions of the RTC ANSI C-style functions. For a complete description of the functions, the arguments and structures they use please refer to the *TMS320C55x Optimizing Compiler User's Guide* (SPRU281).

Table 17-1 lists the configuration structures used to set up the RTC.

Table 17-2 and Table 17-3 lists the functions available for use with the RTC.

Table 17-4 lists macros for the RTC.

Table 17-5 lists RTC registers and fields.

Table 17-1. RTC Configuration Structures

Configuration Structure	Description	See page ...
RTC_Alarm	Structure used to set RTC Time	17-6
RTC_Config	RTC register Configuration Structure	17-7
RTC_Date	Structure used to set RTC Calendar	17-7
RTC_IsrAddr	Structure to set the RTC callback function	17-8
RTC_Time	Structure used to set RTC Alarm Time	17-8

Table 17-2. RTC Functions

Function	Description	See page ...
RTC_bcdToDec	Changes BCD value to a hexadecimal value	17-9
RTC_config	Writes value to initialize RTC using the RTC register Configuration Structure	17-9
RTC_decToBcd	Changes decimal value to BCD value	17-9
RTC_eventDisable	Disables interrupt event specified by the argument	17-10
RTC_eventEnable	Enables RTC interrupt event specified by an argument	17-10
RTC_getConfig	Reads the RTC registers into the RTC register Configuration Structure	17-10
RTC_getDate	Reads current date from RTC Registers	17-11
RTC_getEventId	Obtains IRQ module event ID for RTC	17-11
RTC_getTime	Reads current time from RTC Registers, in a 24-hour format	17-11
RTC_reset	Sets the RTC register to the default (power-on) values	17-12
RTC_setAlarm	Sets alarm to a specific time	17-12
RTC_setCallback	Associates each function to one of the RTC interrupts	17-13
RTC_setDate	Sets RTC Calendar	17-13
RTC_setPeriodicInterval	Sets periodic interrupt rate	17-14
RTC_setTime	Sets time registers	17-14

Table 17-2. RTC Functions(Continued)

Function	Description	See page ...
RTC_start	Instructs the RTC to begin running	17-15
RTC_stop	Stops the RTC	17-15

Table 17-3. RTC ANSI C-Style Time Functions

Function	Description
RTC_asctime	Converts a time to an ASCII string
RTC_ctime	Converts calendar time to local time
RTC_difftime	Returns the difference between two calendar times
RTC_gmtime	Converts calendar time to GMT
RTC_localtime	Converts calendar time to local time
RTC_mktime	Converts local time to calendar time
RTC_strftime	Formats a time into a character string
RTC_time	Returns the current RTC time and date

Note: For documentation on these functions, please refer to the ANSI C equivalent routines in the *TMS320C55x Optimizing C Compiler User's Guide* (SPRU281).

Table 17-4. RTC Macros

Macro	Description	See page ...
RTC_Addr	Reads register address	17-16
RTC_FGET	Reads RTC register field values	17-16
RTC_FSET	Writes RTC register field values	17-16
RTC_REG_FMK	Creates value of RTC register fields	17-16
RTC_REG_RMK	Creates value of RTC registers	17-17
RTC_RGET	Reads RTC register values	17-17
RTC_RSET	Writes RTC register values	17-17

Table 17-5. Registers

Register	Field
RTCSEC	SEC
RTCSECA	SAR
RTCMIN	MIN
RTCMINA	MAR
RTCHOUR	HR, AMPM
RTCHOURA	HAR, AMPM
RTCDAYW	DAY, DAEN, DAR
RTCDAYM	DATE
RTCMONTH	MONTH
RTCYEAR	YEAR
RTCPINTR	RS, (R)UIP
RTCINTEN	TM, UIE, AIE, PIE, SET
RTCINTFL	UF, AF, PF, (R)IRQF

Note: R = Read Only; W = Write; By default, most fields are Read/Write

17.2 Configuration Structures

The following is the configuration structure used to set up the RTC.

RTC_Alarm *Structure used to set RTC time*

Structure RTC_Alarm

Members

Uint16 alhour	Alarm hour (Range: 0x00-0x23 for BCD, for 24-hour format. (12-hour format is not supported.)
Uint16 alminute	Alarm Minute (Range: 0x00-0x59 for BCD)
Uint16 alsecond	Alarm Second (Range:0x00-0x59 for BCD)
Uint16 aldayw	Alarm day of the week. This member is ignored if the Periodic Weekly Alarm bit (DAEN) is set to 0. In this case, the alarm will occur in the current day.

You can use the “DONTCARE” value for each of the structure’s member if you want to set a periodic alarm for that specific interval. For example, using the DONTCARE value in the alminute field will generate an alarm interrupt every minute.

Note: Due to hardware limitations, after the first period, the *every second* periodic alarm does not produce an interrupt. To generate an alarm every second, use instead the update interrupt.

Description Structure used to set the RTC time. After it is created and initialized, the structure is passed to the RTC_setAlarm() function. The values of the structure must be entered in BCD format. You can use the RTC_decToBcd() and RTC_bcdToDec() functions to switch between decimal and BCD values.

RTC_Config *RTC configuration structure*

Structure RTC_Config

Members

Uint16 rtcsec	Seconds Register
Uint16 rtcseca	Seconds Alarm Register
Uint16 rtcmin	Minutes Register
Uint16 rtcmina	Minutes Alarm Register
Uint16 rtchour	Hour Register
Uint16 rtchoura	Hour Alarm Register
Uint16 rtcdayw	Day of the Week and Day Alarm Register
Uint16 rtcdaym	Day of the Month (Date) Register
Uint16 rtcmonth	Month Register
Uint16 rtcyear	Year Register
Uint16 rtcpintr	Periodic Interrupt selection Register
Uint16 rtcinten	Interrupt Enable Register

Description RTC configuration structure. This structure is created and initialized, and then passed to the RTC_Config() function.

The values put in the structure can be literal values or values created by RTC_REG_RMK macro. For the hour registers, the supported mode is 24-hour. The values of all time, alarm, and calendar fields must be entered in BCD format. You can use the RTC_decToBcd() and RTC_bcdToDec() functions to switch between decimal and BCD values.

RTC_Date *Structure used to set RTC calendar*

Structure RTC_Date

Members

Uint16 year	Current year (Range: 0x00–0x99 for BCD)
Uint16 month	Current month (Range: 0x01-0x12 for BCD)
Uint16 daym	Day of the month, or date (Range: 0x01-0x31 for BCD)
Uint16 dayw	Day of the week (Range 1–7, where Sunday is 1)

Description Structure used to set the RTC calendar. After it is created and initialized, the structure is passed to the RTC_setDate() function. The values of the structure must be entered in BCD format. You can use the RTC_decToBcd() and RTC_bcdToDec() functions to switch between decimal and BCD values.

RTC_IsrAddr — *Structure used to set the RTC callback function*

Structure RTC_IsrAddr

Members

void (*periodicAddr)(void)	Pointer to the function called when a periodic interrupt occurs.
void (*alarmAddr)(void)	Pointer to the function called when an alarm interrupt occurs.
void (*updateAddr)(void)	Pointer to the function called when an update interrupt occurs.

Description This structure is used to set the RTC callback function. After it is created and initialized, the structure is passed to RTC_setCallback() function. The values of the structure are pointers to the functions that are executed when the corresponding interrupt is enabled. The functions should not be declared with the *interrupt* keyword.

RTC_Time — *Structure used to set RTC time*

Structure RTC_Time

Members

Uint16 hour	Current time (Range: 0x00-0x23 for BCD, for 24-hour format. 12-hour format is not supported.)
Uint16 minute	Current Minute (Range: 0x00-0x59 for BCD)
Uint16 second	Second (Range: 0x00-0x59 for BCD)

Description Structure used to set the RTC time. After it is created and initialized, the structure is passed to the RTC_setTime() function. The values of the structure must be entered in BCD format. You can use the RTC_decToBcd() and RTC_bcdToDec() functions to switch between decimal and BCD values.

17.3 API Reference

RTC_bcdToDec *Changes BCD value to hexadecimal value*

Function int RTC_bcdToDec(int hex_value);

Arguments hex_value A hexadecimal value

Description Changes a BCD value to a hexadecimal value.

Example

```
 for (i = 10;i<=30;i++)
 {
   printf(”DEC of %x is %d\n”,i,RTC_bcdToDec(i));
}
```

RTC_config *Writes value to initialize RTC using configuration structure*

Function void RTC_config(RTC_Config *myConfig);

Arguments myConfig Pointer to an initialized configuration structure (containing values for all registers that are visible to the user)

Description Writes a value to initialize the RTC using the configuration structure.

Example

```
RTC_Config myConfig = {
           0x0,  /* Seconds                              */
           0x10, /* Seconds Alarm                         */
           0x18, /* Minutes                               */
           0x10, /* Minutes Alarm                         */
           0x10, /* Hour                                  */
           0x13, /* Hours Alarm                           */
           0x06, /* Day of the week and day alarm         */
           0x11, /* Day of the month                      */
           0x05, /* Month                                 */
           0x01, /* Year                                  */
           0x10, /* Peridodic Interrupt Selection register */
           0x02, /* Interrupt Enable register             */
      };
RTC_config(&myConfig);
```

RTC_decToBcd *Changes decimal value to BCD value*

Function int RTC_decToBcd(int dec_value);

Arguments dec_value A decimal value

Description Changes a decimal value to a BCD value, which is what RTC needs.

Example

```
for (i = 10;i<=30;i++)
 {
   printf("BCD of %d is %x\n",i,RTC_decToBcd(i));
}
```

RTC_eventDisable *Disables interrupt event specified by ierMask*

Function void RTC_eventDisable(Uint16 isrMask);

Arguments isrMask Can be one of the following:

RTC_EVT_PERIODIC // Periodic Interrupt
RTC_EVT_ALARM // Alarm Interrupt
RTC_EVT_UPDATE // Update Ended Interrupt

Description It disables the interrupt specified by the ierMask.

Example

```
RTC_eventDisable(RTC_EVT_UPDATE);
```

RTC_eventEnable *Enables RTC interrupt event specified by isrMask*

Function void RTC_eventEnable(Uint16 isrMask);

Arguments isrMask Can be one of the following:

RTC_EVT_PERIODIC // Periodic Interrupt
RTC_EVT_ALARM // Alarm Interrupt
RTC_EVT_UPDATE // Update Ended Interrupt

Description It enables the RTC interrupt specified by the isrMask.

Example

```
RTC_eventEnable(RTC_EVT_PERIODIC);
```

RTC_getConfig *Reads RTC configuration structure*

Function void RTC_getConfig(RTC_Config *myConfig);

Arguments myConfig Pointer to an initialized configuration structure (including all registers that are visible to the user)

Description Reads the RTC register values into the RTC configuration register structure.

Example

```
RTC_Config myConfig;

RTC_getConfig(&myConfig);
```

RTC_getDate — *Reads current date from RTC registers*

Function void RTC_getDate(RTC_Date *myDate);

Arguments myDate — Pointer to an initialized configuration structure that contains values for year, month, day of the month (date), and day of the week.

Description Reads the current date from the RTC registers. Only the 24-hour format is supported. The values of the structure are read in BCD format.

Example

```
RTC_Date getDate;

RTC_getDate(&getDate);
```

RTC_getEventId — *Obtains IRQ module event ID for RTC*

Function int RTC_getEventID()

Arguments None

Description Obtains IRQ module event ID for RTC

Example

```
int id;
id = RTC_getEventId();
```

RTC_getTime — *Reads current time from RTC registers, in 24-hour format*

Function void RTC_getTime(RTC_Time *myTime);

Arguments myTime — Pointer to an initialized configuration structure that contains values for second, minute and hour

Description Reads the current time from the RTC registers, in 24-hour format. Only the 24-hour format is supported. The values of the structure are obtained in BCD format.

Example

```
RTC_Time getTime;

RTC_getTime(&getTime);
```

RTC_reset — *Reset RTC registers to their default values*

Function void RTC_reset();

Arguments None

Description Resets RTC registers to their default values. This function is provided due to the RTC having a separate power supply and will remain powered even if the DSP is turned off.

Example void RTC_reset();

RTC_setAlarm — *Sets alarm at specific time*

Function void RTC_setAlarm(RTC_Alarm *myAlarm);

Arguments myAlarm Pointer to an initialized configuration structure that contains the hour, minute, second, and day of the week for the alarm to occur.

Description Set alarm at a specific time: sec, min, hour, day of week. Only the 24-hour format is supported. The values of the structure must be entered in BCD format.

Example 1

```
RTC_Alarm myAlarm = {
      0x12,           /* alHour , in 24-hour format */
      0x03,           /* alMinutes */
      0x03,           /* alSeconds */
      0x05,           /* alDayw */
};

RTC_setAlarm(&myAlarm);
/*This sets the alarm at 12:03:03am, */
/* every week, on Thursday           */
```

Example 2

```
RTC_Alarm myPeriodicAlarm = {
      0x1,            /* alHour , in 24-hour format */
      DONTCARE,       /* alMinutes */
      0x0,            /* alSeconds */
      0x2,            /* alDayw */
};

RTC_setAlarm(&myAlarm);
/* This sets the alarm every minute, at */
/* 01:**:00, on Monday of every week    */
```

RTC_setCallback *Associates a function to an RTC interrupt*

Function void RTC_setCallback(RTC_IsrAddr *isrAddr);

Arguments

isrAddr	A structure containing pointers to the 3 functions that will be executed when the corresponding interrupt is enabled. The functions should not be declared with the *interrupt* function keyword.

Description RTC_setCallback associates a function to each of the RTC interrupt events (periodic interrupt, alarm interrupt, or update ended interrupt):

Example

```
void myPeriodicIsr();
void myAlarmIsr();
void myUpdateIsr();
RTC_IsrAddr addr = {
myPeriodicIsr,
void myAlarmIsr,
void myUpdateIsr
};

RTC_setCallback(&addr);
```

RTC_setDate *Sets RTC calendar date*

Function void RTC_setDate(RTC_Date *myDate);

Arguments

myDate	Pointer to an initialized configuration structure that contains values for year, month, day of the month (date), and day of the week

Description Sets the RTC calendar. Only the 24-hour format is supported. The values of the structure must be entered in BCD format.

Example

```
RTC_Date myDate = {
      0x01,    /* Year 2001 */
      0x05,    /* Month May*/
      0x10,    /* Day of month */
      0x05     /* Day of week Thursday */
};

RTC_setDate(&myDate);
```

RTC_setPeriodicInterval *Sets periodic interrupt rate*

Function void RTC_setPeriodicInterval(Uint16 interval);

Arguments interval Symbolic value for periodic interrupt rate. An interval can be one of the following values:

RTC_RATE_NONE
RTC_RATE_122us
RTC_RATE_244us
RTC_RATE_488us
RTC_RATE_976us
RTC_RATE_1_95ms
RTC_RATE_3_9ms
RTC_RATE_7_8125ms
RTC_RATE_15_625ms
RTC_RATE_31_25ms
RTC_RATE_62_5ms
RTC_RATE_125ms
RTC_RATE_250ms
RTC_RATE_500ms
RTC_RATE_1min

Description Sets the periodic interrupt rate.

Example

```
RTC_setPeriodicInterval(RTC_RATE_122us);
```

RTC_setTime *Sets time registers, in 24-hour format*

Function void RTC_setTime(RTC_Time *myTime);

Arguments myTime Pointer to an initialized configuration structure that contains values for second, minute and hour

Description Sets the time registers. Only the 24-hour format is supported. The values of the structure must be entered in BCD format.

Example

```
RTC_Time myTime = {
      0x13,   /* Hour in 24-hour format */
      0x58,   /* Minutes */
      0x30    /* Seconds */
};

RTC_setTime(&myTime);
```

This example sets the RTC time to 13:58:30 (24-hour format) and is equivalent to 1:58:30 PM (12-hour format).

RTC_start — *Instructs the RTC to begin running*

Function void RTC_start();

Arguments None

Description Instructs the RTC to begin running and keep the time by setting the SET bit in the RTCINTEN register to 0.

Example RTC_start();

RTC_stop — *Stops the RTC*

Function void RTC_stop();

Arguments None

Description Instructs the RTC to stop running by setting the SET bit in the RTCINTEN register to 0.

Example RTC_stop();

17.4 Macros

The following are macros available for use with the RTC module.

RTC_ADDR *Reads register address*

Macro Uint16 RTC_ADDR(REG)

Description Reads a register address

Example

```
Uint16 x;

x = RTC_ADDR(RTCSEC);
```

RTC_FGET *Reads RTC register field values*

Macro Uint16 RTC_FGET(REG, FIELD)

Description Reads RTC register field values. This is applicable only to registers with more than one field.

Example

```
Uint16 x;

x = RTC_FGET(RTCDAYW, DAEN);
```

RTC_FSET *Writes RTC register field values*

Macro Void RTC_FSET(REG, FIELD, Uint16 fieldval)

Description Writes RTC register field values. This is applicable only to registers with more than one field.

Example

```
Uint16 x = 1;

RTC_FSET(RTCDAYW, DAEN, x);
```

RTC_REG_FMK *Creates value of RTC register fields*

Macro Uint16 RTC_REG_FMK(FIELD, Uint 16 fieldval)

Description Creates value of RTC register fields (only for registers with more than one field).

Example

```
Uint16 x, v = 0x09;

x = RTC_FMK(RTCDAYW, DAY, v);
```

RTC_REG_RMK *Creates value of RTC registers*

Macro Uint16 RTC_REG_RMK(Uint16 fieldval_n, 0, Uint16fieldval_0)

Arguments

REG	Register (RTCxxxx)
FIELD	Register field name. For REG_FSET, REG_FGET and REG_FMK, FIELD must be a writeable field
regval	Value to write in the register REG
fieldval	Value to write in the field FIELD

Description Creates value of RTC registers (only for registers with more than one field).

Example

```
Uint16 x, field1, field2, field3;

x = RTC_RTDAYW_RMK(field1, field2, field3);
```

RTC_RGET *Reads RTC register values*

Macro Uint16 RTC_RGET(REG)

Description Reads RTC register values

Example

```
Uint16 x;

x = RTC_RGET(RTCSEC);
```

RTC_RSET *Writes RTC register values*

Macro Void RTC_RSET(REG, Uint16 regval)

Description Writes RTC register values

Example

```
Uint16 x = 0x15;

RTC_RSET(RTCSEC, x);
```

Chapter 18

Timer Module

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

Topic **Page**

18.1 Overview

Table 18-1 lists the configuration structure used to set the TIMER module.

Table 18-2 lists the functions available for the TIMER module.

Table 18-3 lists registers for the TIMER module.

Section 18.4 inlcudes descriptions for available TIMER macros.

Table 18-1. TIMER Configuration Structure

Syntax	Description	See page ...
TIMER_Config	TIMER configuration structure used to setup the TIMER_config() function	18-3

Table 18-2. TIMER Functions

Syntax	Description	See page ...
TIMER_close()	Closes the TIMER and its corresponding handler	18-4
TIMER_config()	Sets up TIMER using configuration structure (TIMER_Config)	18-4
TIMER_getConfig()	Reads the TIMER configuration	18-5
TIMER_getEventId()	Obtains IRQ event ID for TIMER device	18-5
TIMER_open()	Opens the TIMER and assigns a handler to it	18-6
TIMER_reset()	Resets the TIMER registers with default values	18-7
TIMER_start()	Starts the TIMER device running	18-7
TIMER_stop()	Stops the TIMER device running	18-7
TIMER_tintoutCfg()	Sets up the TIMER Polarity pin along with settings for the FUNC, PWID, CP fields in the TCR register	18-8

Table 18-3. Registers

Register	Field
TCR	IDLEEN, (R)INTEXT, (R)ERRTIM, FUNC, TLB, SOFT, FREE, PWID, ARB, TSS, CP, POLAR, DATOUT
PRD	PRD
TIM	TIM
PRSC	PSC, TDDR

Note: R = Read Only; W = Write; By default, most fields are Read/Write

18.2 Configuration Structures

The following is the configuration structure used to set up the TIMER.

TIMER_Config — *TIMER configuration structure*

Structure TIMER_Config

Members

Uint16 tcr Timer Control Register

Uint16 prd Period Register

Uint16 prsc Timer Pre-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 Config1 = {
   0x0010, /* tcr */
   0xFFFF, /* prd */
   0xF0F0, /* prsc */
};
```

18.3 Functions

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

TIMER_close — *Closes a previously opened TIMER device*

Function

```
void TIMER_close
    TIMER_Handle hTimer
 );
```

Arguments

hTimer	Device Handle (see TIMER_open).

Return Value

TIMER_Handle	Device handler

Description

Closes a previously opened timer device. The timer event is disabled and cleared. The timer registers are set to their default values.

Example

```
TIMER_close(hTimer);
```

TIMER_config — *Writes value to TIMER using configuration structure*

Function

```
void TIMER_config(
        TIMER_Handle hTimer,
        TIMER_Config *Config
);
```

Arguments

Config	Pointer to an initialized configuration structure
hTimer	Device Handle, see TIMER_open

Return Value

none

Description

The values of the configuration structure are written to the timer registers (see also TIMER_Config).

Example

```
TIMER_Config MyConfig = {
   0x0010, /* tcr */
   0xFFFF, /* prd */
   0xF0F0  /* prsc */
};
   TIMER_config(hTimer,&MyConfig);
```

TIMER_getConfig — *Reads the TIMER configuration*

Function

void TIMER_getConfig(
    TIMER_Handle hTimer,
    TIMER_Config *Config
);

Arguments

Config	Pointer to an initialized TIMER configuration structure
hTimer	Timer Device Handle

Return Value None

Description Reads the TIMER configuration into the configuration structure. See also TIMER_Config.

Example

```
TIMER_Config MyConfig;
TIMER_getConfig(hTimer,&MyConfig);
```

TIMER_getEventId — *Obtains IRQ event ID for TIMER device*

Function

Uint16 TIMER_getEventId(
    TIMER_Handle hTimer
);

Arguments hTimer   Device handle (see TIMER_open).

Return Value Event ID   IRQ Event ID for the timer device

Description Obtains the IRQ event ID for the timer device (see Chapter 10, *IRQ Module*).

Example

```
Uint16 TimerEventId;
TimerEventId = TIMER_getEventId(hTimer);
IRQ_enable(TimerEventId);
```

TIMER_open

Opens TIMER for TIMER calls

Function

```
TIMER_Handle TIMER_open(
    int devnum,
    Uint16 flags
);
```

Arguments

devnum	Timer Device Number: TIMER_DEV0, TIMER_DEV1, TIMER_DEV_ANY
flags	Event Flag Number: Logical open or TIMER_OPEN_RESET

Return Value

TIMER_Handle	Device handler

Description

Before a TIMER device can be used, it must first be opened by this function. Once opened, it cannot be opened again until closed, see TIMER_close. The return value is a unique device handle that is used in subsequent TIMER calls. If the function fails, an INV (-1) value is returned. If the TIMER_OPEN_RESET is specified, then the power on defaults are set and any interrupts are disabled and cleared.

Example

```
TIMER_Handle hTimer;
...
hTimer = TIMER_open(TIMER_DEV0,0);
```

TIMER_reset — *Resets TIMER*

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_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);
```

TIMER_tintoutCfg *Configures TINT/TOUT pin*

Function

```
void TIMER_tintoutCfg(
    TIMER_Handle hTimer,
    Uint16 idleen,
    Uint16 func,
    Uint16 pwid,
    Uint16 cp,
    Uint16 polar
 );
```

Arguments

hTimer	Device handle (see TIMER_open).
idleen	Timer idle mode
func	Function of the TIN/TOUT pin and the source of the timer module.
pwid	TIN/TOUT pulse width
cp	Clock or pulse mode
polar	Polarity of the TIN/TOUT pin

Return Value none

Description Configures the TIN/TOUT pin of the device using the TCR register

Example

```
Timer_tintoutCfg(hTimer,
1, /*idleen*/
1, /*funct*/
0, /*pwid*/
0, /*cp*/
0  /*polar*/ );
```

18.4 Macros

The CSL offers a collection of macros to gain individual access to the TIMER peripheral registers and fields.

Table 18-4 lists of macros available for the TIMER module using TIMER port number and Table 18-5 lists the macros for the TIMER module using handle. To use them, include “csl_timer.h.”

Table 18-3 lists DMA registers and fields.

Table 18-4. 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 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*, *fieldval*)

(d) Macros to read a register address

Macro	Syntax
TIMER_ADDR()	Uint16 TIMER_ADDR(*REG#*)

Notes:
1) *REG* indicates the registers: TCR, PRD, TIM, PRSC
2) REG# indicates, if applicable, a register name with the channel number (example: TCR0)
3) *FIELD* indicates the register field name as specified in the *C55x DSP Peripherals Reference Guide*.
   For *REG*_FSET and *REG*_FMK, *FIELD* must be a writable field.
   For *REG*_FGET, the field must be a readable field.
4) *regval* indicates the value to write in the register (*REG*).
5) *fieldval* indicates the value to write in the field (*FIELD*).

Table 18-5. TIMER CSL Macros Using Handle

(a) Macros to read/write TIMER register values

Macro	Syntax
TIMER_RGETH()	Uint16 TIMER_RGETH(TIMER_Handle hTimer, *REG*)
TIMER_RSETH()	Void TIMER_RSETH(TIMER_Handle hTimer, *REG*, Uint16 *regval*)

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

Macro	Syntax
TIMER_FGETH()	Uint16 TIMER_FGETH(TIMER_Handle hTimer, *REG*, *FIELD*)
TIMER_FSETH()	Void TIMER_FSETH(TIMER_Handle hTimer, *REG*, *FIELD*, *Uint16 fieldval*)

(c) Macros to read a register address

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

Notes:
1) *REG* indicates the registers: TCR, PRD, TIM, and PRSC
2) *FIELD* indicates the register field name as specified inthe *C55x DSP Peripherals Reference Guide*.
   For *REG*_FSETH, *FIELD* must be a writable field.
   For *REG*_FGETH, 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*).

Chapter 19

UART Module

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

Topic **Page**

19.1 Overview

The Universal Asynchronous Receiver/Transmitter (UART) controller is the key component of the serial communications subsystem of a computer. Asynchronous transmission allows data to be transmitted without a clock signal to the receiver. Instead, the sender and receiver must agree on timing parameters in advance. Special bits are added to each word that is used to synchronize the sending and receiving units.

The configuration of UART can be performed by using one of the following methods:

1) Register-based configuration

   A register-based configuration can be performed by calling either UART_config() or any of the SET register field macros.

2) Parameter-based configuration (Recommended)

   A parameter-based configuration can be performed by calling UART_setup(). Compared to the register-based approach, this method provides a higher level of abstraction.

Table 19-1 lists the configuration structures and functions used with the UART module.

Table 19-1. UART APIs

Structure	Type	Purpose	See page ...
UART_Config	S	UART configuration structure used to setup the UART	19-5
UART_config	F	Sets up the UART using the configuration structure	19-8
UART_eventDisable	F	Disable UART interrupts	19-8
UART_eventEnable	F	Enable UART interrupts	19-9
UART_fgetc	F	Read a character from UART by polling	19-10
UART_fgets	F	This routine reads a string from the uart	19-11
UART_fputc	F	Write a character from UART by polling	19-11
UART_fputs	F	This routine writes a string from the uart	19-11
UART_getConfig	F	Reads the UART configuration	19-11
UART_read	F	Read a buffer of data from UART by polling	19-12
UART_setCallback	F	Plugs UART interrupt routines into UART dispatcher table	19-12

Note: F = Function; S = Structure

Table 19-1. UART APIs (Continued)

Structure	Type	Purpose	See page ...
UART_Setup	S	UART configuration structure used to setup the UART	19-5
UART_setup	F	Sets up the UART using the register values passed into the code	19-13
UART_write	F	Write a buffer of data to UART by polling	19-13

Note: F = Function; S = Structure

19.2 Configuration Structures

UART_Config

Configuration Structure for UART

Members

Uint16	dll	Divisor Latch Register (low 8 bits)
Uint16	dlm	Divisor Latch Register (high 8 bits)
Uint16	lcr	Line Control Register
Uint16	fcr	FIFO Control Register
Uint16	mcr	Modem Control Register

Description

UART configuration structure. This structure is created and initialized, and then passed to the UART_Config() function.

UART_Setup

Structure used to initialize the UART

Members

Uint16 clkInput — UART input clock frequency. Valid symbolic values are:

UART_CLK_INPUT_20 // Input clock = 20MHz
UART_CLK_INPUT_40 // Input clock = 40MHz
UART_CLK_INPUT_60 // Input clock = 60MHz
UART_CLK_INPUT_80 // Input clock = 80MHz
UART_CLK_INPUT_100 // Input clock = 100MHz
UART_CLK_INPUT_120 // Input clock = 120MHz
UART_CLK_INPUT_140 // Input clock = 140MHz

Uint16 baud — Baud Rate (Range: 150 - 115200). Valid symbolic values are:

UART_BAUD_150
UART_BAUD_300
UART_BAUD_600
UART_BAUD_1200
UART_BAUD_1800
UART_BAUD_2000
UART_BAUD_2400
UART_BAUD_3600

	UART_BAUD_4800
	UART_BAUD_7200
	UART_BAUD_9600
	UART_BAUD_14400
	UART_BAUD_19200
	UART_BAUD_38400
	UART_BAUD_57600
	UART_BAUD_115200
Uint16 wordLength	bits per word (Range: 5,6,7,8).
	Valid symbolic values are:
	UART_WORD5 5 bits per word
	UART_WORD6 6 bits per word
	UART_WORD7 7 bits per word
	UART_WORD8 8 bits per word
Uint16 stopBits	stop bits in a word (1, 1.5, and 2)
	Valid symbolic values are:
	UART_STOP1 1 stop bit
	UART_STOP1_PLUS_HALF
	1 and 1/2 stop bits
	UART_STOP2 2 stop bits
Uint16 parity	parity setups
	Valid symbolic values are:
	UART_DISABLE_PARITY
	UART_ODD_PARITY odd parity
	UART_EVEN_PARITY even parity
	UART_MARK_PARITY mark parity
	(the parity bit is always ‘1’)
	UART_SPACE_PARITY space parity
	(the parity bit is always ‘0’)

Uint16 fifoControl FIFO Control

Valid symbolic values are:

UART_FIFO_DISABLE //Non FIFO mode
UART_FIFO_DMA0_TRIG01 //DMA mode 0 and Trigger level 1
UART_FIFO_DMA0_TRIG04 //DMA mode 0 and Trigger level 4
UART_FIFO_DMA0_TRIG08 //DMA mode 0 and Trigger level 8
UART_FIFO_DMA0_TRIG14 //DMA mode 0 and Trigger level 14

UART_FIFO_DMA1_TRIG01 //DMA mode 1 and Trigger level 1
UART_FIFO_DMA1_TRIG04 //DMA mode 1 and Trigger level 4
UART_FIFO_DMA1_TRIG08 //DMA mode 1 and Trigger level 8
UART_FIFO_DMA1_TRIG14 //DMA mode 1 and Trigger level 14

Uint16 loopbackEnable loopback Enable Valid Symbolic values are:

UART_NO_LOOPBACK
UART_LOOPBACK

Description Structure used to init the UART. After created and initialized, it is passed to the UART_setup() function.

19.3 Functions

19.3.1 CSL Primary Functions

UART_config *Initializes the UART using the configuration structure*

Function void UART_config (UART_Config *Config);

Arguments Configure pointer to an initialized configuration structure (containing values for all registers that are visible to the user)

Description Writes a value to initialize the UART using the configuration structure.

Example

```
UART_Config Config = {
   0x00, /* DLL */
   0x06, /* DLM – baud rate 150 */
   0x18, /* LCR – even parity, 1 stop bit, 5
            bits word length */
   0x00, /* Disable FIFO */
   0x00  /* No Loop Back */
};
UART_config(&Config);
```

UART_eventDisable *Disables UART interrupts*

Function void UART_eventDisable(Uint16 ierMask);

Arguments ierMask can be one or a combination of the following:

```
UART_RINT    0x01     // Enable rx data available
                         interrupt
UART_TINT    0x02     // Enable tx hold register
                         empty interrupt
UART_LSINT   0x04     // Enable rx line status
                         interrupt
UART_MSINT   0x08     // Enable modem status
                         interrupt
UART_ALLINT  0x0f     // Enable all interrupts
```

Description It disables the interrupt specified by the ierMask.

Example `UART_eventDisable(UART_TINT);`

UART_eventEnable *Enables a UART interrupt*

Function void UART_eventEnable (Uint16 isrMask);

Arguments isrMask can be one or a combination of the following:

```
UART_RINT    0x01  // Enable rx data available interrupt
UART_TINT    0x02  // Enable tx hold register
                      empty interrupt
UART_LSINT   0x04  // Enable rx line status interrupt
UART_MSINT   0x08  // Enable modem status interrupt
UART_ALLINT  0x0f  // Enable all interrupts
```

Description It enables the UART interrupt specified by the isrMask.

Example `UART_eventEnable(UART_RINT|UART_TINT);`

UART_fgetc

Reads UART characters

Function

CSLBool UART_fgetc(int *c, Uint32 timeout);

Arguments

c	Character read from UART
timeout	Time out for data ready. If it is setup as 0, means there will be no time out count. The function will block forever until DR bit is set.

Description

Read a character from UART by polling.

Example

Int retChar;
CSLBool returnFlag

```
returnFlag = UART_fgetc(&retChar,0);
```

UART_fgets

Reads UART strings

Function

CSLBool UART_fgets(char* pBuf, int bufSize, Uint32 timeout);

Arguments

pBuf	Pointer to a buffer
bufSize	Length of the buffer
timeout	Time out for data ready. If it is setup as 0, means there will be no time out count. The function will block forever until DR bit is set.

Description

This routine reads a string from the uart. The string will be read upto a newline or until the buffer is filled. The string is always NULL terminated and does not have any newline character removed.

Example

char readBuf[10];
CSLBool returnFlag

```
returnFlag = UART_fgets(&readBuf[0], 10, 0);
```

UART_fputc

Writes characters to the UART

Function

CSLBool UART_fputc(const int c, Uint32 timeout);

Arguments

c	The character, as an int, to be sent to the uart.
timeout	Time out for data ready. If it is setup as 0, means there will be no time out count. The function will block forever if THRE bit is not set.

Description

This routine writes a character out through UART.

Example

Example const int putchar = ‘A’;
CSLBool returnFlag;

```
ReturnFlag = UART_fputc(putchar, 0);
```

UART_fputs

Writes strings to the UART

Function

CSLBool UART_fputs(const char* pBuf, Uint32 timeout);

Arguments

pBuf	Pointer to a buffer
timeout	Time out for data ready. If it is setup as 0, means there will be no time out count. The function will block forever if THRE bit is not set.

Description

This routine writes a string to the uart. The NULL terminator is not written and a newline is not added to the output.

Example

```
UART_fputs(”\n\rthis is a test!\n\r”);
```

UART_getConfig

Reads the UART Configuration Structure

Function

void UART_getConfig (UART_Config *Config);

Arguments

Config Pointer to an initialized configuration structure (including all registers that are visible to the user)

Description

Reads the UART configuration structure.

Example

```
UART_Config Config;

UART_getConfig(&Config);
```

UART_read

Reads received data

Function

CSLBool UART_read(char *pBuf, Uint16 length, Uint32 timeout);

Arguments

pbuf	Pointer to a buffer
length	Length of data to be received
timeout	Time out for data ready. If it is setup as 0, means there will be no time out count. The function will block forever until DR bit is set.

Description

Receive and put the received data to the buffer pointed by pbuf.

Example

```
Uint16 length = 10;
char pbuf[length];
CSLBool returnFlag;

ReturnFlag = UART_read(&pbuf[0],length, 0);
```

UART_setCallback

Associates a function to the UART dispatch table

Function

void UART_setCallback(UART_IsrAddr *isrAddr);

Arguments

isrAddr is a structure containing pointers to the 5 functions that will be executed when the corresponding events is enabled.

Description

It associates each function specified in the isrAddr structure to the UART dispatch table.

Example

```
UART_IsrAddr MyIsrAddr= {
   NULL,          // Receiver line status
   UartRxIsr,     // received data available
   UartTxIsr,     // transmiter holding register empty
   NULL           // character time-out indication
};
UART_setCallback(&MyIsrAddr);
```

UART_setup

Sets the UART based on the UART_Setup configuration structure

Function

void UART_setup (UART_Setup *Params);

Arguments

Params Pointer to an initialized configuration structure that contains values for UART setup.

Description

Sets UART based on UART_Setup structure.

Example

```
UART_Setup Params = {
   UART_CLK_INPUT_60,     /* input clock freq   */
   UART_BAUD_115200,      /* baud rate          */
   UART_WORD8,            /* word length        */
   UART_STOP1,            /* stop bits          */
   UART_DISABLE_PARITY,   /* parity             */
   UART_FIFO_DISABLE,     /* FIFO control       */
   UART_NO_LOOPBACK,      /* Loop Back enable/disable */
};
UART_setup(&Params);
```

UART_write

Transmits buffers of data by polling

Function

CSLBool UART_write(char *pBuf, Uint16 length, Uint32 timeout);

Arguments

pbuf	Pointer to a data buffer
Length	Length of the data buffer
timeout	Time out for data ready. If it is setup as 0, means there will be no time out count. The function will block forever if THRE bit is not set.

Description

Transmit a buffer of data by polling.

Example

```
Uint16 length = 4;
char pbuf[4] = {0x74, 0x65, 0x73, 0x74};
CSLBool returnFlag;
ReturnFlag = UART_write(&pbuf[0],length,0);
```

19.4 Macros

The following macros are used with the UART chip support library.

19.4.1 General Macros

Table 19-2. UART CSL Macros

Macro	Syntax
(a) Macros to read/write UART register values	
UART_RGET()	Uint16 UART_RGET(*REG*)
UART_RSET()	void UART_RSET(*REG*, Uint16 *regval*)
(b) Macros to read/write UART register field values (Applicable only to registers with more than one field)	
UART_FGET()	Uint16 UART_FGET(*REG*, *FIELD*)
UART_FSET()	void UART_FSET(*REG*, *FIELD*, Uint16 *fieldval*)
(c) Macros to create value to write to UART registers and fields (Applicable only to registers with more than one field)	
UART_*REG*_RMK()	Uint16 UART_*REG*_RMK(*fieldval_n*,...*fieldval_0*) Note: *Start with field values with most significant field positions: field_n: MSB field field_0: LSB field * only writable fields allowed
UART_FMK()	Uint16 UART_FMK(*REG*, *FIELD*, *fieldva*l)

Notes:
1) *REG* indicates the registers: URIER, URIIR, URBRB, URTHR, URFCR, URLCR, URMCR, URLSR, URMSR, URDLL or URDLM.
2) *FIELD* indicates the register field name.
3) - or *REG*_FSET and *REG*__FMK, *FIELD* must be a writable field.
4) - For *REG*_FGET, the field must be a readable field.
5) *regval* indicates the value to write in the register (*REG*)
6) *fieldval* indicates the value to write in the field *(FIELD*)

Table 19-2. UART CSL Macros (Continued)

Macro	Syntax
(d) Macros to read a register address	
UART_ADDR()	Uint16 UART_ADDR(*REG*)

Notes: 1) *REG* indicates the registers: URIER, URIIR, URBRB, URTHR, URFCR, URLCR, URMCR, URLSR, URMSR, URDLL or URDLM.

2) *FIELD* indicates the register field name.

3) - or *REG*_FSET and *REG*__FMK, *FIELD* must be a writable field.

4) - For *REG*_FGET, the field must be a readable field.

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

6) *fieldval* indicates the value to write in the field *(FIELD*)

19.4.2 UART Control Signal Macros

All the UART control signals are mapped through HPIGPIO pins. They are configurable through GPIOCR and GPIOSR registers. Since C54x DSP are commonly used as DCE (Data Communication Equipment), these signals are configured as following:

HD0 - DTR – Input

HD1 - RTS – Input

HD2 - CTS – Output

HD3 - DSR – Output

HD4 - DCD – Output

HD5 - RI – Output

UART_ctsOff

Sets a CTS signal to OFF

Macro UART_ctsOff

Arguments None

Description Set CTS signal off.

Example

```
UART_ctsOff;
```

UART_ctsOn

Sets a CTS signal to ON

Macro UART_ctsOn

Arguments None

Description Set CTS signal on.

Example

```
UART_ctsOn;
```

UART_flowCtrlInit

Initializes the HPIGPIO registers for flow control

Macro UART_flowCtrlInit

Arguments None

Description Initialize HPIGPIO registers for flow control.

Example

```
UART_flowCtrlInit;
```

UART_isRts

Verifies that RTS is ON

Macro UART_isRts

Arguments None

Description Check if RTS is on. Return RTS value.

Example CSLBool rtsSignal;

```
rtsSignal = UART_isRts;
```

UART_dtcOff

Sets a DTC signal to OFF

Macro UART_dtcOff

Arguments None

Description Set DTC signal off.

Example

```
UART_dtcOff;
```

UART_dtcOn

Sets a DTC signal to ON

Macro UART_dtcOn

Arguments None

Description Set DTC signal on.

Example

```
UART_dtcOn;
```

UART_riOff

Sets an RI signal to OFF

Macro UART_riOff

Arguments None

Description Set RI signal off.

Example

```
UART_riOff;
```

UART_riOn

Sets an RI signal to ON

Macro UART_riOn

Arguments None

Description Set RI signal on.

Example

```
UART_riOn;
```

UART_dsrOff

Sets a DSR signal to OFF

Macro UART_dsrOff

Arguments None

Description Set DSR signal off.

Example

```
UART_dsrOff;
```

UART_dsrOn

Sets a DSR signal to ON

Macro UART_dsrOn

Arguments None

Description Set DSR signal on.

Example

```
UART_dsrOn;
```

UART_isDtr

Verifies that DTR is ON

Macro UART_isDtr

Arguments Nobe

Description Check if DTR is on. Return DTR value.

Example CSLBool dtrSignal;

```
dtrSignal = UART_isDtr;
```

Chapter 20

WDTIM Module

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

Topic **Page**

20.1 Overview

Table 20-1 lists the configuration structures and functions used with the WDTIM module.

Table 20-1. WDTIM Structure and APIs

Structure	Desctiption	See page...
WDTIM_Config	Structure used to configure a WDTIM Device	20-3

Syntax	Desctiption	See page...
WDTIM_config	Configures WDTIM using configuration structure	20-4
WDTIM_service	Executes the watchdog service sequence	20-9
The following functions are supported by C5509/C5509A only		
WDTIM_getConfig	Reads the WDTIM configuration structure	20-5
WDTIM_start	Starts the WDTIM device running	20-10
The following functions are supported by C5502 Only		
WDTIM_close	Closes previously opened WDTIMER device	20-4
WDTIM_getCnt	Gives the timer count values	20-5
WDTIM_getPID	Gets peripheral ID details	20-6
WDTIM_init64	Intializes the timer in 64 bit mode	20-6
WDTIM_open	Opens the WDTIM device for use	20-9
WDTIM_start	Pulls both timers out of reset before activating the watchdog timer	20-10
WDTIM_stop	Stops all the timers if running	20-12
WDTIM_wdStart	Activates the watchdog timer	20-13

20.2 Configuration Structures

The following is the configuration structure used to set up the Watchdog Timer module.

WDTIM_Config *Structure used to configure a WDTIM device*

Structure WDTIM_Config

Members For C5509/5509A only

Uint16 wdprd	Period register
Uint16 wdtcr	Control register
Uint16 wdtcr2	Secondary register

Members For C5502 only

Uint16 wdtemu	Emulation management register
Uint16 wdtgpint	GPIO interrupt control register
Uint16 wdtgpen	GPIO enable register
Uint16 wdtgpdir	GPIO direction register
Uint16 wdtgpdat	GPIO data register
Uint16 wdtprd1	Timer period register 1
Uint16 wdtprd2	Timer period register 2
Uint16 wdtprd3	Timer period register 3
Uint16 wdtprd4	Timer period register 4
Uint16 wdtctl1	Timer control register 1
Uint16 wdtctl2	Timer control register 2
Uint16 wdtgctl1	Global timer control register
Uint16 wdtwctl1	Watchdog timer control register 1
Uint16 wdtwctl2	Watchdog timer control register 2

Example This example shows how to configure a watchdog timer for C5509/5509A devices.

```
WDTIM_Config MyConfig = {

           0x1000, /* Period */

           0x0000, /* Control */

           0x1000 /* Secondary control */
};

WDTIM_config(&MyConfig);
```

20.3 Functions

The following functions are available for use with the Watchdog Timer module.

WDTIM_close — *Closes a previously opened WDTIMER device*

Function void WDTIM_close(WDTIM_Handle hWdtim)

Arguments hWdtim Device handle; see WDTIM_open

Return Value None

Description WDTIM_close closes a previously opened WDTIMER device

Example

```
WDTIM_Handle hWdtim;
...
WDTIM_close(hWdtim);
```

WDTIM_config — *Configures WDTIM using configuration structure*

Function **For 5509/5509A only**

```
void WDTIM_config(
                    WDTIM_Config *myConfig
);
```

Function **For 5502 only**

```
void  WDTIM_config(
   WDTIM_Handle      hWdtim,
   WDTIM_Config      *myConfig
);
```

Arguments **For 5509/5509A only**

myConfig Pointer to the initialized configuration structure

Arguments **For 5502 only**

hWdtim Device Handle; see WDTIM_open

myConfig Pointer to the initialized configuration structure

Return Value None

Description Configures the WDTIMER device using the configuration structure which contains members corresponding to each of the WDTIM registers. These values are directly written to the corresponding WDTIM device-registers.

Example

This is the example skeleton code for 5502 only

```
WDTIM_Handle hWdtim;
WDTIM_Config MyConfig;
...
WDTIM_config(hWdtim, &MyConfig);
```

WDTIM_getCnt

Gives the timer count values

Function

void WDTIM_getCnt(
WDTIM_Handle h,
Uint32 *hi32,
Uint32 *lo32
)

Arguments

h	Device Handle; see WDTIM_open
hi32	Pointer to obtain CNT3 and CNT4 values
lo32	Pointer to obtain CNT1 and CNT2 values

Return Value

None

Description

Gives the timer count values. hi32 will give CNT1 and CNT2 values aligned in 32 bit. lo32 will give CNT3 and CNT4 values aligned in 32 bit.

Example

```
WDTIM_Handle hWdtim;
Uint32 *hi32,*lo32;
   ...

WDTIM_getCnt(hWdtim,hi32,lo32);
```

WDTIM_getConfig

Gets the WDTIM configuration structure for a specified device

Function

void WDTIMER_getConfig(
    WDTIMER_Config *Config
);

Arguments

Config	Pointer to a WDTIM configuration structure

Return Value

None

Description

Gets the WDTIM configuration structure for a specified device.

Example

```
WDTIM_Config MyConfig;
WDTIM_getConfig(&MyConfig);
```

WDTIM_getPID *Gets peripheral ID details*

Function

```
void  WDTIM_getPID(
WDTIM_Handle hWdtim,
Uint16        *_type,
Uint16        *_class,
Uint16        *revision
)
```

Arguments

hWdtim	Device Handle; see WDTIM_open
_type	Pointer to obtain Device type
_class	Pointer to obtain device class
revision	Pointer to obtain device revision

Return Value None

Description Obtains the peripheral ID details like class ,type and revision

Example

```
WDTIM_Handle hWdtim;
Uint16 *type;
Uint16 *class;
Uint16 *rev;

...
WDTIM_getPID(hWdtim,type,class,rev);
```

WDTIM_init64 *Initializes the timer in 64-bit mode*

Function

```
void  WDTIM_init64(
WDTIM_Handle hWdtim,
Uint16        gptgctl,
Uint16        dt12ctl,
Uint32        prdHigh,
Uint32        prdLow
)
```

Arguments

hWdtim	Device Handle; see WDTIM_open
gptgctl	Global timer control(not used)
dt12ctl	timer1 control value
prdHigh	MSB of timer period value
prdLow	LSB of timer period value

Return Value None

Description This API is used to set up and intialize the timer in 64 bit mode. It allows to initialize the period and also provide arguments to setup the timer control registers.

Example

```
WDTIM_Handle hWdtim;
  ......
  WDTIM_init64(
         hWdtim,   // Device Handle; see WDTIM_open
         0x0000,   // Global timer control(not used)
         0x5F04,   // timer1 control value
         0x0000,   // MSB of timer period value
         0x0000    // LSB of timer period value
```

WDTIM_initChained32 *Initializes the timer in dual 34-bit chained mode*

Function

```
void  WDTIM_initChained32(
    WDTIM_Handle    hWdtim,
    Uint16          gctl,
    Uint16      ctl1,
    Uint32      prdHigh,
    Uint32      prdLow
)
```

Arguments

hWdtim	Device Handle; see WDTIM_open
gctl	Global timer control(not used)
ctl1	Timer1 control value
prdHigh	Higher bytes of timer period value
prdLow	Lower bytes of timer period value

Return Value None

Description This API is used to set up and intialize two 32-bit timers in chained mode. It allows to initialize the period and also provide arguments to set up the timer control registers.

Example

```
WDTIM_Handle hWdtim;
  ......
    WDTIM_initChained32(
      Handle hWdtim,
      0x0000 // Global timer control(not used)
      0x5F04 // Timer1 control value
      0x0000,// MSB of timer period value
      0x0000 // LSB of timer period value
     );
```

WDTIM_initDual32 *Initializes the timer in dual 32-bit unchained mode*

Function

```
void  WDTIM_initDual32(
   WDTIM_Handle      hWdtim,
   Uint16       dt1ctl,
   Uint16       dt2ctl,
   Uint32       dt1prd,
   Uint32       dt2prd,
   Uint16       dt2prsc
)
```

Arguments

hWdtim	Device Handle; see WDTIM_open
dt1ctl	timer1 control value
dt2ctl	timer2 control value
dt1prd	Timer1 period
dt2prd	Timer2 period
dt2prsc	Prescalar count

Return Value None

Description This API is used to set up and intialize the timer in dual 32-bit unchained mode. It allows to initialize the period for both the timers and also the prescalar counter which specify the count of the timer.It also provide arguments to setup the timer control registers.

Example

```
WDTIM_Handle hWdtim;
  ......
WDTIM_initDual32(
         hWdtim,
         0x3FE, // timer1 control value
         0x3FE, // timer2 control value
         0x005, // Timer1 period
         0x008, // Timer2 period
         0x0FF  // Prescalar count
  );
```

WDTIM_open

Opens the WDTIM device for use

Function

```
WDTIM_Handle  WDTIM_open(
   void
)
```

Arguments None

Return Value WDTIM_Handle

Description Before the WDTIM device can be used, it must be 'opened' using this function. Once opened it cannot be opened again until it is 'closed' (see WDTIM_close). The return value is a unique device handle that is used in subsequent WDTIM API calls.

Example

```
WDTIM_Handle hWdtim;
  ...
  hWdtim = WDTIM_open();
```

WDTIM_service

Executes the watchdog service sequence

Function **For 5509/5509A**

```
void WDTIM_service(
                    void
);
```

Arguments void

Return Value None

Description Executes the watchdog timer service sequence

Example

```
WDTIM_service();
```

Function **For C5502**

```
void WDTIM_service(
                    WDTIM_Handle hWdt
);
```

Arguments hWdt Device Handle; see WDTIM_open

Return Value None

Description Executes the watchdog service sequence

Example

```
WDTIM_Handle hWdtim;
...
WDTIM_service(hWdtim);
```

WDTIM_start

Starts the watchdog timer operation (5509/5509A)/ Pulls both timers out of reset (5502)

Function **For 5509/5509A only**

```
void WDTIM_start(
            void
);
```

Arguments void

Return Value None

Description Starts the watchdog timer device running.

Example

```
WDTIM_start();
```

Function **For 5502 only**

```
void WDTIM_start(
            WDTIM_Handle hWdt
);
```

Arguments hWdt Device Handle; see WDTIM_open

Return Value None

Description Starts both the timers running, i.e., timer12 and timer34 are pulled out of reset.

Example

```
WDTIM_Handle hWdtim;
...
WDTIM_start (hWdtim);
```

WDTIM_start12 *Starts the 32-bit timer1 device*

Function

void WDTIM_start12(
   WDTIM_Handle hWdtim
)

Arguments

hWdtim
   Device Handle; see WDTIM_open

Return Value

None

Description

Starts the 32-bit timer1 device

Example

```
WDTIM_Handle hWdtim;
    ....
   WDTIM_start12(hWdtim);
```

WDTIM_start34 *Starts the 32-bit timer2 device*

Function

void WDTIM_start34(
   WDTIM_Handle hWdtim
)

Arguments

hWdtim   Device Handle; see WDTIM_open

Return Value

None

Description

Starts the 32-bit timer2 device

Example

```
WDTIM_Handle hWdtim;
    ....
   WDTIM_start12(hWdtim);
```

WDTIM_stop — *Stops all the timers if running*

Function

```
void  WDTIM_stop(
   WDTIM_Handle      hWdtim
)
```

Arguments hWdtim   Device Handle; see WDTIM_open.

Return Value None

Example Stops the timer if running.

Example

```
WDTIM_Handle hWdtim;
    ....
   WDTIM_stop(hWdtim);
```

WDTIM_stop12 — *Stops the 32-bit timer1 device if running*

Function

```
void  WDTIM_stop12(
   WDTIM_Handle      hWdtim
)
```

Arguments hWdtim   Device Handle; see WDTIM_open

Return Value None

Description Stops the 32-bit timer1 device if running.

Example

```
WDTIM_Handle hWdtim;
    ....
   WDTIM_stop12(hWdtim);
```

WDTIM_stop34 — *Stops the 32-bit timer2 device if running*

Function

```
void  WDTIM_stop34(
   WDTIM_Handle      hWdtim
)
```

Arguments hWdtim   Device Handle; see WDTIM_open

Return Value None

Description Stops the 32-bit timer2 device if running.

Example

```
WDTIM_Handle hWdtim;
    ....
   WDTIM_stop34(hWdtim);
```

WDTIM_wdStart *Activates the watchdog timer*

Function

void WDTIM_wdStart(
   WDTIM_Handle hWdt
)

Arguments

Arguments hWdt
   Device Handle; see WDTIM_open

Return Value

None

Description

Activates the watchdog timer.

Example

```
WDTIM_Handle hWdtim;
    ....
   WDTIM_wdStart(hWdtim);
```

20.4 Macros

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

Table 20-2 lists the WDTIM macros available. To use them, include “csl_wdtimer.h.”

Table 3-3 lists DMA registers and fields.

Table 20-2. WDTIM CSL Macros

(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 registers: WDTCR, WDPRD, WDTCR2, or WDTIM.

2) *FIELD* indicates the register field name.

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*)

Chapter 21

GPT Module

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

Topic **Page**

21.1 Overview

This section describes the interface to the two general purpose timers (GPT0, GPT1) available in TMS320VC5501/5502 DSPs. It also lists the API functions and macros within the module, discusses how to use a GPT device, and provides a GPT API reference section.

Table 21-1 lists the configuration structure used to set the GPT module.

Table 21-2 lists the functions available for the GPT module.

Table 21-1. GPT Configuration Structure

Syntax	Description	See page ...
GPT_Config	Structure used to configure a GPT device	21-3
GPT_OPEN_RESET	GPT reset flag, used while opening the GPT device	21-3

Table 21-2. GPT Functions

Structure	Purpose	See page ...
GPT_close	Closes previously opened GPT device	21-4
GPT_config	Configure GPT using configuration structure	21-4
GPT_getCnt	Gives the timer count values	21-5
GPT_getConfig	Reads the current GPT configuration values	21-5
GPT_getEventId	Returns event ID of the opened GPT device	21-6
GPT_getPID	Gets peripheral ID details	21-6
GPT_init64	Intialize the timer in 64 bit mode	21-7
GPT_initChained32	Intialize the timer in dual 32 bit chained mode	21-8
GPT_initDual32	Intialize the timer in dual 32 bit unchained mode	21-9
GPT_open	Opens a GPT device for use	21-10
GPT_reset	Resets a GPT	21-10
GPT_start	Starts all the timers	21-11
GPT_start12	Starts the 32 bit timer1 device	21-11
GPT_start34	Starts the 32 bit timer2 device	21-11
GPT_stop	Stops the timer if running	21-12
GPT_stop12	Stops the 32 bit timer1 device if running	21-12
GPT_stop34	Stops the 32 bit timer2 device if running	21-13

21.2 Configuration Structure

The following is the configuration structure used to set up the GPT module.

GPT_Config — *Structure used to configure a GPT device*

Structure GPT_Config

Members

```
Uint16 gptemu   //Emulation management register
Uint16 gptgpint //GPIO interrupt control register
Uint16 gptgpen  //GPIO enable register
Uint16 gptgpdir //GPIO direction register
Uint16 gptgpdat //GPIO data register
Uint16 gptprd1  //Timer period register 1
Uint16 gptprd2  //Timer period register 2
Uint16 gptprd3  //Timer period register 3
Uint16 gptprd4  //Timer period register 4
Uint16 gptctl1  //Timer control register 1
Uint16 gptctl2  //Timer control register 2
Uint16 gptgctl1 //Global timer control register
```

Description This is the GPT configuration structure used to configure a GPT device. The user should create and initalize this structure before passing its address to the GPT_config function.

GPT_OPEN_RESET GPT — *Reset flag, used while opening the GPT device*

Constant GPT_OPEN_RESET

Description This flag is used while opening a GPT device.

Example See GPT_open

21.3 Functions

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

GPT_close — *Closes previously opened GPT device*

Function

```
void  GPT_close(
    GPT_Handle        hGpt
)
```

Arguments

hGpt	Device handle; see GPT_open

Return Value none

Description Closes the previously opened GPT device(see GPT_open). The following tasks are performed:

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

Example

```
GPT_Handle hGpt;
   .....
   GPT_close(hGpt);
```

GPT_config — *Configure GPT using configuration structure*

Function

```
void  GPT_config(
    GPT_Handle        hGpt,
    GPT_Config        *myConfig
)
```

Arguments

hGpt	Device Handle; see GPT_open
myConfig	Pointer to the initialized configuration structure

Return Value none

Description Configures the GPT device using the configuration structure which contains members corresponding to each of the GPT registers. These values are directly written to the corresponding GPT device-registers.

Example

```
GPT_Handle hGpt;
  GPT_Config MyConfig
  ...
  GPT_config(hGpt, &MyConfig);
```

GPT_getCnt

Gives the timer count values

Function

```
void  GPT_getCnt(
    GPT_Handle       hGpt,
    Uint32           *tim34,
    Uint32           *tim12
)
```

Arguments

hGpt	Device Handle; see GPT_open
tim34	Pointer to obtain CNT3 and CNT4 values
tim12	Pointer to obtain CNT1 and CNT2 values

Return Value none

Description Gives the timer count values. tim12 will give CNT1 and CNT2 values aligned in 32-bit format. tim34 will give CNT3 and CNT4 values aligned in 32-bit format.

Example

```
GPT_Handle hGpt;
   Uint32 *tim12,*tim34;
   ...

   GPT_getCnt(hGpt,tim34,tim12);
```

GPT_getConfig

Reads the current GPT configuration values

Function

```
void  GPT_getConfig(
    GPT_Handle       hGpt,
    GPT_Config       *myConfig
)
```

Arguments

hGpt	Device Handle; see GPT_open
myConfig	Pointer to the configuration structure

Return Value none

Description Gives the current GPT configuration values.

Example

```
GPT_Handle hGpt;
  GPT_Config gptCfg;
  .....
  GPT_getConfig(hGpt, &gptCfg);
```

GPT_getEventId *Returns event ID of the opened GPT device*

Function

Uint16 GPT_getEventId(
GPT_Handle hgpt
)

Arguments

hGpt Handle of GPT device opened

Return Value

Uint16 Event Id value

Description

Before using IRQ APIs to setup/enable/disable ISR for device, this function must be used. The return value of this function can later be used as an input to IRQ APIs.

Example

```
GPT_Handle hGpt;
Uint16 gptEvt_Id;
...
gptEvt_Id = GPT_getEventId(hGpt);
IRQ_clear(gptEvt_Id);
IRQ_plus (gptEvt_Id, & gptIsr);
IRQ_enable (gptEvt_Id);
```

GPT_getPID *Gets peripheral ID details*

Function

```
void  GPT_getPID(
     GPT_Handle       hGpt,
     Uint16           *_type,
     Uint16           *_class,
     Uint16           *revision
)
```

Arguments

hGpt	Device Handle; see GPT_open
_type	Pointer to obtain device type

_class	Pointer to obtain device class
revision	Pointer to obtain device revision

Return Value none

Description Obtains the peripheral ID details like class, type, and revision.

Example

```
GPT_Handle hGpt;
     Uint16 *type;
     Uint16 *class;
     Uint16 *rev;
     ...

     GPT_getPID(hGpt,type,class,rev);
```

GPT_init64 *Intialize the timer in 64-bit mode*

Function

```
void  GPT_init64(
    GPT_Handle      hGpt,
    Uint16          gptgctl,
    Uint16          dt12ctl,
    Uint32          prdHigh,
    Uint32          prdLow
)
```

Arguments

hGpt	Device Handle; see GPT_open
gptgctl	Global timer control (not used)
dt12ctl	timer1 control value
prdHigh	MSB of timer period value
prdLow	LSB of timer period value

Return Value none

Description This API is used to set up and intialize the timer in 64-bit mode. It allows to initialize the period and also provide arguments to setup the timer control registers.

Example

```
GPT_Handle hGpt;
  ......
  GPT_init64(
         hGpt,  // Device Handle; see GPT_open
         0x0000,  // Global timer control(not used)
         0x5F04,  // timer1 control value
         0x0000,  // MSB of timer period value
         0x0000    // LSB of timer period value
         );
```

GPT_initChained32 *Intialize the timer in dual 32-bit chained mode*

Function

```
void  GPT_initChained32(
    GPT_Handle      hGpt,
    Uint16          gctl,
    Uint16          ctl1,
    Uint32          prdHigh,
    Uint32          prdLow
)
```

Arguments

hGpt	Device Handle; see GPT_open
gctl	Global timer control (not used)
ctl1	Timer1 control value
prdHigh	MSB of timer period value
prdLow	LSB bytes of timer period value

Return Value none

Description This API is used to set up and intialize two 32-bit timers in chained mode. It allows to initialize the period and also provide arguments to setup the timer control registers.

Example

```
GPT_Handle hGpt;
  ......
    GPT_initChained32(
      hGpt,
      0x0000, // Global timer control(not used)
      0x5F04, // Timer1 control value
      0x0000, // MSB of timer period value
      0x0000  // LSB of timer period value
     );
```

GPT_initDual32 *Intialize the timer in dual 32-bit unchained mode*

Function

```
void  GPT_initDual32(
    GPT_Handle      hGpt,
    Uint16          dt1ctl,
    Uint16          dt2ctl,
    Uint32          dt1prd,
    Uint32          dt2prd,
    Uint16          dt2prsc
)
```

Arguments

hGpt	Device Handle; see GPT_open
dt1ctl	Timer1 control value
dt2ctl	Timer2 control value
dt1prd	Timer1 period
dt2prd	Timer2 period
dt2prsc	Prescalar count

Return Value none

Description This API is used to set up and intialize the timer in dual 32-bit unchained mode. It allows to initialize the period for both the timers and also the prescalar counter which specify the count of the timer.It also provide arguments to setup the timer control registers.

Example

```
GPT_Handle hGpt;
  ......
  GPT_initDual32(
         hGpt,
         0x3FE, // ctl1
         0x3FE, // ctl2
         0x005, // prd1
         0x008, // prd2
         0x0FF  // psc34
  );
```

GPT_open — *Opens a GPT device for use*

Function

```
GPT_Handle  GPT_open(
    Uint16          devNum,
    Uint16          flags
)
```

Arguments

devNum	Specifies the GPT device to be opened flags
Open flags	GPT_OPEN_RESET: resets the GPT device

Return Value

GPT_HandleDevice Handle   INV: open failed

Description

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

If the GPT_OPEN_RESET flag is specified, the GPT module registers are set to their power-on defaults and any associated interrupts are disabled and cleared.

Example

```
Handle hGpt;
  ...
  hGpt = GPT_open(GPT_DEV0, GPT_OPEN_RESET);
```

GPT_reset — *Resets a GPT*

Function

```
void  GPT_reset(
    GPT_Handle       hGpt
)
```

Arguments

hGpt	Device Handle; see GPT_open

Return Value

none

Description

Resets the timer device. Disables and clears any interrupt events and sets the GPT registers to default values. If the handle is INV (-1) , all timer devices are reset.

Example

```
GPT_Handle hGpt;
  ......
  GPT_reset(hGpt);
```

GPT_start

Starts all the timers

Function

void GPT_start(
    GPT_Handle hGpt
)

Arguments

hGpt Device Handle; see GPT_open

Return Value none

Description Starts all the timers.

Example

```
GPT_Handle hGpt;
    ....
   GPT_start(hGpt);
```

GPT_start12

Starts the 32-bit timer1 device

Function

void GPT_start12(
    GPT_Handle hGpt
)

Arguments

hGpt Device Handle; see GPT_open

Return Value none

Description Starts the 32-bit timer1 device.

Example

```
GPT_Handle hGpt;
    ....
   GPT_start12(hGpt);
```

GPT_start34

Starts the 32-bit timer2 device

Function

void GPT_start34(
    GPT_Handle hGpt
)

Arguments

hGpt Device Handle; see GPT_open

Return Value none

Description Starts the 32-bit timer2 device.

Example

```
GPT_Handle hGpt;
    ....
   GPT_start34(hGpt);
```

GPT_stop *Stops the timer, if running*

Function

```
void  GPT_stop(
   GPT_Handle        hGpt
)
```

Arguments

hGpt	Device Handle. see GPT_open

Return Value none

Description Stops the timer, if running.

Example

```
GPT_Handle hGpt;
    ....
   GPT_stop(hGpt);
```

GPT_stop12 *Stops the 32-bit timer1 device, if running*

Function

```
void  GPT_stop12(
   GPT_Handle        hGpt
)
```

Arguments

hGpt	Device Handle; see GPT_open

Return Value none

Description Stops the 32-bit timer1 device, if running.

Example

```
GPT_Handle hGpt;
    ....
   GPT_stop12(hGpt);
```

GPT_stop34 *Stops the 32-bit timer2 device, if running*

Function

```
void  GPT_stop34(
    GPT_Handle        hGpt
)
```

Arguments

hGpt	Device Handle; see GPT_open

Return Value none

Description Stops the 32-bit timer2 device, if running.

Example

```
GPT_Handle hGpt;
     ....
   GPT_stop34(hGpt);
```

Index

L

M

N

O

P