TMS320C55x Chip Support Library API Reference Guide

SPRU433J
September 2004
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:

<table>
<thead>
<tr>
<th>Products</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amplifiers</td>
<td>amplifier.ti.com</td>
</tr>
<tr>
<td>Data Converters</td>
<td>dataconverter.ti.com</td>
</tr>
<tr>
<td>DSP</td>
<td>dsp.ti.com</td>
</tr>
<tr>
<td>Interface</td>
<td>interface.ti.com</td>
</tr>
<tr>
<td>Logic</td>
<td>logic.ti.com</td>
</tr>
<tr>
<td>Power Mgmt</td>
<td>power.ti.com</td>
</tr>
<tr>
<td>Microcontrollers</td>
<td>microcontroller.ti.com</td>
</tr>
<tr>
<td>Audio</td>
<td><a href="http://www.ti.com/audio">www.ti.com/audio</a></td>
</tr>
<tr>
<td>Automotive</td>
<td><a href="http://www.ti.com/automotive">www.ti.com/automotive</a></td>
</tr>
<tr>
<td>Broadband</td>
<td><a href="http://www.ti.com/broadband">www.ti.com/broadband</a></td>
</tr>
<tr>
<td>Digital Control</td>
<td><a href="http://www.ti.com/digitalcontrol">www.ti.com/digitalcontrol</a></td>
</tr>
<tr>
<td>Military</td>
<td><a href="http://www.ti.com/military">www.ti.com/military</a></td>
</tr>
<tr>
<td>Optical Networking</td>
<td><a href="http://www.ti.com/opticalnetwork">www.ti.com/opticalnetwork</a></td>
</tr>
<tr>
<td>Security</td>
<td><a href="http://www.ti.com/security">www.ti.com/security</a></td>
</tr>
<tr>
<td>Telephony</td>
<td><a href="http://www.ti.com/telephony">www.ti.com/telephony</a></td>
</tr>
<tr>
<td>Video &amp; Imaging</td>
<td><a href="http://www.ti.com/video">www.ti.com/video</a></td>
</tr>
<tr>
<td>Wireless</td>
<td><a href="http://www.ti.com/wireless">www.ti.com/wireless</a></td>
</tr>
</tbody>
</table>

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

Copyright © 2004, Texas Instruments Incorporated
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.
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

1 CSL Overview ................................................................. 1-1
   1.1 Introduction to CSL .................................................... 1-2
      1.1.1 How the CSL Benefits You ....................................... 1-2
      1.1.2 CSL Architecture ................................................... 1-2
   1.2 Naming Conventions .................................................... 1-6
   1.3 CSL Data Types .......................................................... 1-7
   1.4 CSL Functions ........................................................... 1-8
      1.4.1 Peripheral Initialization via Registers ......................... 1-9
      1.4.2 Peripheral Initialization via Functional Parameters .......... 1-10
   1.5 CSL Macros ............................................................. 1-11
   1.6 CSL Symbolic Constant Values ...................................... 1-13
   1.7 Resource Management and the Use of CSL Handles ............... 1-14
      1.7.1 Using CSL Handles ................................................ 1-14

2 How to Use CSL ............................................................. 2-1
   2.1 Overview ............................................................... 2-2
   2.2 Using the CSL ........................................................... 2-2
      2.2.1 Using the DMA_config() function ............................... 2-2
   2.3 Compiling and Linking with the CSL Using Code Composer Studio 2-7
      2.3.1 Specifying Your Target Device ................................. 2-7

3 ADC Module ................................................................. 3-1
   3.1 Overview ............................................................... 3-2
   3.2 Configuration Structures ............................................ 3-4
   3.3 Functions ............................................................... 3-5
   3.4 Macros ................................................................. 3-8
   3.5 Examples ............................................................. 3-9

4 CHIP Module ............................................................... 4-1
   4.1 Overview ............................................................... 4-2
      4.1.1 CHIP Registers ...................................................... 4-2
   4.2 Functions ............................................................... 4-3
   4.3 Macros ................................................................. 4-4

5 DAT Module ............................................................... 5-1
   5.1 Overview ............................................................... 5-2
   5.2 Functions ............................................................... 5-3
## Contents

<table>
<thead>
<tr>
<th>Module</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>6</strong> DMA Module</td>
<td>6-1</td>
</tr>
<tr>
<td>6.1 Overview</td>
<td>6-2</td>
</tr>
<tr>
<td>6.1.1 DMA Registers</td>
<td>6-4</td>
</tr>
<tr>
<td>6.2 Configuration Structures</td>
<td>6-5</td>
</tr>
<tr>
<td>6.3 Functions</td>
<td>6-6</td>
</tr>
<tr>
<td>6.4 Macros</td>
<td>6-11</td>
</tr>
<tr>
<td><strong>7</strong> EMIF Module</td>
<td>7-1</td>
</tr>
<tr>
<td>7.1 Overview</td>
<td>7-2</td>
</tr>
<tr>
<td>7.1.1 EMIF Registers</td>
<td>7-4</td>
</tr>
<tr>
<td>7.2 Configuration Structure</td>
<td>7-6</td>
</tr>
<tr>
<td>7.3 Functions</td>
<td>7-8</td>
</tr>
<tr>
<td>7.4 Macros</td>
<td>7-11</td>
</tr>
<tr>
<td><strong>8</strong> GPIO Module</td>
<td>8-1</td>
</tr>
<tr>
<td>8.1 Overview</td>
<td>8-2</td>
</tr>
<tr>
<td>8.2 Configuration Structure</td>
<td>8-4</td>
</tr>
<tr>
<td>8.3 Functions</td>
<td>8-5</td>
</tr>
<tr>
<td>8.4 Macros</td>
<td>8-17</td>
</tr>
<tr>
<td><strong>9</strong> HPI Module</td>
<td>9-1</td>
</tr>
<tr>
<td>9.1 Overview</td>
<td>9-2</td>
</tr>
<tr>
<td>9.2 Configuration Structures</td>
<td>9-4</td>
</tr>
<tr>
<td>9.3 Functions</td>
<td>9-5</td>
</tr>
<tr>
<td>9.4 Macros</td>
<td>9-6</td>
</tr>
<tr>
<td><strong>10</strong> I2C Module</td>
<td>10-1</td>
</tr>
<tr>
<td>10.1 Overview</td>
<td>10-2</td>
</tr>
<tr>
<td>10.1.1 I2C Registers</td>
<td>10-4</td>
</tr>
<tr>
<td>10.2 Configuration Structures</td>
<td>10-5</td>
</tr>
<tr>
<td>10.3 Functions</td>
<td>10-7</td>
</tr>
<tr>
<td>10.4 Macros</td>
<td>10-17</td>
</tr>
<tr>
<td>10.5 Examples</td>
<td>10-18</td>
</tr>
<tr>
<td><strong>11</strong> ICACHE Module</td>
<td>11-1</td>
</tr>
<tr>
<td>11.1 Overview</td>
<td>11-2</td>
</tr>
<tr>
<td>11.2 Configuration Structures</td>
<td>11-3</td>
</tr>
<tr>
<td>11.3 Functions</td>
<td>11-5</td>
</tr>
<tr>
<td>11.4 Macros</td>
<td>11-8</td>
</tr>
<tr>
<td><strong>12</strong> IRQ Module</td>
<td>12-1</td>
</tr>
<tr>
<td>12.1 Overview</td>
<td>12-2</td>
</tr>
<tr>
<td>12.1.1 The Event ID Concept</td>
<td>12-3</td>
</tr>
<tr>
<td>12.2 Using Interrupts with CSL</td>
<td>12-7</td>
</tr>
<tr>
<td>12.3 Configuration Structures</td>
<td>12-8</td>
</tr>
<tr>
<td>Section</td>
<td>Contents</td>
</tr>
<tr>
<td>---------</td>
<td>----------</td>
</tr>
<tr>
<td>12.4</td>
<td>Functions</td>
</tr>
<tr>
<td>13</td>
<td>McBSP Module</td>
</tr>
<tr>
<td>13.1</td>
<td>Overview</td>
</tr>
<tr>
<td>13.1.1</td>
<td>MCBSP Registers</td>
</tr>
<tr>
<td>13.2</td>
<td>Configuration Structures</td>
</tr>
<tr>
<td>13.3</td>
<td>Functions</td>
</tr>
<tr>
<td>13.4</td>
<td>Macros</td>
</tr>
<tr>
<td>13.5</td>
<td>Functions</td>
</tr>
<tr>
<td>13.6</td>
<td>Examples</td>
</tr>
<tr>
<td>14</td>
<td>MMC Module</td>
</tr>
<tr>
<td>14.1</td>
<td>Overview</td>
</tr>
<tr>
<td>14.2</td>
<td>Configuration Structures</td>
</tr>
<tr>
<td>14.3</td>
<td>Data Structures</td>
</tr>
<tr>
<td>14.4</td>
<td>Functions</td>
</tr>
<tr>
<td>15</td>
<td>PLL Module</td>
</tr>
<tr>
<td>15.1</td>
<td>Overview</td>
</tr>
<tr>
<td>15.2</td>
<td>Configuration Structures</td>
</tr>
<tr>
<td>15.3</td>
<td>Functions</td>
</tr>
<tr>
<td>15.4</td>
<td>Macros</td>
</tr>
<tr>
<td>16</td>
<td>PWR Module</td>
</tr>
<tr>
<td>16.1</td>
<td>Overview</td>
</tr>
<tr>
<td>16.1.1</td>
<td>PWR Registers</td>
</tr>
<tr>
<td>16.2</td>
<td>Functions</td>
</tr>
<tr>
<td>16.3</td>
<td>Macros</td>
</tr>
<tr>
<td>17</td>
<td>RTC Module</td>
</tr>
<tr>
<td>17.1</td>
<td>Overview</td>
</tr>
<tr>
<td>17.2</td>
<td>Configuration Structures</td>
</tr>
<tr>
<td>17.3</td>
<td>API Reference</td>
</tr>
<tr>
<td>17.4</td>
<td>Macros</td>
</tr>
<tr>
<td>18</td>
<td>Timer Module</td>
</tr>
<tr>
<td>18.1</td>
<td>Overview</td>
</tr>
<tr>
<td>18.2</td>
<td>Configuration Structures</td>
</tr>
<tr>
<td>18.3</td>
<td>Functions</td>
</tr>
<tr>
<td>18.4</td>
<td>Macros</td>
</tr>
<tr>
<td>19</td>
<td>UART Module</td>
</tr>
<tr>
<td>19.1</td>
<td>Overview</td>
</tr>
<tr>
<td>19.2</td>
<td>Configuration Structures</td>
</tr>
<tr>
<td>19.3</td>
<td>Functions</td>
</tr>
<tr>
<td>19.3.1</td>
<td>CSL Primary Functions</td>
</tr>
</tbody>
</table>

Contents ix
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>19.4 Macros</td>
<td>19-14</td>
</tr>
<tr>
<td>19.4.1 General Macros</td>
<td>19-14</td>
</tr>
<tr>
<td>19.4.2 UART Control Signal Macros</td>
<td>19-15</td>
</tr>
<tr>
<td><strong>20 WDTIM Module</strong></td>
<td><strong>20-1</strong></td>
</tr>
<tr>
<td>20.1 Overview</td>
<td>20-2</td>
</tr>
<tr>
<td>20.2 Configuration Structures</td>
<td>20-3</td>
</tr>
<tr>
<td>20.3 Functions</td>
<td>20-4</td>
</tr>
<tr>
<td>20.4 Macros</td>
<td>20-14</td>
</tr>
<tr>
<td><strong>21 GPT Module</strong></td>
<td><strong>21-1</strong></td>
</tr>
<tr>
<td>21.1 Overview</td>
<td>21-2</td>
</tr>
<tr>
<td>21.2 Configuration Structure</td>
<td>21-3</td>
</tr>
<tr>
<td>21.3 Functions</td>
<td>21-4</td>
</tr>
<tr>
<td></td>
<td>Description</td>
</tr>
<tr>
<td>---</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>1–1</td>
<td>CSL Modules</td>
</tr>
<tr>
<td>2–1</td>
<td>Defining the Target Device in the Build Options Dialog</td>
</tr>
<tr>
<td>2–2</td>
<td>Defining Large Memory Model</td>
</tr>
<tr>
<td>2–3</td>
<td>Defining Library Paths</td>
</tr>
</tbody>
</table>
## Tables

<table>
<thead>
<tr>
<th>Page</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>CSL Modules and Include Files</td>
</tr>
<tr>
<td>1-2</td>
<td>CSL Device Support</td>
</tr>
<tr>
<td>1-3</td>
<td>CSL Naming Conventions</td>
</tr>
<tr>
<td>1-4</td>
<td>CSL Data Types</td>
</tr>
<tr>
<td>1-5</td>
<td>Generic CSL Functions</td>
</tr>
<tr>
<td>1-6</td>
<td>Generic CSL Macros</td>
</tr>
<tr>
<td>1-7</td>
<td>Generic CSL Macros (Handle-based)</td>
</tr>
<tr>
<td>1-8</td>
<td>Generic CSL Symbolic Constants</td>
</tr>
<tr>
<td>2-1</td>
<td>CSL Directory Structure</td>
</tr>
<tr>
<td>3-1</td>
<td>ADC Configuration Structures</td>
</tr>
<tr>
<td>3-2</td>
<td>ADC Functions</td>
</tr>
<tr>
<td>3-3</td>
<td>ADC Registers</td>
</tr>
<tr>
<td>3-4</td>
<td>ADC Macros</td>
</tr>
<tr>
<td>4-1</td>
<td>CHIP Functions</td>
</tr>
<tr>
<td>4-2</td>
<td>CHIP Registers</td>
</tr>
<tr>
<td>4-3</td>
<td>CHIP Macros</td>
</tr>
<tr>
<td>5-1</td>
<td>DAT Functions</td>
</tr>
<tr>
<td>6-1</td>
<td>DMA Configuration Structure</td>
</tr>
<tr>
<td>6-2</td>
<td>DMA Functions</td>
</tr>
<tr>
<td>6-3</td>
<td>DMA Macros</td>
</tr>
<tr>
<td>6-4</td>
<td>DMA Registers</td>
</tr>
<tr>
<td>7-1</td>
<td>EMIF Configuration Structure</td>
</tr>
<tr>
<td>7-2</td>
<td>EMIF Functions</td>
</tr>
<tr>
<td>7-3</td>
<td>Registers</td>
</tr>
<tr>
<td>7-4</td>
<td>EMIF CSL Macros Using EMIF Port Number</td>
</tr>
<tr>
<td>8-1</td>
<td>GPIO Functions</td>
</tr>
<tr>
<td>8-2</td>
<td>GPIO Registers</td>
</tr>
<tr>
<td>8-3</td>
<td>GPIO CSL Macros</td>
</tr>
<tr>
<td>9-1</td>
<td>HPI Module Configuration Structure</td>
</tr>
<tr>
<td>9-2</td>
<td>HPI Functions</td>
</tr>
<tr>
<td>9-3</td>
<td>HPI Registers and Bit Field Names</td>
</tr>
<tr>
<td>9-4</td>
<td>HPI Macros</td>
</tr>
<tr>
<td>10-1</td>
<td>I2C Configuration Structure</td>
</tr>
<tr>
<td>10-2</td>
<td>I2C Functions</td>
</tr>
<tr>
<td>10-3</td>
<td>I2C Registers</td>
</tr>
<tr>
<td>10-4</td>
<td>I2C Macros</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>11-1</td>
<td>ICACHE Configuration Structure</td>
</tr>
<tr>
<td>11-2</td>
<td>ICACHE Functions</td>
</tr>
<tr>
<td>11-3</td>
<td>ICACHE CSL Macros</td>
</tr>
<tr>
<td>12-1</td>
<td>IRQ Configuration Structure</td>
</tr>
<tr>
<td>12-2</td>
<td>IRQ Functions</td>
</tr>
<tr>
<td>12-3</td>
<td>IRQ_EVT_NNNN Events List</td>
</tr>
<tr>
<td>13-1</td>
<td>McBSP Configuration Structure</td>
</tr>
<tr>
<td>13-2</td>
<td>McBSP Functions</td>
</tr>
<tr>
<td>13-3</td>
<td>MCBSP Registers</td>
</tr>
<tr>
<td>13-4</td>
<td>McBSP Macros Using McBSP Port Number</td>
</tr>
<tr>
<td>13-5</td>
<td>McBSP CSL Macros Using Handle</td>
</tr>
<tr>
<td>14-1</td>
<td>MMC Configuration Structures</td>
</tr>
<tr>
<td>14-2</td>
<td>MMC Data Structures</td>
</tr>
<tr>
<td>14-3</td>
<td>MMC Functions</td>
</tr>
<tr>
<td>14-4</td>
<td>OCR Register Definitions</td>
</tr>
<tr>
<td>15-1</td>
<td>PLL Configuration Structure</td>
</tr>
<tr>
<td>15-2</td>
<td>PLL Functions</td>
</tr>
<tr>
<td>15-3</td>
<td>PLL Registers</td>
</tr>
<tr>
<td>15-4</td>
<td>PLL CSL Macros Using PLL Port Number</td>
</tr>
<tr>
<td>16-1</td>
<td>PWR Functions</td>
</tr>
<tr>
<td>16-2</td>
<td>PWR Registers</td>
</tr>
<tr>
<td>16-3</td>
<td>PWR CSL Macros</td>
</tr>
<tr>
<td>17-1</td>
<td>RTC Configuration Structures</td>
</tr>
<tr>
<td>17-2</td>
<td>RTC Functions</td>
</tr>
<tr>
<td>17-3</td>
<td>RTC ANSI C-Style Time Functions</td>
</tr>
<tr>
<td>17-4</td>
<td>RTC Macros</td>
</tr>
<tr>
<td>17-5</td>
<td>Registers</td>
</tr>
<tr>
<td>18-1</td>
<td>TIMER Configuration Structure</td>
</tr>
<tr>
<td>18-2</td>
<td>TIMER Functions</td>
</tr>
<tr>
<td>18-3</td>
<td>Registers</td>
</tr>
<tr>
<td>18-4</td>
<td>TIMER CSL Macros Using Timer Port Number</td>
</tr>
<tr>
<td>18-5</td>
<td>TIMER CSL Macros Using Handle</td>
</tr>
<tr>
<td>19-1</td>
<td>UART APIs</td>
</tr>
<tr>
<td>19-2</td>
<td>UART CSL Macros</td>
</tr>
<tr>
<td>20-1</td>
<td>WDTIM Structure and APIs</td>
</tr>
<tr>
<td>20-2</td>
<td>WDTIM CSL Macros</td>
</tr>
<tr>
<td>21-1</td>
<td>GPT Configuration Structure</td>
</tr>
<tr>
<td>21-2</td>
<td>GPT Functions</td>
</tr>
</tbody>
</table>
# Examples

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1−1</td>
<td>Using PER_config</td>
<td>1-10</td>
</tr>
<tr>
<td>1−2</td>
<td>Using PER_setup()</td>
<td>1-10</td>
</tr>
<tr>
<td>2-1</td>
<td>Using a Linker Command File</td>
<td>2-12</td>
</tr>
<tr>
<td>12−1</td>
<td>Manual Interrupt Setting Outside DSP/BIOS HWIs</td>
<td>12-7</td>
</tr>
<tr>
<td>13−1</td>
<td>McBSP Port Initialization Using MCBSP_config()</td>
<td>13-26</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Introduction to CSL</td>
<td>1-2</td>
</tr>
<tr>
<td>1.2 Naming Conventions</td>
<td>1-6</td>
</tr>
<tr>
<td>1.3 CSL Data Types</td>
<td>1-7</td>
</tr>
<tr>
<td>1.4 CSL Functions</td>
<td>1-8</td>
</tr>
<tr>
<td>1.5 CSL Macros</td>
<td>1-11</td>
</tr>
<tr>
<td>1.6 CSL Symbolic Constant Values</td>
<td>1-13</td>
</tr>
<tr>
<td>1.7 Resource Management and the Use of CSL Handles</td>
<td>1-14</td>
</tr>
</tbody>
</table>
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

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

† 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**

<table>
<thead>
<tr>
<th>Device</th>
<th>Small-Model Library</th>
<th>Large-Model Library</th>
<th>Device Support Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>C5501</td>
<td>csl5501.lib</td>
<td>csl5501x.lib</td>
<td>CHIP_5501</td>
</tr>
<tr>
<td>C5502</td>
<td>csl5502.lib</td>
<td>csl5502x.lib</td>
<td>CHIP_5502</td>
</tr>
<tr>
<td>C5509</td>
<td>csl5509.lib</td>
<td>csl5509x.lib</td>
<td>CHIP_5509</td>
</tr>
<tr>
<td>C5509A</td>
<td>csl5509a.lib</td>
<td>CSL5509ax.lig</td>
<td>CHIP_5509A</td>
</tr>
<tr>
<td>C5510PG1.0</td>
<td>csl5510PG1_0.lib</td>
<td>csl5510PG1_0x.lib</td>
<td>CHIP_5510PG1_0</td>
</tr>
<tr>
<td>C5510PG1.2</td>
<td>csl5510PG1_2.lib</td>
<td>csl5510PG1_2x.lib</td>
<td>CHIP_5510PG1_2</td>
</tr>
<tr>
<td>C5510PG2.0</td>
<td>csl5510PG2_0.lib</td>
<td>csl5510PG2_0x.lib</td>
<td>CHIP_5510PG2_0</td>
</tr>
<tr>
<td>C5510PG2.1</td>
<td>csl5510PG2_1.lib</td>
<td>csl5510PG2_1x.lib</td>
<td>CHIP_5510PG2_1</td>
</tr>
<tr>
<td>C5510PG2.2</td>
<td>csl5510PG2_2.lib</td>
<td>csl5510PG2_2x.lib</td>
<td>CHIP_5510PG2_2</td>
</tr>
</tbody>
</table>
1.2 Naming Conventions

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

Table 1–3. CSL Naming Conventions

<table>
<thead>
<tr>
<th>Object Type</th>
<th>Naming Convention</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>PER_funcName()†</td>
</tr>
<tr>
<td>Variable</td>
<td>PER_varName()†</td>
</tr>
<tr>
<td>Macro</td>
<td>PER_MACRO_NAME†</td>
</tr>
<tr>
<td>Typedef</td>
<td>PER_Typename†</td>
</tr>
<tr>
<td>Function Argument</td>
<td>funcArg</td>
</tr>
<tr>
<td>Structure Member</td>
<td>memberName</td>
</tr>
</tbody>
</table>

† 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

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSLBool</td>
<td>unsigned short</td>
</tr>
<tr>
<td>PER_Handle</td>
<td>void *</td>
</tr>
<tr>
<td>Int16</td>
<td>short</td>
</tr>
<tr>
<td>Int32</td>
<td>long</td>
</tr>
<tr>
<td>Uchar</td>
<td>unsigned char</td>
</tr>
<tr>
<td>UInt16</td>
<td>unsigned short</td>
</tr>
<tr>
<td>UInt32</td>
<td>unsigned long</td>
</tr>
<tr>
<td>DMA_AdrPtr</td>
<td>void (*DMA_AdrPtr)()</td>
</tr>
<tr>
<td></td>
<td>pointer to a void function</td>
</tr>
</tbody>
</table>
1.4 CSL Functions

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

\begin{table}
\begin{tabular}{|c|c|}
\hline
\textbf{Note:} & \\
\hline
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. & \\
\hline
\end{tabular}
\end{table}

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

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

CSL functions provide a way to program peripherals by:

- **Direct register initialization** using the \texttt{PER\_config()} function (see section 1.4.1).
- **Using functional parameters** using the \texttt{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.

\begin{table}
\begin{tabular}{|c|c|}
\hline
\textbf{Note:} & \\
\hline
These functions are not available for all CSL peripheral modules. & \\
\hline
\end{tabular}
\end{table}
CSL Functions

Table 1–5. Generic CSL Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>handle = PER_open(channelNumber, [priority,] flags)</code></td>
<td>Opens a peripheral channel and then performs the operation indicated by <code>flags</code>; must be called before using a channel. The return value is a unique device handle to use in subsequent API calls. The <code>priority</code> parameter applies only to the DAT module.</td>
</tr>
</tbody>
</table>
| `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.
CSL Functions

Example 1–1. Using PER_config

```c
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()

```c
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**

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PER_REG_RMK(****,</strong>&lt;br&gt;    <strong>fieldval_15,</strong>&lt;br&gt;    .&lt;br&gt;    .&lt;br&gt;    .&lt;br&gt;    <strong>fieldval_0</strong>**)**</td>
<td>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:&lt;br&gt;  - Defined only for registers with more than one field.&lt;br&gt;  - Include only fields that are writable.&lt;br&gt;  - Specify field arguments as most-significant bit first.&lt;br&gt;  - Whether or not they are used, all writable field values must be included.&lt;br&gt;  - If you pass a field value exceeding the number of bits allowed for that particular field, the _RMK macro truncates that field value.</td>
</tr>
<tr>
<td><strong>PER_RGET(REG#****,</strong>&lt;br&gt;    )</td>
<td>Returns the value in the peripheral register.</td>
</tr>
<tr>
<td><strong>PER_RSET(REG#,****,</strong>&lt;br&gt;    regval****,**&lt;br&gt;    )</td>
<td>Writes the value to the peripheral register.</td>
</tr>
</tbody>
</table>
### Table 1–6. Generic CSL Macros (Continued)

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PER_FMK(REG, FIELD, fieldval)</code></td>
<td>Creates a shifted version of <code>fieldval</code> 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.</td>
</tr>
<tr>
<td><code>PER_FGET(REG#, FIELD)</code></td>
<td>Returns the value of the specified <code>FIELD</code> in the peripheral register.</td>
</tr>
<tr>
<td><code>PER_FSET(REG#, FIELD, fieldval)</code></td>
<td>Writes <code>fieldval</code> to the specified <code>FIELD</code> in the peripheral register.</td>
</tr>
<tr>
<td><code>PER_ADDR(REG#)</code></td>
<td>If applicable, gets the memory address (or sub-address) of the peripheral register REG#.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PER_RGETH(handle, REG)</code></td>
<td>Returns the value of the peripheral register REG associated with Handle.</td>
</tr>
<tr>
<td><code>PER_RSETH(handle, REG, regval)</code></td>
<td>Writes the value to the peripheral register REG associated with Handle.</td>
</tr>
<tr>
<td><code>PER_ADDRH(handle, REG)</code></td>
<td>If applicable, gets the memory address (or sub-address) of the peripheral register REG associated with Handle.</td>
</tr>
<tr>
<td><code>PER_FGETH(handle, REG, FIELD)</code></td>
<td>Returns the value of the specified <code>FIELD</code> in the peripheral register REG associated with Handle.</td>
</tr>
<tr>
<td><code>PER_FSETH(handle, REG, FIELD, fieldval)</code></td>
<td>Sets the value of the specified <code>FIELD</code> in the peripheral register REG to fieldval.</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PER_REG_DEFAULT</strong></td>
<td>Default value for a register; corresponds to the register value after a reset or to 0 if a reset has no effect.</td>
</tr>
</tbody>
</table>

(b) Constant Values for Fields

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PER_REG_FIELD_SYMVAL</strong></td>
<td>Symbolic constant to specify values for individual fields in the specified peripheral register.</td>
</tr>
<tr>
<td><strong>PER_REG_FIELD_DEFAULT</strong></td>
<td>Default value for a field; corresponds to the field value after a reset or to 0 if a reset has no effect.</td>
</tr>
</tbody>
</table>
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:

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

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

```c
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:

```c
DMA_start(myDma); /* Begin transfer */
```

```c
DMA_close(myDma); /* Free DMA channel */
```
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.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Overview</td>
<td>2-2</td>
</tr>
<tr>
<td>2.2 Using the CSL</td>
<td>2-2</td>
</tr>
<tr>
<td>2.3 Compiling and Linking with the CSL Using Code Composer Studio</td>
<td>2-7</td>
</tr>
</tbody>
</table>
## 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.

```c
#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 */
```
Using the CSL

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 */

How to Use CSL 2-3
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_REV0
 DMA_DMACCR_SYNC_XEVT0
 DMA_DMACCR_SYNC_REVTA0
 DMA_DMACCR_SYNC_XEVT1
 DMA_DMACCR_SYNC_REV1
 DMA_DMACCR_SYNC_XEVT1
 DMA_DMACCR_SYNC_REVTA1
 DMA_DMACCR_SYNC_XEVT1
 DMA_DMACCR_SYNC_REVTA1
 DMA_DMACCR_SYNC_REVT2
DMA_DMCCR_SYNC_XEVT2
DMA_DMCCR_SYNC_REVTA2
DMA_DMCCR_SYNC_XEVT2
DMA_DMCCR_SYNC_TIM1INT
DMA_DMCCR_SYNC_TIM2INT
DMA_DMCCR_SYNC_EXTINT0
DMA_DMCCR_SYNC_EXTINT1
DMA_DMCCR_SYNC_EXTINT2
DMA_DMCCR_SYNC_EXTINT3
DMA_DMCCR_SYNC_EXTINT4
DMA_DMCCR_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 interrupt enable :-
DMA_DMACICR_TIMEOUTIE_ON
DMA_DMACICR_TIMEOUTIE_OFF */

), /* DMACCR */

(DMA_AdrPtr) &src, /* DMACSSAL */
0, /* DMACSSAU */
(DMA_AdrPtr)&dst, /* DMACDSAL */
0, /* DMACDSAU */
N, /* DMACEN */
1, /* DACFEN */
0, /* DACFI */
0 /* DACOE */

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

```c
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:

```c
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(),...):

```c
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:

```c
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:

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

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

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

Step 9: Close DMA channel

```c
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>
<thead>
<tr>
<th>This CSL component...</th>
<th>Is located in this directory...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Libraries</td>
<td>&lt;Install_Dir&gt;\c5500\csl\lib</td>
</tr>
<tr>
<td>Source Library</td>
<td>&lt;Install_Dir&gt;\c5500\csl\lib</td>
</tr>
<tr>
<td>Include files</td>
<td>&lt;Install_Dir&gt;\c5500\csl\include</td>
</tr>
<tr>
<td>Examples</td>
<td>&lt;Install_Dir&gt;\examples&lt;target&gt;\csl</td>
</tr>
<tr>
<td>Documentation</td>
<td>&lt;Install_Dir&gt;\docs</td>
</tr>
</tbody>
</table>

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.
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**

```plaintext
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.
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.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Overview</td>
<td>3-2</td>
</tr>
<tr>
<td>3.2 Configuration Structures</td>
<td>3-4</td>
</tr>
<tr>
<td>3.3 Functions</td>
<td>3-5</td>
</tr>
<tr>
<td>3.4 Macros</td>
<td>3-8</td>
</tr>
<tr>
<td>3.5 Examples</td>
<td>3-9</td>
</tr>
</tbody>
</table>
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**

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
<th>See page…</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC_Config</td>
<td>ADC configuration structure used to set up the ADC (register based)</td>
<td>3-4</td>
</tr>
</tbody>
</table>

Table 3–2. **ADC Functions**

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
<th>See page…</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC_config()</td>
<td>Sets up the ADC using the configuration structure</td>
<td>3-5</td>
</tr>
<tr>
<td>ADC_getConfig()</td>
<td>Obtains the current configuration of all the ADC registers</td>
<td>3-5</td>
</tr>
<tr>
<td>ADC_read()</td>
<td>Performs conversion and reads sampled values from the data register</td>
<td>3-6</td>
</tr>
<tr>
<td>ADC_setFreq()</td>
<td>Sets up the ADC using parameters passed</td>
<td>3-6</td>
</tr>
</tbody>
</table>
### Table 3-3. ADC Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADCCTL</td>
<td>CHSELECT, ADCSTART</td>
</tr>
<tr>
<td>ADCDATA</td>
<td>ADCDATA(R), CHSELECT, ADCBUSY(R)</td>
</tr>
<tr>
<td>ADCCLKDIV</td>
<td>CONVRATEDIV, SAMPTIMEDIV</td>
</tr>
<tr>
<td>ADCCLKCTL</td>
<td>CPUCLKDIV, IDLEEN</td>
</tr>
</tbody>
</table>

**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 configuration structure used to set up the ADC interface

Structure | ADC_Config
---|---
Members | Uint16 adcctl | Control Register
       | Uint16 adclklklkdiv | 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, /* ADCCCLKDIV */
            | 0xFFFF /* ADCCLKCTL */
            | }
3.3 Functions

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC_config</td>
<td>Writes the values to ADC registers using the configuration structure</td>
<td>void ADC_config(ADC_Config *Config);</td>
</tr>
<tr>
<td>Arguments</td>
<td>Pointer to an initialized configuration structure (see ADC_Config)</td>
<td></td>
</tr>
<tr>
<td>Return Value</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Description</td>
<td>Writes a value to set up the ADC using the configuration structure. The values of the configuration structure are written to the port registers.</td>
<td>ADC_Config Config = {0xFFFF, /* ADCCTL <em>/ 0xFFFF, /</em> ADCCLKDIV <em>/ 0xFFFF /</em> ADCCLKCTL */};</td>
</tr>
<tr>
<td>Example</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC_getConfig</td>
<td>Writes values to ADC registers using the configuration structure</td>
<td>void ADC_getConfig(ADC_Config *Config);</td>
</tr>
<tr>
<td>Arguments</td>
<td>Pointer to a configuration structure (see ADC_Config)</td>
<td></td>
</tr>
<tr>
<td>Return Value</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Description</td>
<td>Reads the current value of all ADC registers being used and places them into the corresponding configuration structure member.</td>
<td>ADC_Config testConfig;</td>
</tr>
<tr>
<td>Example</td>
<td>ADC_getConfig(&amp;testConfig);</td>
<td></td>
</tr>
</tbody>
</table>
ADC_read

Performs an ADC conversion and reads the digital data

Function

```
void ADC_read(int channelnumber,
              Uint16 *data,
              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

```c
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 an 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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC_RGET()</td>
<td>Uint16 ADC_RGET(REG)</td>
</tr>
<tr>
<td>ADC_RSET()</td>
<td>Void ADC_RSET(REG, Uint16 regval)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC_FGET()</td>
<td>Uint16 ADC_FGET(REG, FIELD)</td>
</tr>
<tr>
<td>ADC_FSET()</td>
<td>Void ADC_FSET(REG,FIELD,Uint16 fieldval)</td>
</tr>
</tbody>
</table>

Notes:
1) REG indicates the registers, ADCCTL, ADCCCLKDIV, ADCCCLKCTL
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)

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC_REG_RMK()</td>
<td>Uint16 ADC_REG_RMK(fieldval_n,…,fieldval_0)</td>
</tr>
<tr>
<td>Note:</td>
<td>*Start with field values with most significant field positions:</td>
</tr>
<tr>
<td></td>
<td>field_n: MSB field</td>
</tr>
<tr>
<td></td>
<td>field_0: LSB field</td>
</tr>
<tr>
<td></td>
<td>*only writable fields allowed</td>
</tr>
</tbody>
</table>

ADC_FMK()     | Uint16 ADC_FMK(REG, FIELD, fieldval)                                   |

(d) Macros to read a register address

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC_ADDR()</td>
<td>Uint16 ADC_ADDR(REG)</td>
</tr>
</tbody>
</table>

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).
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.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Overview</td>
<td>4-2</td>
</tr>
<tr>
<td>4.2 Functions</td>
<td>4-3</td>
</tr>
<tr>
<td>4.3 Macros</td>
<td>4-4</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>See page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHIP_getDieId_High32</td>
<td>Returns the high 32 bits of the DieID register.</td>
<td>4-3</td>
</tr>
<tr>
<td>CHIP_getDieId_Low32</td>
<td>Returns the low 32 bits of the DieID register.</td>
<td>4-3</td>
</tr>
<tr>
<td>CHIP_getRevId</td>
<td>Returns the value of the RevID register.</td>
<td>4-3</td>
</tr>
</tbody>
</table>

### 4.1.1 CHIP Registers

### Table 4–2. CHIP Registers

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

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

The following are functions available for use with the CHIP 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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHIP_RGET()</td>
<td>Uint16 CHIP_RGET(REG)</td>
</tr>
<tr>
<td>CHIP_RSET()</td>
<td>void CHIP_RSET(REG, Uint16 regval)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHIP_FGET()</td>
<td>Uint16 CHIP_FGET(REG, FIELD)</td>
</tr>
<tr>
<td>CHIP_FSET()</td>
<td>void CHIP_FSET(REG, FIELD, Uint16 fieldval)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHIP_REG_RMK()</td>
<td>Uint16 CHIP_REG_RMK(fieldval_n,...fieldval_0)</td>
</tr>
<tr>
<td></td>
<td>Note: *Start with field values with most significant field positions:</td>
</tr>
<tr>
<td></td>
<td>field_n: MSB field</td>
</tr>
<tr>
<td></td>
<td>field_0: LSB field</td>
</tr>
<tr>
<td></td>
<td>* only writeable fields allowed</td>
</tr>
<tr>
<td>CHIP_FMK()</td>
<td>Uint16 CHIP_FMK(REG, FIELD, fieldval)</td>
</tr>
</tbody>
</table>

(d) Macros to read a register address

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHIP_ADDR()</td>
<td>Uint16 CHIP_ADDR(REG)</td>
</tr>
</tbody>
</table>

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).
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.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Overview</td>
<td>5-2</td>
</tr>
<tr>
<td>5.2 Functions</td>
<td>5-3</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
<th>See page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAT_close()</td>
<td>Closes the DAT</td>
<td>5-3</td>
</tr>
<tr>
<td>DAT_copy()</td>
<td>Copies data of specific length from the source memory to the destination memory.</td>
<td>5-3</td>
</tr>
<tr>
<td>DAT_copy2D()</td>
<td>Copies 2D data of specific line length from the source memory to the destination memory.</td>
<td>5-4</td>
</tr>
<tr>
<td>DAT_fill()</td>
<td>Fills the destination memory with a data value</td>
<td>5-5</td>
</tr>
<tr>
<td>DAT_open()</td>
<td>Opens the DAT with a channel number and a channel priority</td>
<td>5-6</td>
</tr>
<tr>
<td>DAT_wait()</td>
<td>DAT wait function</td>
<td>5-7</td>
</tr>
</tbody>
</table>
5.2 Functions

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

**DAT_close**  
_Closes a DAT device_

**Function**  

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

**Arguments**  

- `hDat` : Device handle.

**Return Value**  

None

**Description**  

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

**Example**  

```c
DAT_close(hDat);
```

**DAT_copy**  
_Performs bytewise copy from source to destination memory_

**Function**  

```c
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**  

```c
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.
**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**

```c
Uint16 value;

DAT_fill(hDat, /* Device Handler */
    (DMA_AdrPtr)0x00FF, /* dst */
    0x0010, /* ElemCnt */
    &value /* Value */
);
```

---

**Example**

```c
DAT_copy2D(hDat, /* Device Handler */
    DAT_2D2D, /* Type */
    (DMA_AdrPtr)0xFF00, /* src */
    (DMA_AdrPtr)0xF000, /* dst */
    0x0010, /* linelen */
    0x0004, /* Line Cnt */
    0x0110, /* LinePitch */
);
```
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);
<table>
<thead>
<tr>
<th><strong>DAT_wait</strong></th>
<th><strong>DAT wait function</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function</strong></td>
<td>void DAT_wait</td>
</tr>
<tr>
<td></td>
<td>DAT_Handle hDat</td>
</tr>
<tr>
<td></td>
<td>);</td>
</tr>
<tr>
<td><strong>Arguments</strong></td>
<td>hDat Device handler (see DAT_open).</td>
</tr>
<tr>
<td><strong>Return Value</strong></td>
<td>none</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td>DAT_wait(myhDat);</td>
</tr>
</tbody>
</table>
This chapter describes the DMA module, lists the API structure, functions, and macros within the module, and provides a DMA API reference section.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Overview</td>
<td>6-2</td>
</tr>
<tr>
<td>6.2 Configuration Structures</td>
<td>6-5</td>
</tr>
<tr>
<td>6.3 Functions</td>
<td>6-6</td>
</tr>
<tr>
<td>6.4 Macros</td>
<td>6-11</td>
</tr>
</tbody>
</table>
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.
Overview

**Table 6–1. DMA Configuration Structure**

<table>
<thead>
<tr>
<th>Configuration Structure</th>
<th>Description</th>
<th>See page ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA_Config</td>
<td>DMA configuration structure used to setup the DMA interface</td>
<td>6-5</td>
</tr>
</tbody>
</table>

**Table 6–2. DMA Functions**

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

**Table 6–3. DMA Macros**

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

**Table 6-4. DMA Registers**

<table>
<thead>
<tr>
<th>Register</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMAGCR</td>
<td>FREE, EHPIEXCL, EHPIPRIO</td>
</tr>
<tr>
<td>DMACSDP</td>
<td>DSTBEN, DSTPACK, DST, SRCBEN, SRCPACK, SRC, DATATYPE</td>
</tr>
<tr>
<td>DMACCR</td>
<td>DSTAMODE, SRCAMODE, ENDPROG, FIFOFLUSH, REPEAT, AUTOINIT, EN, PRIO, FS, SYNC</td>
</tr>
<tr>
<td>DMACICR</td>
<td>BLOCKIE, LASTIE, FRAMEIE, FIRSTHALFIE, DROPIE, TIMEOUTIE</td>
</tr>
<tr>
<td>DMACSR</td>
<td>(R)SYNC, (R)BLOCK, (R)LAST, (R)FRAME, (R)HALF, (R)DROP, (R)TIMEOUT</td>
</tr>
<tr>
<td>DMACSSAL</td>
<td>SSAL</td>
</tr>
<tr>
<td>DMACSSAU</td>
<td>SSAU</td>
</tr>
<tr>
<td>DMACDSAL</td>
<td>DSAL</td>
</tr>
<tr>
<td>DMACDSAU</td>
<td>DSAU</td>
</tr>
<tr>
<td>DMACEN</td>
<td>ELEMENTNUM</td>
</tr>
<tr>
<td>DMACFI</td>
<td>FRAMENDX</td>
</tr>
<tr>
<td>DMACEI</td>
<td>ELEMENTNDX</td>
</tr>
<tr>
<td>DMACSF1</td>
<td>FRAMENDX</td>
</tr>
<tr>
<td>DMACSEI</td>
<td>ELEMENTNDX</td>
</tr>
<tr>
<td>DMACDF1</td>
<td>FRAMENDX</td>
</tr>
<tr>
<td>DMACDEI</td>
<td>ELEMENTNDX</td>
</tr>
<tr>
<td>DMACSAC</td>
<td>DMACSAC</td>
</tr>
<tr>
<td>DMACDAC</td>
<td>DMACDAC</td>
</tr>
<tr>
<td>DMAGTCR</td>
<td>PTE, ETE, ITE1, ITE0</td>
</tr>
<tr>
<td>DMAGTCR</td>
<td>DTCE, STCE</td>
</tr>
<tr>
<td>DMAGSRCR</td>
<td>COMPMODE</td>
</tr>
</tbody>
</table>

**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**

<table>
<thead>
<tr>
<th>Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA_Config</td>
<td>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 <em>DMA_RMK</em> macros to create the structure member values.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Members</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uint16 dmacsdp</td>
<td>DMA Channel Control Register</td>
</tr>
<tr>
<td>Uint16 dmaccr</td>
<td>DMA Channel Interrupt Register</td>
</tr>
<tr>
<td>Uint16 dmacicr</td>
<td>DMA Channel Status Register</td>
</tr>
<tr>
<td>(DMA_AdrPtr) dmacssal</td>
<td>DMA Channel Source Start Address (Lower Bits)</td>
</tr>
<tr>
<td>Uint16 dmacssau</td>
<td>DMA Channel Source Start Address (Upper Bits)</td>
</tr>
<tr>
<td>(DMA_AdrPtr) dmacdsal</td>
<td>DMA Channel Source Destination Address (Lower Bits)</td>
</tr>
<tr>
<td>Uint16 dmacdsau</td>
<td>DMA Channel Source Destination Address (Upper Bits)</td>
</tr>
<tr>
<td>Uint16 dmacen</td>
<td>DMA Channel Element Number Register</td>
</tr>
<tr>
<td>Uint16 dmacfn</td>
<td>DMA Channel Frame Number Register</td>
</tr>
</tbody>
</table>

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, 5501
- 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 |

**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**

```c
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**

```c
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

**Function**

```c
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**

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

---

DMA_getEventId

**Function**

```c
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**

```c
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:

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

**Example 2**

For global registers:

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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Uint16 DMA_FGETH (DMA_Handle hDma, LOCALREG, FIELD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td></td>
</tr>
<tr>
<td>hDma</td>
<td>Handle to DMA channel that identifies the specific DMA channel used.</td>
</tr>
<tr>
<td>LOCALREG</td>
<td>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.</td>
</tr>
<tr>
<td>FIELD</td>
<td>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.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Value of register field given by FIELD.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Gets the DMA register field value</td>
</tr>
<tr>
<td>Example</td>
<td></td>
</tr>
</tbody>
</table>

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

DMA Module
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**

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

| 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**

```c
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**

```c
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:
        - `DMACSF#`
        - `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:

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

**Example 2**

For global registers:

```c
Uint16 myVar;
...
myVar = DMA_RGET(DMGCR);
```
**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**

```c
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:

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

**Example 2**

For global registers:

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

<table>
<thead>
<tr>
<th>Macro</th>
<th>void DMA_RSETH (DMA_Handle hDma, LOCALREG, Uint16 regval)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td>hDma Handle to DMA channel that identifies the specific DMA channel used.</td>
</tr>
<tr>
<td></td>
<td>LOCALREG Same register as in DMA_RGET(), but without channel number (#). Example: DMACSDP (instead of DMACSDP#)</td>
</tr>
<tr>
<td></td>
<td>regval value to write to register LOCALREG for the channel associated with handle.</td>
</tr>
<tr>
<td>Return Value</td>
<td>None</td>
</tr>
<tr>
<td>Description</td>
<td>Sets the DMA register LOCALREG for the channel associated with handle to the value regval.</td>
</tr>
<tr>
<td>Example</td>
<td>DMA_Handle myHandle;</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>myHandle = DMA_open (DMA_CHA0, DMA_OPEN_RESET);</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>DMA_RSETH (myHandle, DMACSDP, 0x123);</td>
</tr>
</tbody>
</table>
This chapter describes the EMIF module, lists the API structure, functions, and macros within the module, and provides an EMIF API reference section.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Overview</td>
<td>7-2</td>
</tr>
<tr>
<td>7.2 Configuration Structures</td>
<td>7-6</td>
</tr>
<tr>
<td>7.3 Functions</td>
<td>7-8</td>
</tr>
<tr>
<td>7.4 Macros</td>
<td>7-11</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
<th>See page ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMIF_Config</td>
<td>EMIF configuration structure used to setup the EMIF interface</td>
<td>7-6</td>
</tr>
</tbody>
</table>

Table 7–2. EMIF Functions

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
<th>See page ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMIF_config()</td>
<td>Sets up EMIF using configuration structure (EMIF_Config)</td>
<td>7-8</td>
</tr>
<tr>
<td>EMIF_getConfig()</td>
<td>Reads the EMIF configuration structure</td>
<td>7-9</td>
</tr>
<tr>
<td>EMIF_enterselfRefresh</td>
<td>Places SDRAM in refresh mode</td>
<td>7-9</td>
</tr>
<tr>
<td>(for 5509A only)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EMIF_exitselfRefresh</td>
<td>SDRAM exit refresh mode</td>
<td>7-10</td>
</tr>
<tr>
<td>(for 5509A only)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EMIF_reset</td>
<td>Resets memory connected in EMIF CE Space</td>
<td>7-10</td>
</tr>
<tr>
<td>(for 5510xx, 5509, 5509A only)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 7.1.1 EMIF Registers

#### Table 7–3. Registers

(a) **EMIF Registers**

<table>
<thead>
<tr>
<th>Register</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>EGCR</td>
<td>MEMFREQ, WPE, MEMCEN, (R)ARDY, (R)HOLD, (R)HOLDA, NOHOLD</td>
</tr>
<tr>
<td>EMIRST</td>
<td>(W)EMIRST</td>
</tr>
<tr>
<td>EMIBE</td>
<td>(R)TIME, (R)CE3, (R)CE2, (R)CE1, (R)CE0, (R)DMA, (R)EBUS, (R)EBUS, (R)DBUS, (R)CBUS, (R)PBUS</td>
</tr>
<tr>
<td>CE01</td>
<td>MTYPE, RDSETUP, RDSTROBE, RDHOLD</td>
</tr>
<tr>
<td>CE11</td>
<td>MTYPE, RDSETUP, RDSTROBE, RDHOLD</td>
</tr>
<tr>
<td>CE21</td>
<td>MTYPE, RDSETUP, RDSTROBE, RDHOLD</td>
</tr>
<tr>
<td>CE31</td>
<td>MTYPE, RDSETUP, RDSTROBE, RDHOLD</td>
</tr>
<tr>
<td>CE02</td>
<td>RDEXHLD, WREXHLD, WRSETUP, WRSTROBE, WRHOLD</td>
</tr>
<tr>
<td>CE12</td>
<td>RDEXHLD, WREXHLD, WRSETUP, WRSTROBE, WRHOLD</td>
</tr>
<tr>
<td>CE22</td>
<td>RDEXHLD, WREXHLD, WRSETUP, WRSTROBE, WRHOLD</td>
</tr>
<tr>
<td>CE32</td>
<td>RDEXHLD, WREXHLD, WRSETUP, WRSTROBE, WRHOLD</td>
</tr>
<tr>
<td>CE03</td>
<td>TIMOUT</td>
</tr>
<tr>
<td>CE13</td>
<td>TIMOUT</td>
</tr>
<tr>
<td>CE23</td>
<td>TIMOUT</td>
</tr>
<tr>
<td>CE33</td>
<td>TIMOUT</td>
</tr>
<tr>
<td>SDC1</td>
<td>TRC, SDSIZE, SDWID, RFEN, TRCD, TRP</td>
</tr>
<tr>
<td>SDPER</td>
<td>PERIOD</td>
</tr>
<tr>
<td>SDCNT</td>
<td>(R)COUNTER</td>
</tr>
<tr>
<td>INIT</td>
<td>INIT</td>
</tr>
<tr>
<td>SDC2</td>
<td>TMRD, TRAS, TACTV2ACTV</td>
</tr>
</tbody>
</table>
### Table 7-3: Registers (Continued)

#### (b) 5502 and 5501 Registers

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

**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 configuration structure used to set up EMIF interface

<table>
<thead>
<tr>
<th>Structure</th>
<th>EMIF_Config</th>
</tr>
</thead>
<tbody>
<tr>
<td>Members</td>
<td></td>
</tr>
<tr>
<td>Uint16 egr</td>
<td>Global Control Register</td>
</tr>
<tr>
<td>Uint16 emir</td>
<td>Global Reset Register</td>
</tr>
<tr>
<td>Uint16 ce01</td>
<td>EMIF CE0 Space Control Register 1</td>
</tr>
<tr>
<td>Uint16 ce02</td>
<td>EMIF CE0 Space Control Register 2</td>
</tr>
<tr>
<td>Uint16 ce03</td>
<td>EMIF CE0 Space Control Register 3</td>
</tr>
<tr>
<td>Uint16 ce11</td>
<td>EMIF CE1 Space Control Register 1</td>
</tr>
<tr>
<td>Uint16 ce12</td>
<td>EMIF CE1 Space Control Register 2</td>
</tr>
<tr>
<td>Uint16 ce13</td>
<td>EMIF CE1 Space Control Register 3</td>
</tr>
<tr>
<td>Uint16 ce21</td>
<td>EMIF CE2 Space Control Register 1</td>
</tr>
<tr>
<td>Uint16 ce22</td>
<td>EMIF CE2 Space Control Register 2</td>
</tr>
<tr>
<td>Uint16 ce23</td>
<td>EMIF CE2 Space Control Register 3</td>
</tr>
<tr>
<td>Uint16 ce31</td>
<td>EMIF CE3 Space Control Register 1</td>
</tr>
<tr>
<td>Uint16 ce32</td>
<td>EMIF CE3 Space Control Register 2</td>
</tr>
<tr>
<td>Uint16 ce33</td>
<td>EMIF CE3 Space Control Register 3</td>
</tr>
<tr>
<td>Uint16 sdc1</td>
<td>EMIF SDRAM Control Register 1</td>
</tr>
<tr>
<td>Uint16 sdper</td>
<td>EMIF SDRAM Period Register</td>
</tr>
<tr>
<td>Uint16 init</td>
<td>EMIF SDRAM Initialization Register</td>
</tr>
<tr>
<td>Uint16 sdc2</td>
<td>EMIF SDRAM Control Register 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Members</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Uint16 gblctl1</td>
<td>EMIF Global Control Register 1</td>
</tr>
<tr>
<td>Uint16 gblctl2</td>
<td>EMIF Global Control Register 2</td>
</tr>
<tr>
<td>Uint16 ce1ctl1</td>
<td>CE1 Space Control Register 1</td>
</tr>
<tr>
<td>Uint16 ce1ctl2</td>
<td>CE1 Space Control Register 2</td>
</tr>
<tr>
<td>Uint16 ce0ctl1</td>
<td>CE0 Space Control Register 1</td>
</tr>
<tr>
<td>Uint16 ce0ctl2</td>
<td>CE0 Space Control Register 2</td>
</tr>
<tr>
<td>Uint16 ce2ctl1</td>
<td>CE2 Space Control Register 1</td>
</tr>
<tr>
<td>Uint16 ce2ctl2</td>
<td>CE2 Space Control Register 2</td>
</tr>
<tr>
<td>Uint16 ce3ctl1</td>
<td>CE3 Space Control Register 1</td>
</tr>
<tr>
<td>Uint16 ce3ctl2</td>
<td>CE3 Space Control Register 2</td>
</tr>
<tr>
<td>Uint16 sdctl1</td>
<td>SDRAM Control Register 1</td>
</tr>
<tr>
<td>Uint16 sdctl2</td>
<td>SDRAM Control Register 2</td>
</tr>
</tbody>
</table>

5502 and 5501 only
**EMIF_Config**

| 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**

```c
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.

**Function**

`void EMIF_config(
    EMIF_Config *Config
);

**Arguments**

- `Config` Pointer to an initialized configuration structure

**Return Value**

None

**Description**

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

**Example**

```c
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);
```
<table>
<thead>
<tr>
<th>Function</th>
<th>EMIF_getConfig</th>
<th>Reads the EMIF configuration structure</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function</strong></td>
<td>void EMIF_getConfig(</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EMIF_Config *Config</td>
<td></td>
</tr>
<tr>
<td></td>
<td>);</td>
<td></td>
</tr>
<tr>
<td><strong>Arguments</strong></td>
<td>Config Pointer to an initialized configuration structure</td>
<td></td>
</tr>
<tr>
<td><strong>Return Value</strong></td>
<td>None</td>
<td></td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Reads the EMIF configuration in a configuration structure.</td>
<td></td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td>EMIF_Config myConfig;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EMIF_getConfig(&amp;myConfig);</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>EMIF_enterselfRefresh</th>
<th>Performs self refresh for SDRAM connected to EMIF (5509A only)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function</strong></td>
<td>void EMIF_enterselfRefresh(</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Uint16 ckePin,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Uint16 tRasDelay</td>
<td></td>
</tr>
<tr>
<td></td>
<td>);</td>
<td></td>
</tr>
<tr>
<td><strong>Arguments</strong></td>
<td>ckePin — selects which pin to use for CKE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ckePin — 0 selects XF pin</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ckePin — 1 selects GPIO.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tRasDelay — number of CPU cycles to hold memory in refresh</td>
<td></td>
</tr>
<tr>
<td><strong>Return Value</strong></td>
<td>None</td>
<td></td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td>EMIF_enterselfRefresh(1,1000);</td>
<td></td>
</tr>
</tbody>
</table>
**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 memory in EMIF CE spaces. Has no effect on EMIF configuration registers. These registers 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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMIF_RGET()</td>
<td>Uint16 EMIF_RGET(REG)</td>
</tr>
<tr>
<td>EMIF_RSET()</td>
<td>Void EMIF_RSET(REG, Uint16 regval)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMIF_FGET()</td>
<td>Uint16 EMIF_FGET(REG, FIELD)</td>
</tr>
<tr>
<td>EMIF_FSET()</td>
<td>Void EMIF_FSET(REG, FIELD, Uint16 fieldval)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMIF_REG_RMK()</td>
<td>Uint16 EMIF_REG_RMK(fieldval_n,…fieldval_0)</td>
</tr>
<tr>
<td>Note:</td>
<td>*Start with field values with most significant field positions:</td>
</tr>
<tr>
<td></td>
<td>field_n: MSB field</td>
</tr>
<tr>
<td></td>
<td>field_0: LSB field</td>
</tr>
<tr>
<td></td>
<td>*only writable fields allowed</td>
</tr>
<tr>
<td>EMIF_FMK()</td>
<td>Uint16 EMIF_FMK(REG, FIELD, fieldval)</td>
</tr>
</tbody>
</table>

(d) Macros to read a register address

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMIF_ADDR()</td>
<td>Uint16 EMIF_ADDR(REG)</td>
</tr>
</tbody>
</table>

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.
3) For 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).
7) 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.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1 Overview</td>
<td>8-2</td>
</tr>
<tr>
<td>8.2 Configuration Structure</td>
<td>8-4</td>
</tr>
<tr>
<td>8.3 Functions</td>
<td>8-5</td>
</tr>
<tr>
<td>8.4 Macros</td>
<td>8-17</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
<th>See page</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPIO_pinDirection</td>
<td>Sets the GPIO pins as either an input or output pin</td>
<td>8-8</td>
</tr>
<tr>
<td>GPIO_pinDisable</td>
<td>Disables a pin as a GPIO pin</td>
<td>8-13</td>
</tr>
<tr>
<td>GPIO_pinEnable</td>
<td>Enables a pin as a GPIO pin</td>
<td>8-13</td>
</tr>
<tr>
<td>GPIO_pinRead</td>
<td>Reads the GPIO pin value</td>
<td>8-14</td>
</tr>
<tr>
<td>GPIO_pinWrite</td>
<td>Writes a value to a GPIO pin</td>
<td>8-15</td>
</tr>
</tbody>
</table>

The following functions are supported by C5502 and C5501.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
<th>See page</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPIO_close</td>
<td>Frees one or more GPIO pins for use</td>
<td>8-5</td>
</tr>
<tr>
<td>GPIO_config</td>
<td>Configures GPIO pins</td>
<td>8-7</td>
</tr>
<tr>
<td>GPIO_open</td>
<td>Allocates one or more GPIO pins to the current process</td>
<td>8-5</td>
</tr>
<tr>
<td>GPIO_pinReadAll</td>
<td>Reads the value of one or more pins</td>
<td>8-14</td>
</tr>
<tr>
<td>GPIO_pinWriteAll</td>
<td>Writes the value to one or more pins</td>
<td>8-15</td>
</tr>
<tr>
<td>GPIO_pinReset</td>
<td>Resets the value of one or more pins</td>
<td>8-16</td>
</tr>
</tbody>
</table>
### Table 8-2. GPIO Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>IODIR</td>
<td>IO7DIR, IO6DIR, IO5DIR, IO4DIR, IO3DIR, IO2DIR, IO1DIR, IO0DIR</td>
</tr>
<tr>
<td>IODATA</td>
<td>IO7D, IO6D, IO5D, IO4D, IO3D, IO2D, IO1D, IO0D</td>
</tr>
</tbody>
</table>

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 and C5501.

- **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, IO21DAT, 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 and C5501.

**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
**GPIO_open**

**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 and C5501, 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**

```c
GPIO_Handle hGPIO;

hGPIO = GPIO_open(GPIO_PGPIO_PIN1, 0);
```
### 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**

```c
GPIO_Handle hGpio;
GPIO_Config myConfig = {GPIO_PIN1_OUTPUT |
                        GPIO_PIN3_OUTPUT};

configuration for 5502 and 5501

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

### GPIO_configAll

**Function**

void GPIO_config(GPIO_ConfigAll &gCfg);

**Arguments**

- **gCfg** Configuration structure for both power and non-power, non-muxed GPIO pins.

**Return Value**

None

**Description**

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

**Example**

```c
GPIO_ConfigAll gCfg = {
  GPIO_PIN1_OUTPUT | GPIO_PIN3_OUTPUT, /* IODIR */
  0, /* PGPIEN0 */
  0, /* PGPIODIR0 */
  0, /* PGPIEN1 */
  0, /* PGPIODIR1 */
  0, /* PGPIEN2 */
  0, /* PGPIODIR2 */
};

/* GPIO configuration for 5502 and 5501 */
GPIO_configAll(&gCfg);
```
GPIO_pinDirection

Sets the GPIO pin as either an input or output pin

Function
For C5502 and 5501:
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 and C5501 CSL)
pinMask GPIO pins affected by direction

For 5502 and 5501, 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 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 (Parallel 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_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 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_pinDirection

GPIO_PGPIO_PIN27_INPUT
GPIO_PGPIO_PIN28_INPUT
GPIO_PGPIO_PIN29_INPUT
GPIO_PGPIO_PIN30_INPUT
GPIO_PGPIO_PIN31_INPUT

GPIO Pin Group 3 (Parallel 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
**GPIO_pinEnable**

**Description**
Sets the direction for one or 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**

**Function**
For C5502 and 5501:
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
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**

**Function**
For C5502 and 5501:
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 and C5501 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 and C5501:
```c
int GPIO_pinRead(GPIO_Handle hGpio, Uint32 pinId)
```
For C5509/C5509A/C5510
```c
int GPIO_pinRead(Uint32 pinId)
```

**Arguments**  
- `hGpio` GPIO Handle returned from previous call to GPIO_open().  
  (This argument is only for C5502 and C5501 CSL)
- `pinId` IDs of the GPIO pins to read.

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

**Description**  
Reads the value read in a general purpose input pin.

**Example**  
```c
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 and C5501:
```c
int GPIO_pinReadAll(GPIO_Handle hGpio, Uint32 pinMask)
```
For C5509/C5509A/C5510
```c
int GPIO_pinReadAll(Uint32 pinMask)
```

**Arguments**  
- `hGpio` GPIO Handle returned from previous call to GPIO_open().  
  (This argument is only for C5502 and C5501 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**  
```c
int val;
/* reads IO0 and IO7 pin values */
val=GPIO_pinReadAll (hGpio,GPIO_GPIO_PIN0 | GPIO_GPIO_PIN7);
```
**GPIO_pinWrite**  
*Writes a value to a GPIO pin*

**Function**
For C5502 and C5501:
```c
void GPIO_pinWrite(GPIO_Handle hGpio,
                    Uint32 pinMask,
                    Uint16 val)
```
For C5509/C5509A/C5510:
```c
void GPIO_pinWrite(Uint32 pinMask, Uint16 val)
```

**Arguments**
- `hGpio`  
  GPIO Handle returned from previous call to GPIO_open().  
  (This argument is only for C5502 and C5501 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**
```c
/* 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 and C5501:
```c
void GPIO_pinWriteAll(GPIO_Handle hGpio,
                      Uint32 pinMask,
                      Uint16 val)
```
For C5509/C5509A/C5510:
```c
void GPIO_pinWriteAll(Uint32 pinMask, Uint16 val)
```

**Arguments**
- `hGpio`  
  GPIO Handle returned from previous call to GPIO_open().  
  (This argument is only for C5502 and C5501 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. This function assumes an in-place value mask for writing to the GPIO pins. It will not left-justify values.

**Example**
```c
/* 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

<table>
<thead>
<tr>
<th>Function</th>
<th>void GPIO_pinReset(GPIO_Handle hGpio, Uint32 pinMask)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td>hGpio: GPIO Handle returned from previous call to GPIO_open().</td>
</tr>
<tr>
<td></td>
<td>pinMask: ID of one or more GPIO pins to write. Please see GPIO_pinDirection for list of valid pin IDs.</td>
</tr>
<tr>
<td>Return Value</td>
<td>None</td>
</tr>
<tr>
<td>Description</td>
<td>Restores selected GPIO pins to default value of 0.</td>
</tr>
</tbody>
</table>
| 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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPIO_RGET()</td>
<td>Uint16 GPIO_RGET(REG)</td>
</tr>
<tr>
<td>GPIO_RSET()</td>
<td>Void GPIO_RSET(REG, Uint16 regval)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPIO_FGET()</td>
<td>Uint16 GPIO_FGET(REG, FIELD)</td>
</tr>
<tr>
<td>GPIO_FSET()</td>
<td>Void GPIO_FSET(REG, FIELD, Uint16 fieldval)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPIO_REG_RMK()</td>
<td>Uint16 GPIO_REG_RMK(fieldval_n, …fieldval_0)</td>
</tr>
<tr>
<td></td>
<td>Note: *Start with field values with most significant field positions:</td>
</tr>
<tr>
<td></td>
<td>field_n: MSB field</td>
</tr>
<tr>
<td></td>
<td>field_0: LSB field</td>
</tr>
<tr>
<td></td>
<td>*only writable fields allowed</td>
</tr>
<tr>
<td>GPIO_FMK()</td>
<td>Uint16 GPIO_FMK(REG, FIELD, fieldval)</td>
</tr>
</tbody>
</table>

(d) Macros to read a register address

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPIO_ADDR()</td>
<td>Uint16 GPIO_ADDR(REG)</td>
</tr>
</tbody>
</table>

Notes:

1) REG include the registers IODIR, IODATA, GPIODIR, GPIODATA, GPIOEN, AGIODIR, AGIODATA, 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).
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 and C5501 devices.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1 Overview</td>
<td>9-2</td>
</tr>
<tr>
<td>9.2 Configuration Structures</td>
<td>9-4</td>
</tr>
<tr>
<td>9.3 Functions</td>
<td>9-5</td>
</tr>
<tr>
<td>9.4 Macros</td>
<td>9-6</td>
</tr>
</tbody>
</table>
9.1 Overview

This module enables configuration of the 5502 and 5501 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

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
<th>See page ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>HPI_Config</td>
<td>HPI module configuration structure</td>
<td>9-4</td>
</tr>
</tbody>
</table>

Table 9–2. HPI Functions

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
<th>See page ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>HPI_config()</td>
<td>Sets up HPI using configuration structure (HPI_Config)</td>
<td>9-5</td>
</tr>
<tr>
<td>HPI_getConfig()</td>
<td>Returns current HPI control register values in a configuration structure (HPI_Config)</td>
<td>9-5</td>
</tr>
</tbody>
</table>

Table 9–3. HPI Registers and Bit Field Names

<table>
<thead>
<tr>
<th>Register</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>HGPIOEN</td>
<td>EN0, EN1, EN2, EN4, EN6, EN7, EN8, EN9, EN11, EN12</td>
</tr>
<tr>
<td>HGPIODIR</td>
<td>HDn(n=0–15)</td>
</tr>
<tr>
<td>HGPIODAT</td>
<td>HDn(n=0–15)</td>
</tr>
<tr>
<td>HPIC</td>
<td>HPIASEL, DUALHPIA, BOBSTAT, HPIRST, FETCH, HRDY, HINT, DSPINT, BOB</td>
</tr>
<tr>
<td>HPIAW</td>
<td>HPIAW</td>
</tr>
<tr>
<td>HPIAR</td>
<td>HPIAR</td>
</tr>
<tr>
<td>HPWREMU</td>
<td>FREE, SOFT</td>
</tr>
</tbody>
</table>
Table 9–4. HPI Macros

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
<th>See page</th>
</tr>
</thead>
<tbody>
<tr>
<td>HPI_ADDR</td>
<td>Get the address of a given register</td>
<td>9-6</td>
</tr>
<tr>
<td>HPI_FGET</td>
<td>Gets value of a register field</td>
<td>9-6</td>
</tr>
<tr>
<td>HPI_FMK</td>
<td>Creates register value based on individual field value</td>
<td>9-7</td>
</tr>
<tr>
<td>HPI_FSET</td>
<td>Sets value of register field</td>
<td>9-7</td>
</tr>
<tr>
<td>HPI_REG_RMK</td>
<td>Creates register value based on individual field values</td>
<td>9-8</td>
</tr>
<tr>
<td>HPI_RGET</td>
<td>Gets the value of an HPI register</td>
<td>9-9</td>
</tr>
<tr>
<td>HPI_RSET</td>
<td>Set the value of an HPI register</td>
<td>9-9</td>
</tr>
</tbody>
</table>
### 9.2 Configuration Structures

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

<table>
<thead>
<tr>
<th>Structure</th>
<th>HPI_Config</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Members</strong></td>
<td></td>
</tr>
<tr>
<td>Uint16 hpwremu</td>
<td>HPI power/emulation management register</td>
</tr>
<tr>
<td>Uint16 hpioen</td>
<td>HPI GPIO pin enable register</td>
</tr>
<tr>
<td>Uint16 hpgiodir</td>
<td>HPI GPIO pin direction register</td>
</tr>
<tr>
<td>Uint16 hpic</td>
<td>HPI Control register</td>
</tr>
</tbody>
</table>
9.3 Functions

The following are functions available for the HPI module.

### HPI_config
*Writes to HPI registers using values in configuration structure*

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

**Arguments**
- `myConfig` Pointer to an initialized configuration structure

**Return Value**
None

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

**Example**
```c
HPI_Config myConfig;
HPI_getConfig(&myConfig);
```
9.4 Macros

The following is a listing of HPI macros.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>HPI_ADDR</td>
<td>Gets address of given register</td>
<td>ioprt Uint16 *hpi_ctl;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hpi_ctl = HPI_ADDR(HPIC);</td>
</tr>
<tr>
<td>HPI_FGET</td>
<td>Gets the value of register field</td>
<td>Uint16 bob = HPI_FGET(HPIC,BOB);</td>
</tr>
</tbody>
</table>
## 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 shown below.

### Example

```c
uint16 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

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

**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 (0,1,1,0,0,0,0,0,0);

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.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1 Overview</td>
<td>10-2</td>
</tr>
<tr>
<td>10.2 Configuration Structures</td>
<td>10-5</td>
</tr>
<tr>
<td>10.3 Functions</td>
<td>10-7</td>
</tr>
<tr>
<td>10.4 Macros</td>
<td>10-17</td>
</tr>
<tr>
<td>10.5 Examples</td>
<td>10-18</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Configuration Structure</th>
<th>Description</th>
<th>See page…</th>
</tr>
</thead>
<tbody>
<tr>
<td>I2C_Config</td>
<td>I2C configuration structure used to set up the I2C (register-based)</td>
<td>10-5</td>
</tr>
<tr>
<td>I2C_Setup</td>
<td>Sets up the I2C using the initialization structure</td>
<td>10-6</td>
</tr>
</tbody>
</table>

### Table 10–2. I2C Functions

<table>
<thead>
<tr>
<th>Functions</th>
<th>Description</th>
<th>See page…</th>
</tr>
</thead>
<tbody>
<tr>
<td>I2C_config()</td>
<td>Sets up the I2C using the configuration structure</td>
<td>10-7</td>
</tr>
<tr>
<td>I2C_eventDisable()</td>
<td>Disables the I2C interrupt specified.</td>
<td>10-8</td>
</tr>
<tr>
<td>I2C_eventEnable()</td>
<td>Enables the I2C interrupt specified.</td>
<td>10-8</td>
</tr>
<tr>
<td>I2C_getConfig()</td>
<td>Obtains the current configuration of all the I2C registers</td>
<td>10-8</td>
</tr>
<tr>
<td>I2C_getEventId()</td>
<td>Returns the I2C IRQ event ID</td>
<td>10-9</td>
</tr>
<tr>
<td>I2C_setup()</td>
<td>Sets up the I2C using the initialization structure</td>
<td>10-9</td>
</tr>
<tr>
<td>I2C_IsrAddr</td>
<td>I2C structure containing pointers to functions that will be executed when a specific I2C interrupt is enabled and received.</td>
<td>10-10</td>
</tr>
</tbody>
</table>
### Table 10–2. I2C Functions (Continued)

<table>
<thead>
<tr>
<th>Functions</th>
<th>Description</th>
<th>See page…</th>
</tr>
</thead>
<tbody>
<tr>
<td>I2C_read()</td>
<td>Performs master/slave receiver functions</td>
<td>10-10</td>
</tr>
<tr>
<td>I2C_readByte()</td>
<td>Performs a read from the data receive register (I2CDRR).</td>
<td>10-11</td>
</tr>
<tr>
<td>I2C_reset()</td>
<td>Sets the IRS bit in the I2CMRD register to 1 (performs a reset).</td>
<td>10-12</td>
</tr>
<tr>
<td>I2C_rfull()</td>
<td>Reads the RSFULL bit in the I2CSTR register.</td>
<td>10-12</td>
</tr>
<tr>
<td>I2C_rdy()</td>
<td>Reads the I2CRRDY bit in the I2CSTR register.</td>
<td>10-12</td>
</tr>
<tr>
<td>I2C_sendStop()</td>
<td>Sets the STP bit in the I2CMRD register (generates a stop).</td>
<td>10-13</td>
</tr>
<tr>
<td>I2C_setCallback()</td>
<td>Associates each callback function to one of the I2C interrupt events and installs the I2C dispatcher table.</td>
<td>10-13</td>
</tr>
<tr>
<td>I2C_start()</td>
<td>Sets the STT bit in the I2CMRD register (generates a start).</td>
<td>10-14</td>
</tr>
<tr>
<td>I2C_write()</td>
<td>Performs master/slave transmitter functions</td>
<td>10-14</td>
</tr>
<tr>
<td>I2C_writeByte()</td>
<td>Performs a write to the data transmit register (I2CDXR).</td>
<td>10-15</td>
</tr>
<tr>
<td>I2C_xempty()</td>
<td>Reads the XSMT bit in the I2CSTR register.</td>
<td>10-16</td>
</tr>
<tr>
<td>I2C_xrdy()</td>
<td>Reads the I2CXRDY bit in the I2CSTR register.</td>
<td>10-16</td>
</tr>
</tbody>
</table>
### 10.1.1 I2C Registers

**Table 10-3. I2C Registers**

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

**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**

<table>
<thead>
<tr>
<th>Structure</th>
<th>I2C_Config</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Members</strong></td>
<td></td>
</tr>
<tr>
<td>Uint16 i2coar</td>
<td>Own address register</td>
</tr>
<tr>
<td>Uint16 i2cier</td>
<td>Interrupt mask/status register</td>
</tr>
<tr>
<td>Uint16 i2cstr</td>
<td>Interrupt status register</td>
</tr>
<tr>
<td>Uint16 i2cclkL</td>
<td>Clock Divider Low register</td>
</tr>
<tr>
<td>Uint16 i2cclkH</td>
<td>Clock Divider High register</td>
</tr>
<tr>
<td>Uint16 i2ccnt</td>
<td>Data Count register</td>
</tr>
<tr>
<td>Uint16 i2csar</td>
<td>Slave Address register</td>
</tr>
<tr>
<td>Uint16 i2cmdr</td>
<td>Mode register</td>
</tr>
<tr>
<td>Uint16 i2cisrc</td>
<td>Interrupt source vector register</td>
</tr>
<tr>
<td>Uint16 i2cpsc</td>
<td>Prescaler register</td>
</tr>
</tbody>
</table>

**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**

```c
I2C_Config Config = {
  0xFFFF,    /* I2COAR */
  0x0000,    /* I2CIER */
  0xFFFF,    /* I2CSTR */
  10,        /* I2CCLKL */
  8,         /* I2CCLKH */
  1,         /* I2CCNT */
  0xFFFF,    /* 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:
  - Value | Bits/byte transmitted/received
  - 0   | 8
  - 1   | 1
  - 2   | 2
  - 3   | 3
  - 4   | 4
  - 5   | 5
  - 6   | 6
  - 7   | 7
- **Uint16 db**  
  Data Loopback mode
  - 0 = off, 1 = on
- **Uint16 free**  
  emulator FREE mode
  - 0 = off, 1 = on

**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**

```c
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

<table>
<thead>
<tr>
<th>Function</th>
<th>void I2C_config (I2C_Config *Config);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td>Config Pointer to an initialized configuration structure</td>
</tr>
<tr>
<td>Return Value</td>
<td>none</td>
</tr>
</tbody>
</table>
| 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,   /* I2CCCLKL */  
8,    /* I2CCCLKH */  
1,    /* I2CCNT */  
0xFFFF, /* I2CSAR */  
0x0664, /* I2CMODR */  
0xFFFF, /* I2CSR */  
0x0000 /* I2CPSC */  
};  
I2C_config(&Config); |
I2C_eventDisable

Disables the interrupt specified by the isrMask

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 isrMask.

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 structure

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

<table>
<thead>
<tr>
<th>Structure</th>
<th>I2C_IsrAddr</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Members</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>void (*alAddr)(void);</td>
<td>pointer to function for AL interrupt</td>
</tr>
<tr>
<td>void (*nackAddr)(void);</td>
<td>pointer to function for NACK interrupt</td>
</tr>
<tr>
<td>void (*ardyAddr)(void);</td>
<td>pointer to function for ARDY interrupt</td>
</tr>
<tr>
<td>void (*rrdyAddr)(void);</td>
<td>pointer to function for RRDY interrupt</td>
</tr>
<tr>
<td>void (*xrdyAddr)(void);</td>
<td>pointer to function for XRDY interrupt</td>
</tr>
</tbody>
</table>

| 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. |

<table>
<thead>
<tr>
<th>Example</th>
<th>I2C_IsrAddr addr = {</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>myALIsr,</td>
</tr>
<tr>
<td></td>
<td>myNACKIsr,</td>
</tr>
<tr>
<td></td>
<td>myARDYIsr,</td>
</tr>
<tr>
<td></td>
<td>myRRDYIsr,</td>
</tr>
<tr>
<td></td>
<td>myXRDYIsr</td>
</tr>
<tr>
<td></td>
<td>};</td>
</tr>
</tbody>
</table>

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); |

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Uint16 *data Pointer to data array</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>int length length of data to be received</td>
</tr>
<tr>
<td></td>
<td>int master master mode: 0 = slave, 1 = master</td>
</tr>
<tr>
<td></td>
<td>Uint16 slaveaddress Slave address to receive from</td>
</tr>
<tr>
<td></td>
<td>int transfermode Transfer mode of operation (SADP, SAD, etc.)</td>
</tr>
<tr>
<td></td>
<td>Value  Transfer Mode</td>
</tr>
<tr>
<td></td>
<td>1       S−A−D..(n)..D−P</td>
</tr>
<tr>
<td></td>
<td>2       S−A−D..(n)..D (repeat n times)</td>
</tr>
<tr>
<td></td>
<td>3       S−A−D−D−D..... (continuous)</td>
</tr>
<tr>
<td></td>
<td>int timeout Timeout for bus busy, no acknowledge, transmit ready</td>
</tr>
<tr>
<td></td>
<td>int checkbus flag used to check if bus is busy. Typically, it must be set to 1, except under special I2C program conditions.)</td>
</tr>
</tbody>
</table>
I2C_readByte

**Return Value**

<table>
<thead>
<tr>
<th>Value returned</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No errors</td>
</tr>
<tr>
<td>1</td>
<td>Bus busy; not able to generate start condition</td>
</tr>
<tr>
<td>2</td>
<td>Timeout for transmit ready (first byte)</td>
</tr>
<tr>
<td>4</td>
<td>Timeout for transmit ready (within main loop)</td>
</tr>
</tbody>
</table>

**Description**

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

**Example**

```c
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**

```c
Uint16 I2C_readByte();
```

**Arguments**

None

**Return Value**

Data read for an I2C receive port.
### I2C_readByte

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

**Example**

```c
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

**Function**
```c
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

**Function**
```c
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

**Function**
```c
Uint16 I2C_rrdy();
```

**Arguments**
None
### I2C_readByte

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

I2C_setCallback(&addr);
```
**I2C_readByte**

**Function**  
`void I2C_readByte();`

**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:

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

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

**Example**  
`I2C_start();`

---

**I2C_write**

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

**Arguments**  
<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uint16 *data</td>
<td>Pointer to data array</td>
</tr>
<tr>
<td>int length</td>
<td>length of data to be transmitted</td>
</tr>
<tr>
<td>int master</td>
<td>master mode: 0 = slave, 1 = master</td>
</tr>
<tr>
<td>Uint16 slaveaddress</td>
<td>Slave address to transmit to</td>
</tr>
<tr>
<td>int transfermode</td>
<td>Transfer mode of operation (SADP, SAD, etc.)</td>
</tr>
<tr>
<td>Value</td>
<td>Transfer Mode</td>
</tr>
<tr>
<td>1</td>
<td>S–A–D–(n)–D P</td>
</tr>
<tr>
<td>2</td>
<td>S–A–D–(n)–D (repeat n times)</td>
</tr>
<tr>
<td>3</td>
<td>S–A–D–D–D..... (continuous)</td>
</tr>
</tbody>
</table>

**Return Value**  
<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No errors</td>
</tr>
<tr>
<td>1</td>
<td>Bus busy; not able to generate start condition</td>
</tr>
<tr>
<td>2</td>
<td>Timeout for transmit ready (first byte)</td>
</tr>
<tr>
<td>3</td>
<td>NACK (No--acknowledge) received</td>
</tr>
<tr>
<td>4</td>
<td>Timeout for transmit ready (within main loop)</td>
</tr>
<tr>
<td>5</td>
<td>NACK (No--acknowledge) received (last byte)</td>
</tr>
</tbody>
</table>
**I2C_readByte**

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

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

**Example**
```c
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);
```

```c
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**

**Function**
```c
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**
```c
I2C_writeByte(0x34);
```
**I2C_readByte**

**I2C_xempty**  
*Reads an XMST bit from an I2CST register*

**Function**

```c
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**

```c
if (I2C_xempty()) {
    ...
}
```

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

**Function**

```c
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**

```c
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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>I2C_RGET()</td>
<td>Uint16 I2C_RGET(REG)</td>
</tr>
<tr>
<td>I2C_RSET()</td>
<td>Void I2C_RSET(REG, Uint16 regval)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>I2C_FGET()</td>
<td>Uint16 I2C_FGET(REG, FIELD)</td>
</tr>
<tr>
<td>I2C_FSET()</td>
<td>Void I2C_FSET(REG, FIELD, Uint16 fieldval)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>I2C_REG_RMK()</td>
<td>Uint16 I2C_REG_RMK(fieldval_n,...,fieldval_0)</td>
</tr>
<tr>
<td><strong>Note:</strong></td>
<td>*Start with field values with most significant field positions: field_n: MSB field</td>
</tr>
<tr>
<td></td>
<td>field_0: LSB field</td>
</tr>
<tr>
<td></td>
<td>*only writable fields allowed</td>
</tr>
<tr>
<td>I2C_FMK()</td>
<td>Uint16 I2C_FMK(REG, FIELD, fieldval)</td>
</tr>
</tbody>
</table>

(d) Macros to read a register address

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>I2C_ADDR()</td>
<td>Uint16 I2C_ADDR(REG)</td>
</tr>
</tbody>
</table>
Notes:  1) REG indicates the registers: I2COAR, I2CIMR, I2CSTR, I2CCLKL, I2CCLKH, 
I2CDRR, I2CCSAR, I2CSAR, I2CDXR, I2CMDR, I2CSRC, I2CPSC.

2) FIELD indicates the register field name.
   a) For REG_FSET and REG_FMK, FIELD must be a writable field.
   b) 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.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.1 Overview</td>
<td>11-2</td>
</tr>
<tr>
<td>11.2 Configuration Structures</td>
<td>11-3</td>
</tr>
<tr>
<td>11.3 Functions</td>
<td>11-5</td>
</tr>
<tr>
<td>11.4 Macros</td>
<td>11-8</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Structure</th>
<th>Purpose</th>
<th>See page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICACHE_Config</td>
<td>ICACHE configuration structure used to setup the Instruction Cache</td>
<td>11-3</td>
</tr>
<tr>
<td>ICACHE_Setup</td>
<td>ICACHE Configuration structure used to enable the Instruction Cache.</td>
<td>11-4</td>
</tr>
<tr>
<td>ICACHE_TagSet</td>
<td>ICACHE structure used to set the tag registers.</td>
<td>11-4</td>
</tr>
</tbody>
</table>

Table 11–2. ICACHE Functions

<table>
<thead>
<tr>
<th>Structure</th>
<th>Purpose</th>
<th>See page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICACHE_config</td>
<td>Sets up the ICACHE register using the configuration structure</td>
<td>11-5</td>
</tr>
<tr>
<td>ICACHE_disable</td>
<td>Resets the Cache Enable bit in status register 3</td>
<td>11-5</td>
</tr>
<tr>
<td>ICACHE_enable</td>
<td>Sets the Cache Enable bit in status register 3</td>
<td>11-6</td>
</tr>
<tr>
<td>ICACHE_flush</td>
<td>Sets the Cache Flush bit in status register 3</td>
<td>11-6</td>
</tr>
<tr>
<td>ICACHE_freeze</td>
<td>Sets the Cache Freeze bit in status register 3</td>
<td>11-6</td>
</tr>
<tr>
<td>ICACHE_setup</td>
<td>Configures the ICACHE and enables it</td>
<td>11-7</td>
</tr>
<tr>
<td>ICACHE_tagset</td>
<td>Sets the values of the Ramset Tags</td>
<td>11-7</td>
</tr>
<tr>
<td>ICACHE_unfreeze</td>
<td>Resets the Cache Freeze bit in status register 3</td>
<td>11-7</td>
</tr>
</tbody>
</table>
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**

**Members**

- Uint16 icgc  
  Global Control Register
- Uint16 icwc  
  N-way Control Register (not supported on C5502/5501)
- Uint16 icrc1  
  Ramset 1 Control Register (not supported on C5502/5501)
- Uint16 icrtag1  
  Ramset 1 Tag Register (not supported on C5502/5501)
- Uint16 icrc2  
  Ramset 2 Control Register (not supported on C5502/5501)
- Uint16 icrtag2  
  Ramset 2 Tag Register (not supported on C5502/5501)

**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**

```c
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 and C5501**

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

**Structure used to configure and enable the ICACHE**

**Structure**

ICACHE_Setup

**Members**

- **rmode** (Uint16)
- **r1addr** (Uint32)
- **r2addr** (Uint32)

**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**

```c
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**

- **r1addr** (Uint32)
- **r2addr** (Uint32)

**Description**

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

**Example**

```c
ICACHE_Tagset Mytagset = {
    0x500000,
    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**

```c
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 preferred since this function will also enable the ICACHE.

**Example**

```c
ICACHE_enable();
```

### ICACHE_flush

Sets the ICACHE flush bit in the Status Register 3

**Function**

```c
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**

```c
ICACHE_flush();
```

### ICACHE_freeze

Sets the ICACHE freeze bit in the Status Register 3

**Function**

```c
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**

```c
ICACHE_freeze();
```
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICACHE_setup</td>
<td>Configures the ICACHE and enables it</td>
</tr>
<tr>
<td>Function</td>
<td>void ICACHE_setup(ICACHE_Setup *setup);</td>
</tr>
<tr>
<td>Arguments</td>
<td>setup Pointer to an initialized setup structure</td>
</tr>
<tr>
<td>Return Value</td>
<td>None</td>
</tr>
<tr>
<td>Description</td>
<td>Sets the Ramset Mode and enables the ICACHE</td>
</tr>
<tr>
<td>Example</td>
<td>ICACHE_Setup mySetup = { };</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>ICACHE_setup (&amp;mySetup);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Sets the address in the Ramset Tag registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>void ICACHE_tagset(ICACHE_Tagset *params);</td>
</tr>
<tr>
<td>Arguments</td>
<td>params Pointer to an initialized tagset structure</td>
</tr>
<tr>
<td>Return Value</td>
<td>None</td>
</tr>
<tr>
<td>Description</td>
<td>Function sets the addresses in the Ramset Tag registers. This function is</td>
</tr>
<tr>
<td></td>
<td>useful when the user wants to change the Ramset addresses after the ICACHE</td>
</tr>
<tr>
<td></td>
<td>had been flushed.</td>
</tr>
<tr>
<td>Example</td>
<td>ICACHE_Tagset mySetup = { };</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>ICACHE_tagset(&amp;mySetup);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Resets the ICACHE freeze bit in the Status Register 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>void ICACHE_unfreeze();</td>
</tr>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Return Value</td>
<td>None</td>
</tr>
<tr>
<td>Description</td>
<td>Function resets the ICACHE freeze bit in the Status Register 3 the content</td>
</tr>
<tr>
<td></td>
<td>of the ICACHE is unfrozen.</td>
</tr>
<tr>
<td>Example</td>
<td>ICACHE_unfreeze();</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICACHE_RGET()</td>
<td>UInt16 ICACHE_RGET(REG)</td>
</tr>
<tr>
<td>ICACHE_RSET()</td>
<td>void ICACHE_RSET(REG, UInt16 regval)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICACHE_FGET()</td>
<td>UInt16 ICACHE_FGET(REG, FIELD)</td>
</tr>
<tr>
<td>ICACHE_FSET()</td>
<td>void ICACHE_FSET(REG, FIELD, UInt16 fieldval)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICACHE_REG_RMK()</td>
<td>UInt16 ICACHE_REG_RMK(fieldval_n,...fieldval_0)</td>
</tr>
</tbody>
</table>
| 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, fieldval) |

(d) Macros to read a register address

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICACHE_ADDR()</td>
<td>UInt16 ICACHE_ADDR(REG)</td>
</tr>
</tbody>
</table>

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)
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.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.1 Overview</td>
<td>12-2</td>
</tr>
<tr>
<td>12.2 Using Interrupts with CSL</td>
<td>12-7</td>
</tr>
<tr>
<td>12.3 Configuration Structures</td>
<td>12-8</td>
</tr>
<tr>
<td>12.4 Functions</td>
<td>12-9</td>
</tr>
</tbody>
</table>
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 IMRx register.
- Polling for the interrupt status from the IFRx 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>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
<th>See page</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRQ_Config</td>
<td>IRQ structure that contains all local registers required to set up a specific IRQ channel.</td>
<td>12-8</td>
</tr>
</tbody>
</table>
Table 12–2. **IRQ Functions**

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
<th>See page</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRQ_clear()</td>
<td>Clears the interrupt flag in the IFR0/1 registers for the specified event.</td>
<td>12-9</td>
</tr>
<tr>
<td>IRQ_config()†</td>
<td>Updates the DSP/BIOS dispatch table with a new configuration for the specified event.</td>
<td>12-9</td>
</tr>
<tr>
<td>IRQ_disable()</td>
<td>Disables the specified event in the IMR0/1 registers.</td>
<td>12-10</td>
</tr>
<tr>
<td>IRQ_enable()</td>
<td>Enables the specified event in the IMR0/1 register flags.</td>
<td>12-10</td>
</tr>
<tr>
<td>IRQ_getArg()†</td>
<td>Returns value of the argument to the interrupt service routine that the DSP/BIOS dispatcher passes when the interrupt occurs.</td>
<td>12-10</td>
</tr>
<tr>
<td>IRQ_getConfig()†</td>
<td>Returns current DSP/BIOS dispatch table entries for the specified event.</td>
<td>12-11</td>
</tr>
<tr>
<td>IRQ_globalDisable()</td>
<td>Globally disables all maskable interrupts. (INTM = 1)</td>
<td>12-11</td>
</tr>
<tr>
<td>IRQ_globalEnable()</td>
<td>Globally enables all maskable interrupts. (INTM = 0)</td>
<td>12-12</td>
</tr>
<tr>
<td>IRQ_globalRestore()</td>
<td>Restores the status of global interrupt enable/disable (INTM).</td>
<td>12-12</td>
</tr>
<tr>
<td>IRQ_map()†</td>
<td>Maps a logical event to its physical interrupt.</td>
<td>12-13</td>
</tr>
<tr>
<td>IRQ_plug()</td>
<td>Writes the necessary code in the interrupt vector location to branch to the interrupt service routine for the specified event.</td>
<td>12-13</td>
</tr>
<tr>
<td><strong>Caution:</strong> Do not use this function if the DSP/BIOS HWI module or the DSP/BIOS dispatcher are in use.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IRQ_restore()</td>
<td>Restores the status of the specified event in the IMR0/1 register.</td>
<td>12-14</td>
</tr>
<tr>
<td>IRQ_setArg()†</td>
<td>Sets the value of the argument for DSP/BIOS dispatch to pass to the interrupt service routine for the specified event.</td>
<td>12-14</td>
</tr>
<tr>
<td>IRQ_setVecs()</td>
<td>Sets the base address of the interrupt vector table.</td>
<td>12-15</td>
</tr>
<tr>
<td>IRQ_test()</td>
<td>Polls the interrupt flag in IFR register the specified event.</td>
<td>12-15</td>
</tr>
</tbody>
</table>

**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**

<table>
<thead>
<tr>
<th>Constant</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRQ_EVT_RS</td>
<td>Reset</td>
</tr>
<tr>
<td>IRQ_EVT_SINTR</td>
<td>Software Interrupt</td>
</tr>
<tr>
<td>IRQ_EVT_NMI</td>
<td>Non-Maskable Interrupt (NMI)</td>
</tr>
<tr>
<td>IRQ_EVT_SINT16</td>
<td>Software Interrupt #16</td>
</tr>
<tr>
<td>IRQ_EVT_SINT17</td>
<td>Software Interrupt #17</td>
</tr>
<tr>
<td>IRQ_EVT_SINT18</td>
<td>Software Interrupt #18</td>
</tr>
<tr>
<td>IRQ_EVT_SINT19</td>
<td>Software Interrupt #19</td>
</tr>
<tr>
<td>IRQ_EVT_SINT20</td>
<td>Software Interrupt #20</td>
</tr>
<tr>
<td>IRQ_EVT_SINT21</td>
<td>Software Interrupt #21</td>
</tr>
<tr>
<td>IRQ_EVT_SINT22</td>
<td>Software Interrupt #22</td>
</tr>
<tr>
<td>IRQ_EVT_SINT23</td>
<td>Software Interrupt #23</td>
</tr>
<tr>
<td>IRQ_EVT_SINT24</td>
<td>Software Interrupt #24</td>
</tr>
<tr>
<td>IRQ_EVT_SINT25</td>
<td>Software Interrupt #25</td>
</tr>
<tr>
<td>IRQ_EVT_SINT26</td>
<td>Software Interrupt #26</td>
</tr>
<tr>
<td>IRQ_EVT_SINT27</td>
<td>Software Interrupt #27</td>
</tr>
<tr>
<td>IRQ_EVT_SINT28</td>
<td>Software Interrupt #28</td>
</tr>
<tr>
<td>IRQ_EVT_SINT29</td>
<td>Software Interrupt #29</td>
</tr>
<tr>
<td>IRQ_EVT_SINT30</td>
<td>Software Interrupt #30</td>
</tr>
<tr>
<td>IRQ_EVT_SINT0</td>
<td>Software Interrupt #0</td>
</tr>
<tr>
<td>IRQ_EVT_SINT1</td>
<td>Software Interrupt #1</td>
</tr>
<tr>
<td>IRQ_EVT_SINT2</td>
<td>Software Interrupt #2</td>
</tr>
<tr>
<td>IRQ_EVT_SINT3</td>
<td>Software Interrupt #3</td>
</tr>
<tr>
<td>IRQ_EVT_SINT4</td>
<td>Software Interrupt #4</td>
</tr>
<tr>
<td>IRQ_EVT_SINT5</td>
<td>Software Interrupt #5</td>
</tr>
</tbody>
</table>
Table 12–3. IRQ_EVT_NNNN Events List (Continued)

<table>
<thead>
<tr>
<th>Constant</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRQ_EVT_SINT6</td>
<td>Software Interrupt #6</td>
</tr>
<tr>
<td>IRQ_EVT_SINT7</td>
<td>Software Interrupt #7</td>
</tr>
<tr>
<td>IRQ_EVT_SINT8</td>
<td>Software Interrupt #8</td>
</tr>
<tr>
<td>IRQ_EVT_SINT9</td>
<td>Software Interrupt #9</td>
</tr>
<tr>
<td>IRQ_EVT_SINT10</td>
<td>Software Interrupt #10</td>
</tr>
<tr>
<td>IRQ_EVT_SINT11</td>
<td>Software Interrupt #11</td>
</tr>
<tr>
<td>IRQ_EVT_SINT12</td>
<td>Software Interrupt #12</td>
</tr>
<tr>
<td>IRQ_EVT_SINT13</td>
<td>Software Interrupt #13</td>
</tr>
<tr>
<td>IRQ_EVT_INT0</td>
<td>External User Interrupt #0</td>
</tr>
<tr>
<td>IRQ_EVT_INT1</td>
<td>External User Interrupt #1</td>
</tr>
<tr>
<td>IRQ_EVT_INT2</td>
<td>External User Interrupt #2</td>
</tr>
<tr>
<td>IRQ_EVT_INT3</td>
<td>External User Interrupt #3</td>
</tr>
<tr>
<td>IRQ_EVT_TINT0</td>
<td>Timer 0 Interrupt</td>
</tr>
<tr>
<td>IRQ_EVT_HINT</td>
<td>Host Interrupt (HPI)</td>
</tr>
<tr>
<td>IRQ_EVT_DMA0</td>
<td>DMA Channel 0 Interrupt</td>
</tr>
<tr>
<td>IRQ_EVT_DMA1</td>
<td>DMA Channel 1 Interrupt</td>
</tr>
<tr>
<td>IRQ_EVT_DMA2</td>
<td>DMA Channel 2 Interrupt</td>
</tr>
<tr>
<td>IRQ_EVT_DMA3</td>
<td>DMA Channel 3 Interrupt</td>
</tr>
<tr>
<td>IRQ_EVT_DMA4</td>
<td>DMA Channel 4 Interrupt</td>
</tr>
<tr>
<td>IRQ_EVT_DMA5</td>
<td>DMA Channel 5 Interrupt</td>
</tr>
<tr>
<td>IRQ_EVT_RINT0</td>
<td>MCBSP Port #0 Receive Interrupt</td>
</tr>
<tr>
<td>IRQ_EVT_XINT0</td>
<td>MCBSP Port #0 Transmit Interrupt</td>
</tr>
<tr>
<td>IRQ_EVT_RINT2</td>
<td>MCBSP Port #2 Receive Interrupt</td>
</tr>
<tr>
<td>IRQ_EVT_XINT2</td>
<td>MCBSP Port #2 Transmit Interrupt</td>
</tr>
<tr>
<td>IRQ_EVT_TINT1</td>
<td>Timer #1 Interrupt</td>
</tr>
<tr>
<td>IRQ_EVT_HPIINT</td>
<td>Host Interrupt (HPI)</td>
</tr>
</tbody>
</table>
Table 12–3. IRQ_EVT_NNNN Events List (Continued)

<table>
<thead>
<tr>
<th>Constant</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRQ_EVT_RINT1</td>
<td>MCBSP Port #1 Receive Interrupt</td>
</tr>
<tr>
<td>IRQ_EVT_XINT1</td>
<td>MCBSP Port #1 Transmit Interrupt</td>
</tr>
<tr>
<td>IRQ_EVT_IPINT</td>
<td>FIFO Full Interrupt</td>
</tr>
<tr>
<td>IRQ_EVT_SINT14</td>
<td>Software Interrupt #14</td>
</tr>
<tr>
<td>IRQ_EVT_RTC</td>
<td>RTC Interrupt</td>
</tr>
<tr>
<td>IRQ_EVT_I2C</td>
<td>I2C Interrupt</td>
</tr>
<tr>
<td>IRQ_EVT_WDTINT</td>
<td>Watchdog Timer Interrupt</td>
</tr>
</tbody>
</table>
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**

```c
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.

**IRA_Config** /* IRQ configuration structure */

**Structure**
- IRQ_Config

**Members**
- IRA_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

```c
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

```c
IRQ_clear(IRQ_EVT_TINT0);
```

**IRQ_config**

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

Function

```c
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

```c
    IRQ_config myConfig = {  
        0X0000,  
        0X0300,  
        0X0000,  
        0X0000
    };
    IRQ_config (IRQ_EVT_TINT0, &myConfig);
```
**IRQ_disable**  
*Disables specified event*

**Function**
```c
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**
```c
Uint32 oldint;
oldint = IRQ_disable(IRQ_EVT_TINT0);
```

**IRQ_enable**  
*Enables specified event*

**Function**
```c
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 EventID.

**Return Value**
None

**Description**
Enables the specified event.

**Example**
```c
Uint32 oldint;
oldint = IRQ_enable(IRQ_EVT_TINT0);
```

**IRQ_getArg**  
*Gets value for specified event*

**Function**
```c
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);

**IRAQ_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: 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

```c
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

```c
int intm;
intm = IRQ_globalEnable();
...
IRQ_globalRestore (intm);
```

**IRQ_globalRestore**

**Restores the global interrupt mask state**

Function

```c
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

```c
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, seeIRQ.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, seeIRQ.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 Module*  
12-13
**IRQ_setArg**

Sets value of argument for DSPBIOS dispatch entry

Function

```c
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.

- **val**: Value that DSP/BIOS dispatcher will pass to the interrupt service routine for the specified event.

Return Value

None

Description

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

Example

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

---

**IRQ_restore**

Restores the state of a specified event

Function

```c
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

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

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.1 Overview</td>
<td>13-2</td>
</tr>
<tr>
<td>13.2 Configuration Structures</td>
<td>13-6</td>
</tr>
<tr>
<td>13.4 Functions</td>
<td>13-8</td>
</tr>
<tr>
<td>13.5 Macros</td>
<td>13-23</td>
</tr>
<tr>
<td>13.6 Examples</td>
<td>13-26</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
<th>See page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCBSP_Config</td>
<td>McBSP configuration structure used to setup a McBSP port.</td>
<td>13-6</td>
</tr>
</tbody>
</table>

Table 13–2. McBSP Functions

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

#### Syntax

| 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**

<table>
<thead>
<tr>
<th>Register</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPCR1</td>
<td>DLB, RJUST, CLKSTP, DXENA, ABIS, RINTM, RSYNCERR, (R)RFULL, (R)RRDY, RRST</td>
</tr>
<tr>
<td>SPCR2</td>
<td>FREE, SOFT, FRST, GRST, XINTM, XSYNCERR, (R)XEMPTY, (R)XRDY, XRST</td>
</tr>
<tr>
<td>PCR</td>
<td>SCLKME, (R)CLKSSTAT, DXSTAT, (R)DRSTAT, FSXP, FSRP, CLKXP, CLKRP, IDLEEN, XIOEN, RIOEN, FSXM, FSRM, CLKXM, CLKRM</td>
</tr>
<tr>
<td>RCR1</td>
<td>RFRLEN1, RWDLEN1</td>
</tr>
<tr>
<td>RCR2</td>
<td>RPHASE, RFRLEN2, RWDLEN2, RCOMPAND, RFIG, RDATDLY</td>
</tr>
<tr>
<td>XCR1</td>
<td>XFRLEN1, XWDLEN1</td>
</tr>
<tr>
<td>XCR2</td>
<td>XPHASE, XFRLEN2, XWDLEN2, XCOMPAND, XFIG, XDATDLY</td>
</tr>
<tr>
<td>SRGR1</td>
<td>FWID, CLKGDV</td>
</tr>
<tr>
<td>SRGR2</td>
<td>GSYNC, CLKSP, CLKSM, FSGM, FPER</td>
</tr>
<tr>
<td>MCR1</td>
<td>RMCME, RPBBLK, RPABLK, (R)RCBLK, RMCM</td>
</tr>
<tr>
<td>MCR2</td>
<td>XMCME, XPBBLK, XPABLK, (R)XCBLK, XMCM</td>
</tr>
<tr>
<td>XCERA</td>
<td>XCEY15, XCEY14, XCEY13, XCEY12, XCEY11, XCEY10, XCEY9, XCEY8, XCEY7, XCEY6, XCEY5, XCEY4, XCEY3, XCEY2, XCEY1, XCEY0</td>
</tr>
</tbody>
</table>
### Table 13-3. MCBSP Registers (Continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>XCRB</td>
<td>XCEY15, XCEY14, XCEY13, XCEY12, XCEY11, XCEY10, XCEY9, XCEY8, XCEY7, XCEY6, XCEY5, XCEY4, XCEY3, XCEY2, XCEY1, XCEY0</td>
</tr>
<tr>
<td>XCRC</td>
<td>XCEY15, XCEY14, XCEY13, XCEY12, XCEY11, XCEY10, XCEY9, XCEY8, XCEY7, XCEY6, XCEY5, XCEY4, XCEY3, XCEY2, XCEY1, XCEY0</td>
</tr>
<tr>
<td>XCRD</td>
<td>XCEY15, XCEY14, XCEY13, XCEY12, XCEY11, XCEY10, XCEY9, XCEY8, XCEY7, XCEY6, XCEY5, XCEY4, XCEY3, XCEY2, XCEY1, XCEY0</td>
</tr>
<tr>
<td>XCRE</td>
<td>XCEY15, XCEY14, XCEY13, XCEY12, XCEY11, XCEY10, XCEY9, XCEY8, XCEY7, XCEY6, XCEY5, XCEY4, XCEY3, XCEY2, XCEY1, XCEY0</td>
</tr>
<tr>
<td>XCRF</td>
<td>XCEY15, XCEY14, XCEY13, XCEY12, XCEY11, XCEY10, XCEY9, XCEY8, XCEY7, XCEY6, XCEY5, XCEY4, XCEY3, XCEY2, XCEY1, XCEY0</td>
</tr>
<tr>
<td>XCRG</td>
<td>XCEY15, XCEY14, XCEY13, XCEY12, XCEY11, XCEY10, XCEY9, XCEY8, XCEY7, XCEY6, XCEY5, XCEY4, XCEY3, XCEY2, XCEY1, XCEY0</td>
</tr>
<tr>
<td>XCRH</td>
<td>XCEY15, XCEY14, XCEY13, XCEY12, XCEY11, XCEY10, XCEY9, XCEY8, XCEY7, XCEY6, XCEY5, XCEY4, XCEY3, XCEY2, XCEY1, XCEY0</td>
</tr>
<tr>
<td>RCRA</td>
<td>RCEY15, RCEY14, RCEY13, RCEY12, RCEY11, RCEY10, RCEY9, RCEY8, RCEY7, RCEY6, RCEY5, RCEY4, RCEY3, RCEY2, RCEY1, RCEY0</td>
</tr>
<tr>
<td>RCRA</td>
<td>RCEY15, RCEY14, RCEY13, RCEY12, RCEY11, RCEY10, RCEY9, RCEY8, RCEY7, RCEY6, RCEY5, RCEY4, RCEY3, RCEY2, RCEY1, RCEY0</td>
</tr>
<tr>
<td>RCRC</td>
<td>RCEY15, RCEY14, RCEY13, RCEY12, RCEY11, RCEY10, RCEY9, RCEY8, RCEY7, RCEY6, RCEY5, RCEY4, RCEY3, RCEY2, RCEY1, RCEY0</td>
</tr>
<tr>
<td>RCRC</td>
<td>RCEY15, RCEY14, RCEY13, RCEY12, RCEY11, RCEY10, RCEY9, RCEY8, RCEY7, RCEY6, RCEY5, RCEY4, RCEY3, RCEY2, RCEY1, RCEY0</td>
</tr>
<tr>
<td>RCRD</td>
<td>RCEY15, RCEY14, RCEY13, RCEY12, RCEY11, RCEY10, RCEY9, RCEY8, RCEY7, RCEY6, RCEY5, RCEY4, RCEY3, RCEY2, RCEY1, RCEY0</td>
</tr>
<tr>
<td>RCRD</td>
<td>RCEY15, RCEY14, RCEY13, RCEY12, RCEY11, RCEY10, RCEY9, RCEY8, RCEY7, RCEY6, RCEY5, RCEY4, RCEY3, RCEY2, RCEY1, RCEY0</td>
</tr>
<tr>
<td>RCRE</td>
<td>RCEY15, RCEY14, RCEY13, RCEY12, RCEY11, RCEY10, RCEY9, RCEY8, RCEY7, RCEY6, RCEY5, RCEY4, RCEY3, RCEY2, RCEY1, RCEY0</td>
</tr>
<tr>
<td>RCRE</td>
<td>RCEY15, RCEY14, RCEY13, RCEY12, RCEY11, RCEY10, RCEY9, RCEY8, RCEY7, RCEY6, RCEY5, RCEY4, RCEY3, RCEY2, RCEY1, RCEY0</td>
</tr>
<tr>
<td>RCF</td>
<td>RCEY15, RCEY14, RCEY13, RCEY12, RCEY11, RCEY10, RCEY9, RCEY8, RCEY7, RCEY6, RCEY5, RCEY4, RCEY3, RCEY2, RCEY1, RCEY0</td>
</tr>
<tr>
<td>RCF</td>
<td>RCEY15, RCEY14, RCEY13, RCEY12, RCEY11, RCEY10, RCEY9, RCEY8, RCEY7, RCEY6, RCEY5, RCEY4, RCEY3, RCEY2, RCEY1, RCEY0</td>
</tr>
<tr>
<td>RCG</td>
<td>RCEY15, RCEY14, RCEY13, RCEY12, RCEY11, RCEY10, RCEY9, RCEY8, RCEY7, RCEY6, RCEY5, RCEY4, RCEY3, RCEY2, RCEY1, RCEY0</td>
</tr>
<tr>
<td>RCG</td>
<td>RCEY15, RCEY14, RCEY13, RCEY12, RCEY11, RCEY10, RCEY9, RCEY8, RCEY7, RCEY6, RCEY5, RCEY4, RCEY3, RCEY2, RCEY1, RCEY0</td>
</tr>
<tr>
<td>RCCH</td>
<td>RCEY15, RCEY14, RCEY13, RCEY12, RCEY11, RCEY10, RCEY9, RCEY8, RCEY7, RCEY6, RCEY5, RCEY4, RCEY3, RCEY2, RCEY1, RCEY0</td>
</tr>
<tr>
<td>DRR1</td>
<td>RCEY15, RCEY14, RCEY13, RCEY12, RCEY11, RCEY10, RCEY9, RCEY8, RCEY7, RCEY6, RCEY5, RCEY4, RCEY3, RCEY2, RCEY1, RCEY0</td>
</tr>
</tbody>
</table>
Table 13–3. MCBSP Registers (Continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRR2</td>
<td>RCEY15, RCEY14, RCEY13, RCEY12, RCEY11, RCEY10, RCEY9, RCEY8, RCEY7, RCEY6, RCEY5, RCEY4, RCEY3, RCEY2, RCEY1, RCEY0</td>
</tr>
<tr>
<td>DXR1</td>
<td>RCEY15, RCEY14, RCEY13, RCEY12, RCEY11, RCEY10, RCEY9, RCEY8, RCEY7, RCEY6, RCEY5, RCEY4, RCEY3, RCEY2, RCEY1, RCEY0</td>
</tr>
<tr>
<td>DXR2</td>
<td>RCEY15, RCEY14, RCEY13, RCEY12, RCEY11, RCEY10, RCEY9, RCEY8, RCEY7, RCEY6, RCEY5, RCEY4, RCEY3, RCEY2, RCEY1, RCEY0</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Members</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uint16 spcr1</td>
<td>Serial port control register 1 value</td>
</tr>
<tr>
<td>Uint16 spcr2</td>
<td>Serial port control register 2 value</td>
</tr>
<tr>
<td>Uint16 rcr1</td>
<td>Receive control register 1 value</td>
</tr>
<tr>
<td>Uint16 rcr2</td>
<td>Receive control register 2 value</td>
</tr>
<tr>
<td>Uint16 xcr1</td>
<td>Transmit control register 1 value</td>
</tr>
<tr>
<td>Uint16 xcr2</td>
<td>Transmit control register 2 value</td>
</tr>
<tr>
<td>Uint16 srgr1</td>
<td>Sample rate generator register 1 value</td>
</tr>
<tr>
<td>Uint16 srgr2</td>
<td>Sample rate generator register 2 value</td>
</tr>
<tr>
<td>Uint16 mcr1</td>
<td>Multi-channel control register 1 value</td>
</tr>
<tr>
<td>Uint16 mcr2</td>
<td>Multi-channel control register 2 value</td>
</tr>
<tr>
<td>Uint16 pcr</td>
<td>Pin control register value</td>
</tr>
<tr>
<td>Uint16 rcera</td>
<td>Receive channel enable register partition A value</td>
</tr>
<tr>
<td>Uint16 rcerb</td>
<td>Receive channel enable register partition B value</td>
</tr>
<tr>
<td>Uint16 cerc</td>
<td>Receive channel enable register partition C value</td>
</tr>
<tr>
<td>Uint16 cerd</td>
<td>Receive channel enable register partition D value</td>
</tr>
<tr>
<td>Uint16 cere</td>
<td>Receive channel enable register partition E value</td>
</tr>
<tr>
<td>Uint16 cera</td>
<td>Receive channel enable register partition F value</td>
</tr>
<tr>
<td>Uint16 xcera</td>
<td>Receive channel enable register partition G value</td>
</tr>
<tr>
<td>Uint16 xcerb</td>
<td>Receive channel enable register partition H value</td>
</tr>
<tr>
<td>Uint16 xcerb</td>
<td>Transmit channel enable register partition A value</td>
</tr>
<tr>
<td>Uint16 xcerb</td>
<td>Transmit channel enable register partition B value</td>
</tr>
<tr>
<td>Uint16 xcerb</td>
<td>Transmit channel enable register partition C value</td>
</tr>
<tr>
<td>Uint16 xcerb</td>
<td>Transmit channel enable register partition D value</td>
</tr>
<tr>
<td>Uint16 xcerb</td>
<td>Transmit channel enable register partition E value</td>
</tr>
<tr>
<td>Uint16 xcerb</td>
<td>Transmit channel enable register partition F value</td>
</tr>
<tr>
<td>Uint16 xcerb</td>
<td>Transmit channel enable register partition G value</td>
</tr>
<tr>
<td>Uint16 xcerb</td>
<td>Transmit channel enable register partition H value</td>
</tr>
</tbody>
</table>

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.
Example

```c
MCBSP_Config config1 = {
  0xFFFF, /* spcr1 */
  0x03FF, /* spcr2 */
  0x7FE0, /* rcr1 */
  0xFFFF, /* rcr2 */
  0x7FE0, /* xcrl */
  0xFFFF, /* xcr2 */
  0xFFFF, /* srgr1 */
  0xFFFF, /* srgr2 */
  0x03FF, /* mcr1 */
  0x03FF, /* mcr2 */
  0xFFFF, /* pcr */
  0xFFFF, /* rcera */
  0xFFFF, /* rcerb */
  0xFFFF, /* rcerc */
  0xFFFF, /* rcerd */
  0xFFFF, /* rcer */
  0xFFFF, /* rcerf */
  0xFFFF, /* rcer */
  0xFFFF, /* xcera */
  0xFFFF, /* xcerb */
  0xFFFF, /* xcer */
  0xFFFF, /* xcer */
  0xFFFF, /* xcer */
  0xFFFF, /* xcer */
  0xFFFF, /* xcer */
  0xFFFF, /* xcer */

};
```

```c
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.

**Disables one or several McBSP channels**

Function

```c
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_channelEnable

- 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));

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
**MCBSP_channelEnable**

- XERF
- XERG
- XERH

**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_CHANNEL14
  - MCBSP_CHAN15
**MCBSP_close**

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 cleared.

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

```c
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, /* xcera */
   0xFFFF, /* xcerb */
   0xFFFF, /* xcerc */
   0xFFFF, /* xced */
   0xFFFF, /* xcerf */
   0xFFFF, /* xcerg */
   0xFFFF /* xcerh */
};
...
MCBSP_getPort

MCBSP_getConfig  *Reads the MCBSP configuration in the configuration structure*

Function  

```c
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  

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

MCBSP_getPort  *Get McBSP port number used in given handle*

Function  

```c
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  

```c
Uint16  PortNum;
...
    PortNum = MCBSP_getPort (hMcbsp));
```
**MCBSP_getRcvEventId**  
*Retrieves the receive event ID for a given McBSP port*

**Function**

```c
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**

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

---

**MCBSP_getXmtEventId**  
*Retrieves the transmit event ID for a given McBSP port*

**Function**

```c
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**

```c
Uint16 XmtEventId;
...
XmtEventId = MCBSP_getXmtEventId(hMcbsp);
IRQ_enable(XmtEventId);
```
**MCBSP_open**

*Opens a McBSP port*

**Function**

```c
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 interrupts are disabled and cleared.

**Example**

```c
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**
  ```c
  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**
  ```c
  MCBSP_reset(hMcbsp);
  MCBSP_reset(INV);
  ```

MCBSP_rfull

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

  **Function**
  ```c
  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 occurred.)

  **Example**
  ```c
  if (MCBSP_rfull(hMcbsp)) {
      ...
  }
  ```
MCBSP_start

### MCBSP_rrdy

**Reads RRDY status bit of SPCR1 register**

**Function**

```c
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**

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

---

### MCBSP_start

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

**Function**

```c
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:
MCBSP_start

SampleRateGenDelay = \frac{2 \times \text{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

```c
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

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

### MCBSP_write32

**Writes a 32-bit value**

#### Function

```c
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

```c
Uint32 val32;
MCBSP_write32(hMcbsp, val32);
```
**MCBSP_xempty**

**MCBSP_xempty**  *Reads XEMPTY bit from SPCR2 register*

**Function**  
CSLBool MCBSP_xempty(
    MCBSP_Handle hMcbsp
);  

**Arguments**  

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hMcbsp</td>
<td>Handle to McBSP port obtained by MCBSP_open()</td>
</tr>
</tbody>
</table>

**Return Value**  

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>transmit buffer empty</td>
</tr>
<tr>
<td>1</td>
<td>transmit buffer full</td>
</tr>
</tbody>
</table>

**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**

**MCBSP_xrdy**  *Reads XRDY status bit of SPCR2 register*

**Function**  
CSLBool MCBSP_xrdy(
    MCBSP_Handle hMcbsp
);  

**Arguments**  

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hMcbsp</td>
<td>Handle to McBSP port obtained by MCBSP_open()</td>
</tr>
</tbody>
</table>

**Return Value**  

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>not ready to transmit new data</td>
</tr>
<tr>
<td>1</td>
<td>ready to transmit new data</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCBSP_RGET()</td>
<td>Uint16 MCBSP_RGET(REG#)</td>
</tr>
<tr>
<td>MCBSP_RSET()</td>
<td>Void MCBSP_RSET(REG#, Uint16 regval)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCBSP_FGET()</td>
<td>Uint16 MCBSP_FGET(REG#, FIELD)</td>
</tr>
<tr>
<td>MCBSP_FSET()</td>
<td>Void MCBSP_FSET(REG#, FIELD, Uint16 fieldval)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCBSP_REG_RMK()</td>
<td>Uint16 MCBSP_REG_RMK(fieldval_n,...,fieldval_0)</td>
</tr>
<tr>
<td>Note:</td>
<td>*Start with field values with most significant field positions: field_n: MSB field field_0: LSB field *only writable fields allowed</td>
</tr>
<tr>
<td>MCBSP_FMK()</td>
<td>Uint16 MCBSP_FMK(REG, FIELD, fieldval)</td>
</tr>
</tbody>
</table>
Table 13–4. McBSP Macros Using McBSP Port Number (Continued)

(d) Macros to read a register address

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCBSP_ADDR()</td>
<td>Uint16 MCBSP_ADDR(REG#)</td>
</tr>
</tbody>
</table>

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, XCRB, XCRC, XCRE, XCRF, XCRG, XCRH, 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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCBSP_RGETH()</td>
<td>Uint16 MCBSP_RGETH(MCBSP_Handle hMCBSP, REG)</td>
</tr>
<tr>
<td>MCBSP_RSETH()</td>
<td>Void MCBSP_RSETH(MCBSP_Handle hMCBSP, REG, Uint16 regval)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCBSP_FGETH()</td>
<td>Uint16 MCBSP_FGETH(MCBSP_Handle hMCBSP, REG, FIELD)</td>
</tr>
<tr>
<td>MCBSP_FSETH()</td>
<td>Void MCBSP_FSETH(MCBSP_Handle hMCBSP, REG, FIELD, Uint16 fieldval)</td>
</tr>
</tbody>
</table>
Table 13–5. McBSP CSL Macros Using Handle (Continued)

(c) Macros to read a register address

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCBSP_ADDRH()</td>
<td>Uint16 MCBSP_ADDRH(MCBSP_Handle hMCBSP, REG)</td>
</tr>
</tbody>
</table>

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()**

```c
#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 generation 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 13-2  McBSP Port Initialization Using MCBSP_config() (Continued)

```c
/* 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 structures, and functions available in the multimedia controller (MMC) module. This module supports both MMC and SD cards. The initialization and data transfer to MMC and SD cards differ in a few aspects, and there are SD_<function> APIs provided for accessing the SD card. The MMC APIs and data structures that are valid only for MMC cards are marked accordingly. All other APIs can be used for both MMC and SD cards.

Note: The SPI mode is no longer supported on the 5509 MMC Controller.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>14.1 Overview</td>
<td>14-2</td>
</tr>
<tr>
<td>14.2 Configuration Structures</td>
<td>14-5</td>
</tr>
<tr>
<td>14.3 Data Structures</td>
<td>14-6</td>
</tr>
<tr>
<td>14.4 Functions</td>
<td>14-13</td>
</tr>
</tbody>
</table>
14.1 Overview

Table 14–1. MMC Configuration Structures

<table>
<thead>
<tr>
<th>Config Structure</th>
<th>Description</th>
<th>See Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMC_Config</td>
<td>MMC configuration structure</td>
<td>14-5</td>
</tr>
</tbody>
</table>

Table 14–2. MMC Data Structures

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Description</th>
<th>See Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMC_CallBackObj</td>
<td>Structure used to assign functions for each interrupt</td>
<td>14-6</td>
</tr>
<tr>
<td>MMC_CardCsdObj†</td>
<td>Contains card-specific data</td>
<td>14-7</td>
</tr>
<tr>
<td>SD_CardCsdObj‡</td>
<td>Contains card-specific data</td>
<td>14-8</td>
</tr>
<tr>
<td>MMC_CardIdObj</td>
<td>Contains card identification (CID)</td>
<td>14-9</td>
</tr>
<tr>
<td>MMC_CardObj</td>
<td>Contains information about memory cards including CID and CSD structures for MMC/SD cards</td>
<td>14-10</td>
</tr>
<tr>
<td>MMC_CardXCsdObj</td>
<td>Extended card-specific data (XCSD)</td>
<td>14-10</td>
</tr>
<tr>
<td>MMC_Cmdobj</td>
<td>Structure to store commands</td>
<td>14-11</td>
</tr>
<tr>
<td>MMC_MmcRegObj</td>
<td>Structure to store values of all MMC regs</td>
<td>14-11</td>
</tr>
<tr>
<td>MMC_SetupNative</td>
<td>Native mode Initialization Structure</td>
<td>14-12</td>
</tr>
<tr>
<td>MMC_RspRegObj</td>
<td>Structure to store values of MMC response registers</td>
<td>14-12</td>
</tr>
</tbody>
</table>
† Only for MMC card
‡ Only for SD card

Table 14–3. MMC Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>See Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMC_clearResponse</td>
<td>Clears the MMC response registers</td>
<td>14-13</td>
</tr>
<tr>
<td>MMC_close</td>
<td>Frees MMC controller reserved by call to MMC_open</td>
<td>14-13</td>
</tr>
<tr>
<td>MMC_config</td>
<td>Writes the values of the configuration structure into the control registers for the specified MMC controller</td>
<td>14-14</td>
</tr>
<tr>
<td>MMC_dispatch0</td>
<td>ISR dispatch function to service MMC0 (port0) isrs</td>
<td>14-14</td>
</tr>
<tr>
<td>MMC_dispatch1</td>
<td>ISR dispatch function to service MMC1 (port1) isrs</td>
<td>14-14</td>
</tr>
</tbody>
</table>
### Table 14–3. MMC Functions (Continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>See Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMC_drrdy</td>
<td>Returns the contents of the DRRDY status bit in the MMCST0 register</td>
<td>14-15</td>
</tr>
<tr>
<td>MMC_dxrdy</td>
<td>Returns the contents of the DXRDY status bit in the MMCST0 register</td>
<td>14-15</td>
</tr>
<tr>
<td>MMC_getCardCsd†</td>
<td>Reads the card-specific data from response registers</td>
<td>14-16</td>
</tr>
<tr>
<td>MMC_getCardId†</td>
<td>Reads card ID from the MMC response registers</td>
<td>14-16</td>
</tr>
<tr>
<td>MMC_getConfig</td>
<td>Returns the current contents of the MMC control registers. This excludes the MMC response registers.</td>
<td>14-17</td>
</tr>
<tr>
<td>MMC_getNumberOfCards</td>
<td>Returns the number of cards found when MMC_open is called with MMC_OPEN_SENDALLCID option</td>
<td>14-17</td>
</tr>
<tr>
<td>MMC_getStatus</td>
<td>Returns the status of the specified field in the MMCST0 register</td>
<td>14-18</td>
</tr>
<tr>
<td>MMC_intEnable</td>
<td>Enables interrupts by writing to the MMCIE register</td>
<td>14-28</td>
</tr>
<tr>
<td>MMC_SetupNative</td>
<td>Initializes the controller when in Native mode</td>
<td>14-12</td>
</tr>
<tr>
<td>MMC_open</td>
<td>Reserves the MMC device specified by, device</td>
<td>14-18</td>
</tr>
<tr>
<td>MMC_read</td>
<td>Sends commands to read blocks of data. This is a blocking function in that it does not return until all data has been transferred.</td>
<td>14-19</td>
</tr>
<tr>
<td>MMC_responseDone</td>
<td>Checks the status of a register for a response complete condition</td>
<td>14-19</td>
</tr>
<tr>
<td>MMC_saveStatus</td>
<td>Saves current contents of MMCST0 register in MMC Handle</td>
<td>14-20</td>
</tr>
<tr>
<td>MMC_selectCard</td>
<td>Selects card with specified relative address for communication</td>
<td>14-20</td>
</tr>
<tr>
<td>MMC_sendAllCID†</td>
<td>Sends broadcast command to all cards to identify themselves</td>
<td>14-21</td>
</tr>
<tr>
<td>MMC_sendCmd</td>
<td>Sends a command to selected memory card/s. Optionally waits for a response</td>
<td>14-22</td>
</tr>
<tr>
<td>MMC_sendCSD</td>
<td>Sends a request to card to submit its card-specific data or CSD structure</td>
<td>14-22</td>
</tr>
<tr>
<td>MMC_sendGoIdle</td>
<td>Sends a broadcast GO_IDLE command</td>
<td>14-23</td>
</tr>
<tr>
<td>MMC_setCardPtr</td>
<td>Sets the card pointer in the MMC global status table</td>
<td>14-23</td>
</tr>
<tr>
<td>MMC_setCardType</td>
<td>Writes the card type (MMC or SD) to the MMC_CardObj structure</td>
<td>14-29</td>
</tr>
</tbody>
</table>
## Table 14–3. MMC Functions (Continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>See Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMC_sendOpCond</td>
<td>Sets the operating voltage window while in Native mode</td>
<td>14-24</td>
</tr>
<tr>
<td>MMC_setCallBack</td>
<td>Associated functions to interrupts and installs dispatcher routines</td>
<td>14-25</td>
</tr>
<tr>
<td>MMC_setRca</td>
<td>Set the relative card address of an attached memory card</td>
<td>14-25</td>
</tr>
<tr>
<td>MMC_stop</td>
<td>Halts a current data transfer</td>
<td>14-26</td>
</tr>
<tr>
<td>MMC_waitForFlag</td>
<td>Waits for a particular field in the MMCST0 register to be set</td>
<td>14-26</td>
</tr>
<tr>
<td>MMC_write</td>
<td>Writes a block of data. This is a blocking function in that it does not return until all data has been transferred</td>
<td>14-27</td>
</tr>
<tr>
<td>SD_sendAllCID</td>
<td>Sends broadcast command to SD cards to identify themselves</td>
<td>14-29</td>
</tr>
<tr>
<td>SD_getCardId</td>
<td>Reads SD specific card ID from MMC response registers</td>
<td>14-30</td>
</tr>
<tr>
<td>SD_getCardCsd</td>
<td>Reads SD card-specific data from response registers</td>
<td>14-30</td>
</tr>
<tr>
<td>SD_sendRca</td>
<td>Asks the SD card to respond with its relative card address</td>
<td>14-31</td>
</tr>
<tr>
<td>SD_setWidth</td>
<td>Sets the data bus width to either 1 bit or 4 bits</td>
<td>14-31</td>
</tr>
</tbody>
</table>

† Only for MMC card
14.2 Configuration Structures

The section contains the configuration structures available for the MMC module.

### MMC Configuration Structure

**Structure**

void MMC_Config

**Members**

- `Uint16 mmcctl /* Control register */`
- `Uint16 mmcfclk /* Functional Clock register */`
- `Uint16 mmcclk /* Clock Control register */`
- `Uint16 mmcie /* Interrupt Enable register */`
- `Uint16 mmctor /* Timeout Response register */`
- `Uint16 mmctod /* Timeout Data Read register */`
- `Uint16 mmcblen /* Block Length register */`
- `Uint16 mmcnblk /* Number of Block register */`

**Description**

MMC configuration structure used to set up the MMC interface. You create and initialize this structure and then pass its address to the MMC_config() function.

**Example**

```c
MMC_Config Config = {
    0x000F, /* MMCCTL */
    0x0F00, /* MMCFCLK */
    0x0001, /* MMCCLK */
    0x0FA0, /* MMCIE */
    0x0500, /* MMCTOR */
    0x0500, /* MMCTOD */
    0x0200, /* MMCBLEN */
    0x0001 /* MMCNBLK */
};
```
**MMC_CallBackObj**

### 14.3 Data Structures

This section contains the data structures available for use with the MMC module.

** MMC_CallBackObj – Configures pointers to functions **

- **Structure:** MMC_CallBackObj
- **Members**

  - MMC_CallBackPtr isr [12]
  - Holds the functions to be involved for MMC interrupt

- **Description**

  Configures pointers to functions.

- **Example**

  ```c
  MMC_CallBackObj cback = {
      (MMC_CallBackPtr)0x0000, /* Callback for Data Transfer Done */
      (MMC_CallBackPtr)0x0000, /* Callback for Busy Done */
      (MMC_CallBackPtr)0x0000, /* Callback for response Done */
      (MMC_CallBackPtr)0x0000, /* Callback for Read-data time-out */
      (MMC_CallBackPtr)0x0000, /* Callback for Response time-out */
      (MMC_CallBackPtr)0x0000, /* Callback for write-data CRC error */
      (MMC_CallBackPtr)0x0000, /* Callback for read-data CRC error */
      (MMC_CallBackPtr)0x0000, /* Callback for response CRC error */
      (MMC_CallBackPtr)0x0000, /* This is never used */
      write_interrupt, /* Callback for data xmt ready */
      read_interrupt, /* Callback for data rcv ready */
      (MMC_CallBackPtr)0x0000 /* Callback for DAT3 edge */
  };
  ```
MMC_CardCsdobj

Contains Card Specific Data (CSD)

Structure

MMC_CardCsdObj

Members

<table>
<thead>
<tr>
<th>Uint16</th>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>csdStructure</td>
<td>2-bit structure type field</td>
</tr>
<tr>
<td></td>
<td>mmcProt</td>
<td>2-bit MMC protocol</td>
</tr>
<tr>
<td></td>
<td>taac</td>
<td>8-bit TAAC</td>
</tr>
<tr>
<td></td>
<td>nsac</td>
<td>8-bit NSAC</td>
</tr>
<tr>
<td></td>
<td>tranSpeed</td>
<td>8-bit max data transmission speed</td>
</tr>
<tr>
<td></td>
<td>ccc</td>
<td>12-bit card command classes</td>
</tr>
<tr>
<td></td>
<td>readBLLen</td>
<td>4-bit maximum Read Block Length</td>
</tr>
<tr>
<td></td>
<td>readBIPartial</td>
<td>1-bit indicates if partial read blocks allowed</td>
</tr>
<tr>
<td></td>
<td>writeBkMisalign</td>
<td>1-bit flag indicates write block misalignment</td>
</tr>
<tr>
<td></td>
<td>readBkMisalign</td>
<td>1-bit flag indicates read block misalignment</td>
</tr>
<tr>
<td></td>
<td>dsrlmp</td>
<td>1-bit flag indicates whether card has DSR reg</td>
</tr>
<tr>
<td></td>
<td>cSize</td>
<td>12-bit device size</td>
</tr>
<tr>
<td></td>
<td>vddRCurrMin</td>
<td>3-bit Max. Read Current @ Vdd Min</td>
</tr>
<tr>
<td></td>
<td>vddRCurrMax</td>
<td>3-bit Max. Read Current @ Vdd Max</td>
</tr>
<tr>
<td></td>
<td>vddWCurrMin</td>
<td>3-bit Max. Write Current @ Vdd Min</td>
</tr>
<tr>
<td></td>
<td>vddWCurrMax</td>
<td>3-bit Max. Write Current @ Vdd Max</td>
</tr>
<tr>
<td></td>
<td>cSizeMult</td>
<td>3-bit device size multiplier</td>
</tr>
<tr>
<td></td>
<td>eraseGrpSize</td>
<td>5-bit erase sector size</td>
</tr>
<tr>
<td></td>
<td>eraseGrpMult</td>
<td>5-bit erase group multiplier</td>
</tr>
<tr>
<td></td>
<td>wpGrpSize</td>
<td>5-bit write protect group size</td>
</tr>
<tr>
<td></td>
<td>wpGrpEnable</td>
<td>1-bit write protect enable flag</td>
</tr>
<tr>
<td></td>
<td>defaultEcc</td>
<td>2-bit Manufacturer default ECC</td>
</tr>
<tr>
<td></td>
<td>r2wFactor</td>
<td>3-bit stream write factor</td>
</tr>
<tr>
<td></td>
<td>writeBLLen</td>
<td>4-bit maximum write block length</td>
</tr>
<tr>
<td></td>
<td>writeBIPartial</td>
<td>1-bit indicates if partial write blocks allowed</td>
</tr>
<tr>
<td></td>
<td>fileFmtGrp</td>
<td>1-bit file format group</td>
</tr>
<tr>
<td></td>
<td>copy</td>
<td>1-bit copy flag</td>
</tr>
<tr>
<td></td>
<td>permWriteProtect</td>
<td>1-bit to disable/enable permanent write protection</td>
</tr>
<tr>
<td></td>
<td>tmpWriteProtect</td>
<td>1-bit to disable/enable temporary write protection</td>
</tr>
</tbody>
</table>
**SD_CardCsdObj**

<table>
<thead>
<tr>
<th>Uint16 fileFmt</th>
<th>2-bit file format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uint16 ecc</td>
<td>2-bit ECC code</td>
</tr>
<tr>
<td>Uint16 crc</td>
<td>7-bit r/w/e redundancy check</td>
</tr>
</tbody>
</table>

**Description**

Contains card specific data (CSD) for MMC cards

**Example**

None

---

**SD_CardCsdObj**  *Contains card-specific data (CSD)*

**Structure**

SD_CardCsdObj

**Members**

- Uint16 csdStructure: 2-bit structure type field
- Uint16 taac: 8-bit TAAC
- Uint16 nsac: 8-bit NSAC
- Uint16 tranSpeed: 8-bit max data transmission speed
- Uint16 ccc: 12-bit card command classes
- Uint16 readBILen: 4-bit maximum Read Block Length
- Uint16 readBIPartial: 1-bit indicates if partial read blocks allowed
- Uint16 writeBilMisalign: 1-bit flag indicates write block misalignment
- Uint16 readBilMisalign: 1-bit flag indicates read block misalignment
- Uint16 dsrlmp: 1-bit flag indicates whether card has DSR reg
- Uint16 cSize: 12-bit device size
- Uint16 vddRCurrMin: 3-bit Max. Read Current @ Vdd Min
- Uint16 vddRCurrMax: 3-bit Max. Read Current @ Vdd Max
- Uint16 vddWCurrMin: 3-bit Max. Write Current @ Vdd Min
- Uint16 vddWCurrMax: 3-bit Max. Write Current @ Vdd Max
- Uint16 cSizeMult: 3-bit device size multiplier
- Uint16 eraseBilEn: 1-bit erase single block enable
- Uint16 sectorSize: 7-bit erase group size
- Uint16 wpGrpSize: 7-bit write protect group size
- Uint16 wpGrpEnable: 1-bit write protect enable flag
- Uint16 r2wFactor: 3-bit stream write factor
- Uint16 writeBILen: 4-bit maximum write block length
MMC_CardIdObj

<table>
<thead>
<tr>
<th>Member</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>writeBIPartial</td>
<td>Uint16</td>
<td>1-bit indicates if partial write blocks allowed</td>
</tr>
<tr>
<td>fileFmtGrp</td>
<td>Uint16</td>
<td>1-bit file format group</td>
</tr>
<tr>
<td>copy</td>
<td>Uint16</td>
<td>1-bit copy flag</td>
</tr>
<tr>
<td>permWriteProtect</td>
<td>Uint16</td>
<td>1-bit to disable/enable permanent write protection</td>
</tr>
<tr>
<td>tmpWriteProtect</td>
<td>Uint16</td>
<td>1-bit to disable/enable temporary write protection</td>
</tr>
<tr>
<td>fileFmt</td>
<td>Uint16</td>
<td>2-bit file format</td>
</tr>
<tr>
<td>crc</td>
<td>Uint16</td>
<td>7-bit r/w/e redundancy check</td>
</tr>
</tbody>
</table>

**Description**
Contains card-specific data (CSD) for SD cards

**Example**
None

**MMC_CardIdObj**

*Contains Card Identification (CID)*

**Structure**
MMC_CardIdObj

**Members**

<table>
<thead>
<tr>
<th>Member</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mfgId</td>
<td>Uint32</td>
<td>24-bit Manufacturer's ID</td>
</tr>
<tr>
<td>productName</td>
<td>Char</td>
<td>8-character Product Name</td>
</tr>
<tr>
<td>hwRev</td>
<td>Uint16</td>
<td>4-bit Hardware Revision Number</td>
</tr>
<tr>
<td>fwRev</td>
<td>Uint16</td>
<td>4-bit Firmware Revision Number</td>
</tr>
<tr>
<td>serialNumber</td>
<td>Uint32</td>
<td>24-bit Serial Number</td>
</tr>
<tr>
<td>monthCode</td>
<td>Uint16</td>
<td>4-bit Manufacturing Date (Month)</td>
</tr>
<tr>
<td>yearCode</td>
<td>Uint16</td>
<td>bit Manufacturing Date (Year)</td>
</tr>
<tr>
<td>checksum</td>
<td>Uint16</td>
<td>7-bit crc</td>
</tr>
</tbody>
</table>

**Description**
Contains card identification

**Example**
None
**MMC_CardObj**

*Contains information about Memory Cards, including CID and CSD*

**Structure**

MMC_CardObj

**Members**

- Uint32 rca: User assigned relative card address (RCA)
- Uint16 status: Last read status value
- Uint16 CardIndex: MMC module assigned index for card
- 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 cid: Manufacturers Card ID
- MMC_CardCsdObj *MMC_csd: Card-specific data
- SD_CardCsdObj *SD_csd: Card either sd or mmc; we will use the appropriate csd.
- MMC_CardXCsdObj *xcsd: Extended CSD

**Description**

Contains information about memory cards, including CID and CSD.

**Example**

None

**MMC_CardXCsdObj**

*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**

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uint16</td>
<td>argh</td>
<td>High part of command argument</td>
</tr>
<tr>
<td>Uint16</td>
<td>argl</td>
<td>Low part of command argument</td>
</tr>
<tr>
<td>Uint16</td>
<td>cmd</td>
<td>MMC command</td>
</tr>
</tbody>
</table>

**Description**

Stores an MMC command

**Example**

None

## MMC_MmcRegObj

**Structure to store values of all MMC regs**

**Structure**

MMC_MmcRegObj

**Members**

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uint16</td>
<td>mmcfclk</td>
<td>MMCFCLK register</td>
</tr>
<tr>
<td>Uint16</td>
<td>mmcctl</td>
<td>MMCCTL register</td>
</tr>
<tr>
<td>Uint16</td>
<td>mmcclk</td>
<td>MMCCCLK register</td>
</tr>
<tr>
<td>Uint16</td>
<td>mmcst0</td>
<td>MMCST0 register</td>
</tr>
<tr>
<td>Uint16</td>
<td>mmcst1</td>
<td>MMCST1 register</td>
</tr>
<tr>
<td>Uint16</td>
<td>mmcie</td>
<td>MMCIIE register</td>
</tr>
<tr>
<td>Uint16</td>
<td>mmctor</td>
<td>MMCTOR register</td>
</tr>
<tr>
<td>Uint16</td>
<td>mmctod</td>
<td>MMCTOD register</td>
</tr>
<tr>
<td>Uint16</td>
<td>mmcblen</td>
<td>MMCBLEN register</td>
</tr>
<tr>
<td>Uint16</td>
<td>mmcnblk</td>
<td>MMCNBLK register</td>
</tr>
<tr>
<td>Uint16</td>
<td>mmdrr</td>
<td>MMCDRR register</td>
</tr>
<tr>
<td>Uint16</td>
<td>mmcdxr</td>
<td>MMCDXR register</td>
</tr>
<tr>
<td>Uint16</td>
<td>mmccmd</td>
<td>MMCCMD register</td>
</tr>
<tr>
<td>Uint16</td>
<td>mmccargl</td>
<td>MMCARGL register</td>
</tr>
<tr>
<td>Uint16</td>
<td>mmccargh</td>
<td>MMCARGH register</td>
</tr>
<tr>
<td>MMC_RspRegObj</td>
<td>mmcrsp</td>
<td>MCRSP registers</td>
</tr>
<tr>
<td>Uint16</td>
<td>mmcdrsp</td>
<td>MCDRSP register</td>
</tr>
<tr>
<td>Uint16</td>
<td>mmccidx</td>
<td>MCCIDX register</td>
</tr>
</tbody>
</table>

**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
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**

```c
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**

```c
MMC_config(myMMC, &myMMCCfg);
```

**MMC_dispatch0**  
*ISR dispatch function to service the MMC0 isrs*

**Function**

```c
void MMC_dispatch0(
);
```

**Arguments**

None

**Description**

Interrupt service routine dispatch function to service interrupts that occur on MMC port 0.

**Example**

```c
MMC_dispatch0();
```

**MMC_dispatch1**  
*ISR dispatch function to service the MMC1 isrs*

**Function**

```c
void MMC_dispatch1(
);
```

**Arguments**

None

**Description**

Interrupt service routine dispatch function to service interrupts that occur on MMC port 1.

**Example**

```c
MMC_dispatch1();
```
### MMC_drrdy

**Function**

```c
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**

```c
MMC_Handle myMmc;
int i;
.
.
.
i = MMC_drrdy(myMmc);
```

### MMC_dxrdy

**Function**

```c
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**

```c
MMC_Handle myMmc;
int i;
.
.
.
i = MMC_dxrdy(myMmc);
```
**MMC_getCardCSD**

**Reads card specific data from response registers**

**Function**

```c
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**

```c
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**

```c
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**

```c
MMC_Handle myMmc;
MMC_CardIdObj myCardId;
myMmc = MMC_open(MMC_DEV1);
...
MMC_getCardId(myMmc,&myCardId);
```
**MMC_getConfig**

*Returns the current contents of the MMC control 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_getStatus

**Returns the status of a specified field in the status register**

**Function**

```c
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**

```c
MMC_Handle myMmc;
Uint16 ready;

read = MMC_getStatus(myMmc, MMC_ST0_DXRDY);
```

MMC_open

**Reserves the MMC device as specified by a device**

**Function**

```c
MMC_Handle MMC_open(
    Uint16 device,
);
```

**Arguments**

- `device` Device (port) number. It can be one of the following:
  - MMC_DEV0
  - MMC_DEV1

**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**

```c
MMC_Handle myMmc;

myMmc = MMC_open(MMC_DEV0);
```
## MMC_read

**Reads a block of data from a pre-selected memory card**

### Function

```c
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 bytes 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

```c
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

```c
int MMC_responseDone(
    MMCHandle 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

```c
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**

```c
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**

```c
MMC_Handle myMmc;
.
.
MMC_saveStatus(myMmc);
```

MMC_selectCard

**Selects card with specified relative address for communication**

**Function**

```c
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**

```c
MMC_InitObj myMmcInit;
MMC_Handle myMmc;
MMC_CardObj card;
Uint16 rca = 2;

myMmc = MMC_open(MMC_DEV1);

MMC_selectCard(myMmc, &card);
```
### MMC_sendAllCID

**Function**

```c
void MMC_sendAllCID(
    MMC_Handle mmc,
    MMC_CardIdObj *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**

```c
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**

```c
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.

**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**

```c
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**

```c
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**

```c
MMC_Handle myMmc;

MMC_sendCSD(myMmc);
```
### MMC_sendGoIdle

**Function**

```c
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**

```c
MMC_Handle myMmc;
MMC_sendGoIdle(myMmc);
```

---

### MMC_setCardPtr

**Function**

```c
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**

```c
MMC_Handle myMmc;
MMC_cardObj *card;
MMC_setCardPtr(myMmc, &card);
```
MMC_sendOpCond

Sends the SEND_OP_COND command to a card

Function

```c
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.

Table 14-4. OCR Register Definitions

<table>
<thead>
<tr>
<th>OCR Bit</th>
<th>VDD Voltage Window</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-7</td>
<td>Reserved</td>
</tr>
<tr>
<td>8</td>
<td>2.0-2.1</td>
</tr>
<tr>
<td>9</td>
<td>2.1-2.2</td>
</tr>
<tr>
<td>10</td>
<td>2.2-2.3</td>
</tr>
<tr>
<td>11</td>
<td>2.3-2.4</td>
</tr>
<tr>
<td>12</td>
<td>2.4-2.5</td>
</tr>
<tr>
<td>13</td>
<td>2.5-2.6</td>
</tr>
<tr>
<td>14</td>
<td>2.6-2.7</td>
</tr>
<tr>
<td>15</td>
<td>2.7-2.8</td>
</tr>
<tr>
<td>16</td>
<td>2.8-2.9</td>
</tr>
<tr>
<td>17</td>
<td>2.9-3.0</td>
</tr>
<tr>
<td>18</td>
<td>3.0-3.1</td>
</tr>
<tr>
<td>19</td>
<td>3.1-3.2</td>
</tr>
<tr>
<td>20</td>
<td>3.2-3.3</td>
</tr>
<tr>
<td>21</td>
<td>3.3-3.4</td>
</tr>
<tr>
<td>22</td>
<td>3.4-3.5</td>
</tr>
<tr>
<td>23</td>
<td>3.5-3.6</td>
</tr>
<tr>
<td>24-30</td>
<td>reserved</td>
</tr>
<tr>
<td>31</td>
<td>Card power-up status bit (busy)</td>
</tr>
</tbody>
</table>

Example

```c
MMC_Handle myMmc;
```
MMC_setCallBack

**Associates functions to interrupts and installs dispatcher routines**

**Function**
```c
void MMC_setCallBack(
    MMC_Handle mmc,
    MMC_callBackObj *callbackfuncs
);
```

**Arguments**
- mmc: MMC_Handle returned from a call to MMC_open
- 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.

**Example**
```c
MMC_Handle myMmc;
MMC_callBackObj *callback;

MMC_setCallBack(myMmc, &callback);
```

MMC_setRca

**Sets the relative card address of an attached memory card**

**Function**
```c
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**
```c
MMC_Handle myMmc;
```
MMC_stop

```c
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**
```c
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**
```c
MMC_Handle myMmc;
.
.
MMC_stop(myMmc);
```

MMC_waitForFlag

**Function**
```c
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**
```c
MMC_Handle myMmc;
.
.
MMC_waitForFlag(myMmc, 0x0100);
```
**MMC_write**

*Writes a block of data to a pre-selected memory card*

**Function**

```c
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 bytes to store in buffer.

**Description**

 Writes a block of data to the pre-selected memory card.

**Example**

```c
MMC_Handle myMmc;
Uint16 mybuf[512];

myMmc = MMC_open(MMC_DEV1);
.
.
.
MMC_write(myMmc, 0, mybuf, 512);
```
### MMC_intEnable

**Function**

```c
void MMC_intEnable(
    MMC_Handle mmc,
    Uint16 enableMask
);
```

**Arguments**

- `mmc`: MMC Handle returned by call to MMC_open
- `enableMask`: 16-bit value to be written to the MMCIE. A bit value of 1 enables an interrupt while a bit value of 0 resets it

**Description**

Enables interrupts by writing to the MMCIE register. The functions that service MMC events need to be associated to the interrupts using the MMC_setCallBack before calling this function.

**Example**

```c
MMC_Handle myMmc;
Uint16 enableMask;
MMC_callBackObj callback;
...
myMmc = MMC_open(MMC_DEV1);
.
.
MMC_setCallBack(myMmc, &callback);
MMC_intEnable(myMmc, 0x200);
```
### MMC_setCardType
*Writes the card type (MMC ro SD) to the MMC_CardObj structure*

**Function**
```c
void MMC_setCardType(
    MMC_CardObj *card,
    Uint16 type
);
```

**Arguments**
- *card*: Pointer to card obj for the controller
- *type*: Card type (MMC_CARD/SD_CARD) returned by the MMC_sendOpCond function

**Description**
Sets the card type in the card obj for later reference

**Example**
```c
MMC_CardObj *card;
Uint16 type;
.
.
type = MMC_sendOpCond(myMmc,0x00100000);
/* Returns either MMC_CARD or SD_CARD */
.
.
MMC_setCardType(card, type);
```

### SD_sendAllCID
*Sends broadcast command to SD cards to identify themselves*

**Function**
```c
int SD_sendAllCID(
    MMC_Handle sd,
    MMC_CardIdObj *cid
);
```

**Arguments**
- *sd*: MMC Handle returned by call to MMC_open
- *cid*: Pointer to card ID object

**Description**
Sends the MMC_SEND_ALL_CID command to initiate identification of all SDmemory 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**
```c
MMC_Handle mySD;
MMC_CardIdObj myCardId;
mySD = MMC_open(MMC_DEV1);
.
.
SD_sendAllCID(mySD, &myCardId);
```
SD_getCardId

**SD_getCardId**  *Reads SD-specific card ID from MMC response registers*

**Function**

```c
void SD_getCardID(
    MMC_Handle sd,
    MMC_CardIdObj *cid
);
```

**Arguments**

- **sd**  SD handle returned by call to MMC_open
- **cid**  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**

```c
MMC_Handle mySD;
MMC_CardIdObj myCardId;
mySD = MMC_open(MMC_DEV1);
...
MMC_getCardId(mySD, &myCardId);
```

SD_getCardCsd

**SD_getCardCsd**  *Reads SD card-specific data from response registers*

**Function**

```c
void SD_getCardCsd(
    MMC_Handle sd,
    SD_CardCsdObj *csd
);
```

**Arguments**

- **sd**  SD 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**

```c
MMC_Handle mySD;
MMC_CardCsd Obj *csd;
...
...
MMC_sendCSD(mySD);
SD_getCardCSD(mySD, csd);
```
**SD_sendRca**  
*Asks the SD card to respond with its relative card address*

**Function**

```c
int SD_sendRca(
    MMC_Handle sd,
    MMC_CardObj *card
);
```

**Arguments**

- `sd`  SD handle returned by call to MMC_open
- `card` Pointer to card object

**Description**

The host requests the SD card to set its Relative Card Address and send it to the host once done. This RCA will be used for all future communication with the card.

**Example**

```c
MMC_Handle mySD;
MMC_CardObj *card;
.
.
SD_sendRca(mySD, card);
```

**SD_setWidth**  
*Sets the data bus width to either 1 bit or 4 bits*

**Function**

```c
int SD_setWidth(
    MMC_Handle sd,
    Uint16 width
);
```

**Arguments**

- `sd`  SD Handle returned by call to MMC_open
- `width` Value to set bus width for data transfer from/to the card. Width of 0x1 sets the bus width to 1 bit and a width of 0x4 sets it to 4–bits.

**Description**

The SD card supports a 4-bit width data transfer. The bus width can be set after the card is selected using the MMC_selCard API.

**Example**

```c
MMC_Handle mySD;
Uint16 retVal;
MMC_CardObj *card;
.
.
mySD = MMC_open(MMC_DEV1);
.
.
retVal = MMC_selectCard(mySD, card);
retVal = SD_setWidth(mySD, 0x4);
```
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.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.1 Overview</td>
<td>15-2</td>
</tr>
<tr>
<td>15.2 Configuration Structures</td>
<td>15-4</td>
</tr>
<tr>
<td>15.3 Functions</td>
<td>15-5</td>
</tr>
<tr>
<td>15.4 Macros</td>
<td>15-7</td>
</tr>
</tbody>
</table>
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 ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PLL_Config</td>
<td>PLL configuration structure used to set up the PLL interface</td>
<td>15-4</td>
</tr>
</tbody>
</table>

**Table 15–2. PLL Functions**

| Syntax     | Description                                           | See page ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PLL_config()</td>
<td>Sets up PLL using configuration structure (PLL_Config)</td>
<td>15-5</td>
</tr>
<tr>
<td>PLL_setFreq()</td>
<td>Initializes the PLL to produce the desired CPU (core)/Fast peripherals/Slow peripherals/EMIF output frequency</td>
<td>15-6</td>
</tr>
</tbody>
</table>
Table 15–3. PLL Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLKMD</td>
<td>PLLENABLE, PLLDIV, PLLMULT, VCOONOFF</td>
</tr>
<tr>
<td>PLLCSR</td>
<td>PLLEN, PLLPWRDN, OSCPWRDN, PLLRST, LOCK, STABLE</td>
</tr>
<tr>
<td>PLLM</td>
<td>PLLM</td>
</tr>
<tr>
<td>PLLDIV0</td>
<td>PLLDIV0, D0EN</td>
</tr>
<tr>
<td>PLLDIV1</td>
<td>PLLDIV1, D1EN</td>
</tr>
<tr>
<td>PLLDIV2</td>
<td>PLLDIV2, D2EN</td>
</tr>
<tr>
<td>PLLDIV3</td>
<td>PLLDIV3, D3EN</td>
</tr>
<tr>
<td>OSCDIV1</td>
<td>OSCDIV1, OD1EN</td>
</tr>
<tr>
<td>WAKEUP</td>
<td>WKEN0, WKEN1, WKEN2, WKEN3</td>
</tr>
<tr>
<td>CLKMD</td>
<td>CLKMD0</td>
</tr>
<tr>
<td>CLKOUTSR</td>
<td>CLKOUTDIS, CKOSEL</td>
</tr>
</tbody>
</table>

**For C5502 and C5501**

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 and 5501 device:
- 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 osccdiv1 // 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**

```c
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**

```c
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*/
   
   ```c
   PLL_Config MyConfig = {
       1, /* iai */
       1, /* iab */
       31, /* pllmult */
       3 /* div */
   };
   PLL_config(&MyConfig);
   ```

2. /* Using PLL_config function and PLL_Config structure for 5502/5501 PLL*/
   
   ```c
   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 and C5501 device):
void PLL_setFreq (Uint16 mode, Uint16 mul, Uint16 div0, Uint16 div1, Uint16 div2, Uint16 div3, Uint16 oscdiv);

**Arguments**

- Uint16 mode  // PLL mode  
  //PLL_PLLCSR_PLLLEN_BYPASS_MODE  
  //PLL_PLLCSR_PLLLEN_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 5502/5501 */  
   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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLL_RGET()</td>
<td>Uint16 PLL_RGET(REG)</td>
</tr>
<tr>
<td>PLL_RSET()</td>
<td>Void PLL_RSET(REG, Uint16 regval)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLL_FGET()</td>
<td>Uint16 PLL_FGET(REG, FIELD)</td>
</tr>
<tr>
<td>PLL_FSET()</td>
<td>Void PLL_FSET(REG, FIELD, Uint16 fieldval)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLL_REG_RMK()</td>
<td>Uint16 PLL_REG_RMK(fieldval_n,…fieldval_0)</td>
</tr>
<tr>
<td>PLL_FMK()</td>
<td>Uint16 PLL_FMK(REG, FIELD, fieldval)</td>
</tr>
</tbody>
</table>

Note: *Start with field values with most significant field positions:
field_n: MSB field
field_0: LSB field
*only writable fields allowed

(d) Macros to read a register address

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLL_ADDR()</td>
<td>Uint16 PLL_ADDR(REG)</td>
</tr>
</tbody>
</table>

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).
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.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.1 Overview</td>
<td>16-2</td>
</tr>
<tr>
<td>16.2 Functions</td>
<td>16-3</td>
</tr>
<tr>
<td>16.3 Macros</td>
<td>16-4</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>Table 16–1. PWR Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functions</td>
</tr>
<tr>
<td>PWR_powerDown</td>
</tr>
</tbody>
</table>

16.1.1 PWR Registers

<table>
<thead>
<tr>
<th>Table 16–2. PWR Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
</tr>
<tr>
<td>Only for C5509 and C5510</td>
</tr>
<tr>
<td>ICR</td>
</tr>
<tr>
<td>ISTR</td>
</tr>
<tr>
<td>Only for C5502 and C5501</td>
</tr>
<tr>
<td>ICR</td>
</tr>
<tr>
<td>ISTR</td>
</tr>
<tr>
<td>PICR</td>
</tr>
<tr>
<td>PISTR</td>
</tr>
<tr>
<td>MICR</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>PWR_RGET()</td>
<td>Uint16 PWR_RGET(REG)</td>
</tr>
<tr>
<td>PWR_RSET()</td>
<td>Void PWR_RSET(REG, Uint16 regval)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>PWR_FGET()</td>
<td>Uint16 PWR_FGET(REG, FIELD)</td>
</tr>
<tr>
<td>PWR_FSET()</td>
<td>Void PWR_FSET(REG, FIELD, Uint16 fieldval)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>PWR_REG_RMK()</td>
<td>Uint16 PWR_REG_RMK(fieldval_n,...,fieldval_0)</td>
</tr>
<tr>
<td>Note: *Start with field values with most significant field positions:</td>
<td></td>
</tr>
<tr>
<td>field_n: MSB field</td>
<td></td>
</tr>
<tr>
<td>field_0: LSB field</td>
<td></td>
</tr>
<tr>
<td>*only writable fields allowed</td>
<td></td>
</tr>
<tr>
<td>PWR_FMK()</td>
<td>Uint16 PWR_FMK(REG, FIELD, fieldval)</td>
</tr>
</tbody>
</table>

(d) Macros to read a register address

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>PWR_ADDR()</td>
<td>Uint16 PWR_ADDR(REG)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.1 Overview</td>
<td>17-2</td>
</tr>
<tr>
<td>17.2 Configuration Structures</td>
<td>17-6</td>
</tr>
<tr>
<td>17.3 API Reference</td>
<td>17-9</td>
</tr>
<tr>
<td>17.4 Macros</td>
<td>17-16</td>
</tr>
</tbody>
</table>
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**

<table>
<thead>
<tr>
<th>Configuration Structure</th>
<th>Description</th>
<th>See page</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTC_Alarm</td>
<td>Structure used to set RTC Time</td>
<td>17-6</td>
</tr>
<tr>
<td>RTC_Config</td>
<td>RTC register Configuration Structure</td>
<td>17-7</td>
</tr>
<tr>
<td>RTC_Date</td>
<td>Structure used to set RTC Calendar</td>
<td>17-7</td>
</tr>
<tr>
<td>RTC_IsrAddr</td>
<td>Structure to set the RTC callback function</td>
<td>17-8</td>
</tr>
<tr>
<td>RTC_Time</td>
<td>Structure used to set RTC Alarm Time</td>
<td>17-8</td>
</tr>
</tbody>
</table>

Table 17–2.  **RTC Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>See page</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTC_bcdToDec</td>
<td>Changes BCD value to a hexadecimal value</td>
<td>17-9</td>
</tr>
<tr>
<td>RTC_config</td>
<td>Writes value to initialize RTC using the RTC register Configuration Structure</td>
<td>17-9</td>
</tr>
<tr>
<td>RTC_decToBcd</td>
<td>Changes decimal value to BCD value</td>
<td>17-9</td>
</tr>
<tr>
<td>RTC_eventDisable</td>
<td>Disables interrupt event specified by the argument</td>
<td>17-10</td>
</tr>
<tr>
<td>RTC_eventEnable</td>
<td>Enables RTC interrupt event specified by an argument</td>
<td>17-10</td>
</tr>
<tr>
<td>RTC_getConfig</td>
<td>Reads the RTC registers into the RTC register Configuration Structure</td>
<td>17-10</td>
</tr>
<tr>
<td>RTC_getDate</td>
<td>Reads current date from RTC Registers</td>
<td>17-11</td>
</tr>
<tr>
<td>RTC_getEventId</td>
<td>Obtains IRQ module event ID for RTC</td>
<td>17-11</td>
</tr>
<tr>
<td>RTC_getTime</td>
<td>Reads current time from RTC Registers, in a 24-hour format</td>
<td>17-11</td>
</tr>
<tr>
<td>RTC_reset</td>
<td>Sets the RTC register to the default (power-on) values</td>
<td>17-12</td>
</tr>
<tr>
<td>RTC_setAlarm</td>
<td>Sets alarm to a specific time</td>
<td>17-12</td>
</tr>
<tr>
<td>RTC_setCallback</td>
<td>Associates each function to one of the RTC interrupts</td>
<td>17-13</td>
</tr>
<tr>
<td>RTC_setDate</td>
<td>Sets RTC Calendar</td>
<td>17-13</td>
</tr>
<tr>
<td>RTC_setPeriodicInterval</td>
<td>Sets periodic interrupt rate</td>
<td>17-14</td>
</tr>
<tr>
<td>RTC_setTime</td>
<td>Sets time registers</td>
<td>17-14</td>
</tr>
</tbody>
</table>
### Table 17–2. RTC Functions (Continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>See page</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTC_start</td>
<td>Instructs the RTC to begin running</td>
<td>17-15</td>
</tr>
<tr>
<td>RTC_stop</td>
<td>Stops the RTC</td>
<td>17-15</td>
</tr>
</tbody>
</table>

### Table 17–3. RTC ANSI C-Style Time Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTCasctime</td>
<td>Converts a time to an ASCII string</td>
</tr>
<tr>
<td>RTCctime</td>
<td>Converts calendar time to local time</td>
</tr>
<tr>
<td>RTCdifftime</td>
<td>Returns the difference between two calendar times</td>
</tr>
<tr>
<td>RTCgmtime</td>
<td>Converts calendar time to GMT</td>
</tr>
<tr>
<td>RTClocaltime</td>
<td>Converts calendar time to local time</td>
</tr>
<tr>
<td>RTCmktime</td>
<td>Converts local time to calendar time</td>
</tr>
<tr>
<td>RTCstrftime</td>
<td>Formats a time into a character string</td>
</tr>
<tr>
<td>RTCtime</td>
<td>Returns the current RTC time and date</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
<th>See page</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTC_Addr</td>
<td>Reads register address</td>
<td>17-16</td>
</tr>
<tr>
<td>RTC_FGET</td>
<td>Reads RTC register field values</td>
<td>17-16</td>
</tr>
<tr>
<td>RTC_FSET</td>
<td>Writes RTC register field values</td>
<td>17-16</td>
</tr>
<tr>
<td>RTC_REG_FMk</td>
<td>Creates value of RTC register fields</td>
<td>17-16</td>
</tr>
<tr>
<td>RTC_REG_RMk</td>
<td>Creates value of RTC registers</td>
<td>17-17</td>
</tr>
<tr>
<td>RTC_RGET</td>
<td>Reads RTC register values</td>
<td>17-17</td>
</tr>
<tr>
<td>RTC_RSet</td>
<td>Writes RTC register values</td>
<td>17-17</td>
</tr>
</tbody>
</table>
### Table 17–5. Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTCSEC</td>
<td>SEC</td>
</tr>
<tr>
<td>RTCSECA</td>
<td>SAR</td>
</tr>
<tr>
<td>RTCMIN</td>
<td>MIN</td>
</tr>
<tr>
<td>RTCMINA</td>
<td>MAR</td>
</tr>
<tr>
<td>RTOCHOUR</td>
<td>HR, AMPM</td>
</tr>
<tr>
<td>RTCOURLRA</td>
<td>HAR, AMPM</td>
</tr>
<tr>
<td>RTCWDAYW</td>
<td>DAY, DAEN, DAR</td>
</tr>
<tr>
<td>RTCDAYM</td>
<td>DATE</td>
</tr>
<tr>
<td>RTCMONTH</td>
<td>MONTH</td>
</tr>
<tr>
<td>RTCYEAR</td>
<td>YEAR</td>
</tr>
<tr>
<td>RTCPINTR</td>
<td>RS, (R)UIP</td>
</tr>
<tr>
<td>RTCINTEN</td>
<td>TM, UIE, AIE, PIE, SET</td>
</tr>
<tr>
<td>RTCINTFL</td>
<td>UF, AF, PF, (R)IRQF</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTC_Alarm</td>
<td>Structure used to set the RTC time. After it is created and initialized, the</td>
</tr>
<tr>
<td></td>
<td>structure is passed to the RTC_setAlarm() function. The values of the structure</td>
</tr>
<tr>
<td></td>
<td>must be entered in BCD format. You can use the RTC_decToBcd() and RTC_bcdToDec()</td>
</tr>
<tr>
<td></td>
<td>functions to switch between decimal and BCD values.</td>
</tr>
</tbody>
</table>

#### 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.
### RTC_Config

**RTC configuration structure**

<table>
<thead>
<tr>
<th>Structure</th>
<th>RTC_Config</th>
</tr>
</thead>
<tbody>
<tr>
<td>Members</td>
<td></td>
</tr>
<tr>
<td>Uint16 rtcsec</td>
<td>Seconds Register</td>
</tr>
<tr>
<td>Uint16 rtcseca</td>
<td>Seconds Alarm Register</td>
</tr>
<tr>
<td>Uint16 rtcm</td>
<td>Minutes Register</td>
</tr>
<tr>
<td>Uint16 rtcm</td>
<td>Minutes Alarm Register</td>
</tr>
<tr>
<td>Uint16 rthour</td>
<td>Hour Register</td>
</tr>
<tr>
<td>Uint16 rthour</td>
<td>Hour Alarm Register</td>
</tr>
<tr>
<td>Uint16 rtcdyw</td>
<td>Day of the Week and Day Alarm Register</td>
</tr>
<tr>
<td>Uint16 rtcdyw</td>
<td>Day of the Month (Date) Register</td>
</tr>
<tr>
<td>Uint16 rtm</td>
<td>Month Register</td>
</tr>
<tr>
<td>Uint16 rtcyear</td>
<td>Year Register</td>
</tr>
<tr>
<td>Uint16 rtc</td>
<td>Periodic Interrupt selection Register</td>
</tr>
<tr>
<td>Uint16 rtcinten</td>
<td>Interrupt Enable Register</td>
</tr>
</tbody>
</table>

**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**

<table>
<thead>
<tr>
<th>Structure</th>
<th>RTC_Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Members</td>
<td></td>
</tr>
<tr>
<td>Uint16 year</td>
<td>Current year (Range: 0x00–0x99 for BCD)</td>
</tr>
<tr>
<td>Uint16 month</td>
<td>Current month (Range: 0x01-0x12 for BCD)</td>
</tr>
<tr>
<td>Uint16 daym</td>
<td>Day of the month, or date (Range: 0x01-0x31 for BCD)</td>
</tr>
<tr>
<td>Uint16 dayw</td>
<td>Day of the week (Range 1–7, where Sunday is 1)</td>
</tr>
</tbody>
</table>

**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_Time

**Structure used to set the RTC callback function**

<table>
<thead>
<tr>
<th>Structure</th>
<th>Members</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTC_IsrAddr</td>
<td>void (*periodicAddr)(void) Pointer to the function called when a periodic interrupt occurs.</td>
<td>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 <code>interrupt</code> keyword.</td>
</tr>
<tr>
<td></td>
<td>void (*alarmAddr)(void) Pointer to the function called when an alarm interrupt occurs.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>void (*updateAddr)(void) Pointer to the function called when an update interrupt occurs.</td>
<td></td>
</tr>
</tbody>
</table>

### RTC_Time

**Structure used to set RTC time**

<table>
<thead>
<tr>
<th>Structure</th>
<th>Members</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTC_Time</td>
<td>Uint16 hour Current time (Range: 0x00–0x23 for BCD, for 24-hour format. 12-hour format is not supported.)</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>Uint16 minute Current Minute (Range: 0x00–0x59 for BCD)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Uint16 second Second (Range: 0x00–0x59 for BCD)</td>
<td></td>
</tr>
</tbody>
</table>
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
RTC_eventDisable

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**

```c
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**

```c
RTC_Date getDate;
RTC_getDate(&getDate);
```

### RTC_getEventId

**Obtains IRQ module event ID for RTC**

**Function**

```c
int RTC_getEventID()
```

**Arguments**

None

**Description**

Obtains IRQ module event ID for RTC

**Example**

```c
int id;
id = RTC_getEventId();
```

### RTC_getTime

**Reads current time from RTC registers, in 24-hour format**

**Function**

```c
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**

```c
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**

```c
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**

```c
RTC_Alarm myPeriodicAlarm = {
    0x1, /* alHour, in 24-hour format */
    DONTCARE, /* alMinutes */
    0x0, /* alSeconds */
    0x2, /* alDayw */
};

RTC_setAlarm(&myPeriodicAlarm);
/* 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**  
```c
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**  
```c
RTC_Date myDate = {
    0x01, /* Year 2001 */
    0x05, /* Month May*/
    0x10, /* Day of month */
    0x05 /* Day of week Thursday */
};
RTC_setDate(&myDate);
```
RTC_setPeriodicInterval

**RTC_setPeriodicInterval**  *Sets periodic interrupt rate*

**Function**

```c
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**

```c
RTC_setPeriodicInterval(RTC_RATE_122us);
```

---

RTC_setTime

**RTC_setTime**  *Sets time registers, in 24-hour format*

**Function**

```c
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**

```c
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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uint16 RTC_ADDR(REG)</td>
<td>Reads register address</td>
<td>x = RTC_ADDR(RTCSEC);</td>
</tr>
</tbody>
</table>

RTC_FGET

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uint16 RTC_FGET(REG, FIELD)</td>
<td>Reads RTC register field values. This is applicable only to registers with more than one field.</td>
<td>x = RTC_FGET(RTCDAYW, DAEN);</td>
</tr>
</tbody>
</table>

RTC_FSET

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Void RTC_FSET(REG, FIELD, Uint16 fieldval)</td>
<td>Writes RTC register field values. This is applicable only to registers with more than one field.</td>
<td>x = RTC_FSET(RTCDAYW, DAEN, x);</td>
</tr>
</tbody>
</table>

RTC_REG_FMK

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uint16 RTC_REG_FMK(FIELD, Uint16 fieldval)</td>
<td>Creates value of RTC register fields (only for registers with more than one field).</td>
<td>x = RTC_FM(FIELD, DAY, v);</td>
</tr>
</tbody>
</table>
**RTC_REG_RMK**

*Creates value of RTC registers*

**Macro**

Uint16 RTC_REG_RMK(Uint16 fieldval_n, 0, Uint16 fieldval_0)

**Arguments**

<table>
<thead>
<tr>
<th>Reg</th>
<th>FIELD</th>
<th>regval</th>
<th>fieldval</th>
</tr>
</thead>
<tbody>
<tr>
<td>REG</td>
<td>FIELD</td>
<td>regval</td>
<td>Value to write in the register REG</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>fieldval</td>
</tr>
</tbody>
</table>

**Description**

Creates value of RTC registers (only for registers with more than one field).

**Example**

Uint16 x, field1, field2, field3;

\[
x = \text{RTC_RTMAYW_RMK(field1, field2, field3)};
\]

**RTC_RGET**

*Reads RTC register values*

**Macro**

Uint16 RTC_RGET(REG)

**Description**

Reads RTC register values

**Example**

Uint16 x;

\[
x = \text{RTC_RGET(RTCSEC)};
\]

**RTC_RSET**

*Writes RTC register values*

**Macro**

Void RTC_RSET(REG, Uint16 regval)

**Description**

Writes RTC register values

**Example**

Uint16 x = 0x15;

\[
\text{RTC_RSET(RTCSEC, x)};
\]
This chapter describes the TIMER module, lists the API structure, functions and macros within the module, and provides a TIMER API reference section.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.1 Overview</td>
<td>18-2</td>
</tr>
<tr>
<td>18.2 Configuration Structures</td>
<td>18-3</td>
</tr>
<tr>
<td>18.3 Functions</td>
<td>18-4</td>
</tr>
<tr>
<td>18.4 Macros</td>
<td>18-9</td>
</tr>
</tbody>
</table>
### 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 includes descriptions for available TIMER macros.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
<th>See page ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMER_Config</td>
<td>TIMER configuration structure used to setup the TIMER_config() function</td>
<td>18-3</td>
</tr>
</tbody>
</table>

#### Table 18–2. TIMER Functions

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
<th>See page ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMER_close()</td>
<td>Closes the TIMER and its corresponding handler</td>
<td>18-4</td>
</tr>
<tr>
<td>TIMER_config()</td>
<td>Sets up TIMER using configuration structure (TIMER_Config)</td>
<td>18-4</td>
</tr>
<tr>
<td>TIMER_getConfig()</td>
<td>Reads the TIMER configuration</td>
<td>18-5</td>
</tr>
<tr>
<td>TIMER_getEventId()</td>
<td>Obtains IRQ event ID for TIMER device</td>
<td>18-5</td>
</tr>
<tr>
<td>TIMER_open()</td>
<td>Opens the TIMER and assigns a handler to it</td>
<td>18-6</td>
</tr>
<tr>
<td>TIMER_reset()</td>
<td>Resets the TIMER registers with default values</td>
<td>18-7</td>
</tr>
<tr>
<td>TIMER_start()</td>
<td>Starts the TIMER device running</td>
<td>18-7</td>
</tr>
<tr>
<td>TIMER_stop()</td>
<td>Stops the TIMER device running</td>
<td>18-7</td>
</tr>
<tr>
<td>TIMER_tintoutCfg()</td>
<td>Sets up the TIMER Polarity pin along with settings for the FUNC, PWID, CP fields in the TCR register</td>
<td>18-8</td>
</tr>
</tbody>
</table>

#### Table 18–3. Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCR</td>
<td>IDLEEN, (R)INTEXT, (R)ERRTIM, FUNC, TLB, SOFT, FREE, PWID, ARB, TSS, CP, POLAR, DATOUT</td>
</tr>
<tr>
<td>PRD</td>
<td>PRD</td>
</tr>
<tr>
<td>TIM</td>
<td>TIM</td>
</tr>
<tr>
<td>PRSC</td>
<td>PSC, TDDR</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Structure</th>
<th>TIMER configuration structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Members</td>
<td>Uint16 tcr       Timer Control Register</td>
</tr>
<tr>
<td></td>
<td>Uint16 prd       Period Register</td>
</tr>
<tr>
<td></td>
<td>Uint16 prsc      Timer Pre-scaler Register</td>
</tr>
</tbody>
</table>

**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**
```c
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

```c
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

```c
TIMER_close(hTimer);
```

**TIMER_config**  
*Writes value to TIMER using configuration structure*

Function

```c
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

```c
TIMER_Config MyConfig = {
    0x0010, /* tcr */
    0xFFFF, /* prd */
    0xF0F0  /* prsc */
};

TIMER_config(hTimer,&MyConfig);
```
**TIMER_getConfig**

*Reads the TIMER configuration*

**Function**

```c
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**

```c
TIMER_Config MyConfig;
TIMER_getConfig(hTimer,&MyConfig);
```

---

**TIMER_getEventId**

*Obtains IRQ event ID for TIMER device*

**Function**

```c
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**

```c
Uint16 TimerEventId;
TimerEventId = TIMER_getEventId(hTimer);
IRQ_enable(TimerEventId);
```
**TIMER_open**  

*Opens TIMER for TIMER calls*

**Function**

```c
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**

```c
TIMER_Handle hTimer;
...
hTimer = TIMER_open(TIMER_DEV0, 0);
```
## TIMER_reset

### Resets TIMER

**Function**

```c
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**

```c
TIMER_reset(hTimer);
```

## TIMER_start

### Starts TIMER device running

**Function**

```c
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**

```c
TIMER_start(hTimer);
```

## TIMER_stop

### Stops TIMER device running

**Function**

```c
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**

```c
TIMER_stop(hTimer);
```
**TIMER_tintoutCfg**

*Configures TINT/TOUT pin*

**Function**

```c
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**

```c
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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMER_RGET()</td>
<td>Uint16 TIMER_RGET(REG#)</td>
</tr>
<tr>
<td>TIMER_RSET()</td>
<td>Void TIMER_RSET(REG#, Uint16 regval)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMER_FGET()</td>
<td>Uint16 TIMER_FGET(REG#, FIELD)</td>
</tr>
<tr>
<td>TIMER_FSET()</td>
<td>Void TIMER_FSET(REG#, FIELD, Uint16 fieldval)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMER_REG_RMK()</td>
<td>Uint16 TIMER_REG_RMK(fieldval_n,...fieldval_0)</td>
</tr>
<tr>
<td>Note:</td>
<td>*Start with field values with most significant field positions: field_n: MSB field field_0: LSB field *only writable fields allowed</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>TIMER_FMK()</td>
<td>Uint16 TIMER_FMK(REG, FIELD, fieldval)</td>
</tr>
</tbody>
</table>

(d) Macros to read a register address

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMER_ADDR()</td>
<td>Uint16 TIMER_ADDR(REG#)</td>
</tr>
</tbody>
</table>

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_FGET 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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMER_RGETH()</td>
<td>Uint16 TIMER_RGETH(TIMER_Handle hTimer, REG)</td>
</tr>
<tr>
<td>TIMER_RSETH()</td>
<td>Void TIMER_RSETH(</td>
</tr>
<tr>
<td></td>
<td>TIMER_Handle hTimer,</td>
</tr>
<tr>
<td></td>
<td>REG,</td>
</tr>
<tr>
<td></td>
<td>Uint16 regval</td>
</tr>
<tr>
<td></td>
<td>)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMER_FGETH()</td>
<td>Uint16 TIMER_FGETH(TIMER_Handle hTimer, REG, FIELD)</td>
</tr>
<tr>
<td>TIMER_FSETH()</td>
<td>Void TIMER_FSETH(</td>
</tr>
<tr>
<td></td>
<td>TIMER_Handle hTimer,</td>
</tr>
<tr>
<td></td>
<td>REG,</td>
</tr>
<tr>
<td></td>
<td>FIELD,</td>
</tr>
<tr>
<td></td>
<td>Uint16 fieldval</td>
</tr>
</tbody>
</table>

#### (c) Macros to read a register address

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMER_ADDRH()</td>
<td>Uint16 TIMER_ADDRH(TIMER_Handle hTimer, REG)</td>
</tr>
</tbody>
</table>

**Notes:**

1) **REG** indicates the registers: TCR, PRD, TIM, and PRSC
2) **FIELD** indicates the register field name as specified in the 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**).
This chapter describes the UART module, lists the API structure, functions, and macros within the module, and provides a UART API reference section.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>19.1 Overview</td>
<td>19-2</td>
</tr>
<tr>
<td>19.2 Configuration Structures</td>
<td>19-5</td>
</tr>
<tr>
<td>19.3 Functions</td>
<td>19-8</td>
</tr>
<tr>
<td>19.4 Macros</td>
<td>19-14</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Structure</th>
<th>Type</th>
<th>Purpose</th>
<th>See page</th>
</tr>
</thead>
<tbody>
<tr>
<td>UART_Config</td>
<td>S</td>
<td>UART configuration structure used to setup the UART</td>
<td>19-5</td>
</tr>
<tr>
<td>UART_config</td>
<td>F</td>
<td>Sets up the UART using the configuration structure</td>
<td>19-8</td>
</tr>
<tr>
<td>UART_eventDisable</td>
<td>F</td>
<td>Disable UART interrupts</td>
<td>19-8</td>
</tr>
<tr>
<td>UART_eventEnable</td>
<td>F</td>
<td>Enable UART interrupts</td>
<td>19-9</td>
</tr>
<tr>
<td>UART_fgetc</td>
<td>F</td>
<td>Read a character from UART by polling</td>
<td>19-10</td>
</tr>
<tr>
<td>UART_fgets</td>
<td>F</td>
<td>This routine reads a string from the uart</td>
<td>19-11</td>
</tr>
<tr>
<td>UART_fputc</td>
<td>F</td>
<td>Write a character from UART by polling</td>
<td>19-11</td>
</tr>
<tr>
<td>UART_fputs</td>
<td>F</td>
<td>This routine writes a string from the uart</td>
<td>19-11</td>
</tr>
<tr>
<td>UART_getConfig</td>
<td>F</td>
<td>Reads the UART configuration</td>
<td>19-11</td>
</tr>
<tr>
<td>UART_read</td>
<td>F</td>
<td>Read a buffer of data from UART by polling</td>
<td>19-12</td>
</tr>
<tr>
<td>UART_setCallback</td>
<td>F</td>
<td>Plugs UART interrupt routines into UART dispatcher table</td>
<td>19-12</td>
</tr>
</tbody>
</table>

Note: F = Function; S = Structure
### Table 19–1. UART APIs (Continued)

<table>
<thead>
<tr>
<th>Structure</th>
<th>Type</th>
<th>Purpose</th>
<th>See page</th>
</tr>
</thead>
<tbody>
<tr>
<td>UART_Setup</td>
<td>S</td>
<td>UART configuration structure used to setup the UART</td>
<td>19-5</td>
</tr>
<tr>
<td>UART_setup</td>
<td>F</td>
<td>Sets up the UART using the register values passed into the code</td>
<td>19-13</td>
</tr>
<tr>
<td>UART_write</td>
<td>F</td>
<td>Write a buffer of data to UART by polling</td>
<td>19-13</td>
</tr>
</tbody>
</table>

**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’)

UART_Setup

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.
UART_config

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
**UART_eventEnable**

**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

**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.
  
**Description**

This routine writes a character out through UART.

**Example**

```c
const int putchar = 'A';
CSLBool returnFlag;

returnFlag = UART_fputc(putchar, 0);
```

### UART_fputs

**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.
  
**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**

```c
UART_fputs("\n\rthis is a test!\n\r");
```

### UART_getConfig

**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**

```c
UART_Config Config;
UART_getConfig(&Config);
```
UART_read

**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**

```c
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**

```c
UART_IsrAddr MyIsrAddr = {
    NULL,  // Receiver line status
    UartRxIsr, // received data available
    UartTxIsr, // transmitter 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**

```c
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**

```c
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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) Macros to read/write UART register values</td>
<td></td>
</tr>
<tr>
<td>UART_RGET()</td>
<td>Uint16 UART_RGET(REG)</td>
</tr>
<tr>
<td>UART_RSET()</td>
<td>void UART_RSET(REG, Uint16 regval)</td>
</tr>
<tr>
<td>(b) Macros to read/write UART register field values</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(Applicable only to registers with more than one field)</td>
</tr>
<tr>
<td>UART_FGET()</td>
<td>Uint16 UART_FGET(REG, FIELD)</td>
</tr>
<tr>
<td>UART_FSET()</td>
<td>void UART_FSET(REG, FIELD, Uint16 fieldval)</td>
</tr>
<tr>
<td>(c) Macros to create value to write to UART registers and fields (Applicable only to registers with more than one field)</td>
<td></td>
</tr>
<tr>
<td>UART_REG_RMK()</td>
<td>Uint16 UART_REG_RMK(fieldval_n,...,fieldval_0)</td>
</tr>
<tr>
<td></td>
<td>Note: *Start with field values with most significant field positions: field_n: MSB field</td>
</tr>
<tr>
<td></td>
<td>field_0: LSB field</td>
</tr>
<tr>
<td></td>
<td>* only writable fields allowed</td>
</tr>
<tr>
<td>UART_FMK()</td>
<td>Uint16 UART_FMK(REG, FIELD, fieldval)</td>
</tr>
</tbody>
</table>

Notes: 1) REG indicates the registers: URIER, URIIR, URBRR, URTHR, URFCR, URLCR, URMCR, URLSR, URMSR, URDLL or URDLR.
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)

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>(d) Macros to read a register address</td>
<td></td>
</tr>
<tr>
<td>UART_ADDR()</td>
<td>Uint16 UART_ADDR(REG)</td>
</tr>
</tbody>
</table>

Notes:  
1) REG indicates the registers: URIER, URIIR, URBRB, URTHR, URFCR, URCR, URCMR, URLSR, URSR, URMSR, URDLL or URDLM.  
2) FIELD indicates the register field name.  
3) – or REG_FSET and REG_FM, 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_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_isDtr

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 None
Description Check if DTR is on. Return DTR value.
Example CSLBool dtrSignal;
dtrSignal = UART_isDtr;
This chapter describes the WDTIM module, lists the API structure, functions, and macros within the module, and provides a WDTIM API reference section.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>20.1 Overview</td>
<td>20-2</td>
</tr>
<tr>
<td>20.2 Configuration Structures</td>
<td>20-3</td>
</tr>
<tr>
<td>20.3 Functions</td>
<td>20-4</td>
</tr>
<tr>
<td>20.4 Macros</td>
<td>20-14</td>
</tr>
</tbody>
</table>
20.1 Overview

Table 20–1 lists the configuration structures and functions used with the WDTIM module.

Table 20–1. WDTIM Structure and APIs

<table>
<thead>
<tr>
<th>Structure</th>
<th>Description</th>
<th>See page...</th>
</tr>
</thead>
<tbody>
<tr>
<td>WDTIM_Config</td>
<td>Structure used to configure a WDTIM Device</td>
<td>20-3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
<th>See page...</th>
</tr>
</thead>
<tbody>
<tr>
<td>WDTIM_config</td>
<td>Configures WDTIM using configuration structure</td>
<td>20-4</td>
</tr>
<tr>
<td>WDTIM_service</td>
<td>Executes the watchdog service sequence</td>
<td>20-9</td>
</tr>
</tbody>
</table>

The following functions are supported by C5509/C5509A only

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
<th>See page...</th>
</tr>
</thead>
<tbody>
<tr>
<td>WDTIM_getConfig</td>
<td>Reads the WDTIM configuration structure</td>
<td>20-5</td>
</tr>
<tr>
<td>WDTIM_start</td>
<td>Starts the WDTIM device running</td>
<td>20-10</td>
</tr>
</tbody>
</table>

The following functions are supported by C5502 and C5501

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
<th>See page...</th>
</tr>
</thead>
<tbody>
<tr>
<td>WDTIM_close</td>
<td>Closes previously opened WDTIMER device</td>
<td>20-4</td>
</tr>
<tr>
<td>WDTIM_getCnt</td>
<td>Gives the timer count values</td>
<td>20-5</td>
</tr>
<tr>
<td>WDTIM_getPID</td>
<td>Gets peripheral ID details</td>
<td>20-6</td>
</tr>
<tr>
<td>WDTIM_init64</td>
<td>Initializes the timer in 64 bit mode</td>
<td>20-6</td>
</tr>
<tr>
<td>WDTIM_open</td>
<td>Opens the WDTIM device for use</td>
<td>20-9</td>
</tr>
<tr>
<td>WDTIM_start</td>
<td>Pulls both timers out of reset before activating the watchdog timer</td>
<td>20-10</td>
</tr>
<tr>
<td>WDTIM_stop</td>
<td>Stops all the timers if running</td>
<td>20-12</td>
</tr>
<tr>
<td>WDTIM_wdStart</td>
<td>Activates the watchdog timer</td>
<td>20-13</td>
</tr>
</tbody>
</table>
20.2 Configuration Structures

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

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 and C5501
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 wdtctrl1 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.

```c
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*

<table>
<thead>
<tr>
<th>Function</th>
<th>void WDTIM_close(WDTIM_Handle hWdtim)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td>hWdtim      Device handle; see WDTIM_open</td>
</tr>
<tr>
<td>Return Value</td>
<td>None</td>
</tr>
<tr>
<td>Description</td>
<td>WDTIM_close closes a previously opened WDTIMER device</td>
</tr>
</tbody>
</table>
| Example | WDTIM_Handle hWdtim;  
...  
WDTIM_close(hWdtim); |

**WDTIM_config**  
*Configures WDTIM using configuration structure*

<table>
<thead>
<tr>
<th>Function</th>
<th>For 5509/5509A only</th>
</tr>
</thead>
</table>
| void WDTIM_config(  
  WDTIM_Config *myConfig  
) |

<table>
<thead>
<tr>
<th>Function</th>
<th>For 5502 and 5501 only</th>
</tr>
</thead>
</table>
| void WDTIM_config(  
  WDTIM_Handle hWdtim,  
  WDTIM_Config *myConfig  
) |

<table>
<thead>
<tr>
<th>Arguments</th>
<th>For 5509/5509A only</th>
</tr>
</thead>
<tbody>
<tr>
<td>myConfig</td>
<td>Pointer to the initialized configuration structure</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arguments</th>
<th>For 5502 and 5501 only</th>
</tr>
</thead>
<tbody>
<tr>
<td>hWdtim</td>
<td>Device handle; see WDTIM_open</td>
</tr>
<tr>
<td>myConfig</td>
<td>Pointer to the initialized configuration structure</td>
</tr>
</tbody>
</table>

| 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. |
WDTIM_getConfig

Example
This is the example skeleton code for 5502 and 5501 only
WDTIM_Handle hWdtim;
WDTIM_Config MyConfig;
...
WDTIM_config(hWdtim, &MyConfig);

WDTIM_getCnt

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

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

**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

**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 initialize the timer in 64-bit mode. It allows to initialize the period and also provide arguments to setup the timer control registers.
**WDTIM_initChained32**  
*Initializes the timer in dual 34-bit chained mode*

**Function**  
```c
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 initialize 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**  
```c
WDTIM_Handle hWdtim;
......
WDTIM_initChained32(
    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_initDual32

Initializes the timer in dual 32-bit unchained mode

Function

```c
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 initialize 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 provides arguments to set up the timer control registers.

Example
```c
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**

```c
WDTIM_Handle hWdtim;
...
    hWdtim = WDTIM_open();
```

---

**WDTIM_service**

*Executes the watchdog service sequence*

**Function**

For 5509/5509A

```c
void WDTIM_service(
    void
);
```

**Arguments**

void

**Return Value**

None

**Description**

Executes the watchdog timer service sequence

**Example**

```c
WDTIM_service();
```

**Function**

For C5502 and 5501

```c
void WDTIM_service(
    WDTIM_Handle hWdt
);
```

**Arguments**

hWdt Device Handle; see WDTIM_open

**Return Value**

None
## WDTIM_start

**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 (5502/5501)

**Function**
For 5509/5509A only

```c
void WDTIM_start();
```

**Arguments**
void

**Return Value**
None

**Description**
Starts the watchdog timer device running.

**Example**
WDTIM_start();

**Function**
For 5502 and 5501 only

```c
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**

**Function**

```c
void WDTIM_stop(
   WDTIM_Handle hWdtim
)
```

**Arguments**

- `hWdtim`: Device Handle; see `WDTIM_open`.

**Return Value**

None

**Example**

Stops the timer if running.

```c
WDTIM_Handle hWdtim;
    ....
    WDTIM_stop(hWdtim);
```

**WDTIM_stop12**

**Function**

```c
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**

```c
WDTIM_Handle hWdtim;
    ....
    WDTIM_stop12(hWdtim);
```

**WDTIM_stop34**

**Function**

```c
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**

```c
WDTIM_Handle hWdtim;
    ....
    WDTIM_stop34(hWdtim);
```
## WDTIM_wdStart

**Activates the watchdog timer**

### Function

```c
void WDTIM_wdStart(
    WDTIM_Handle hWdt
)
```

### Arguments

- **Arguments hWdt**
  - Device Handle; see WDTIM_open

### Return Value

None

### Description

Activates the watchdog timer.

### Example

```c
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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>WDTIM_RGET()</td>
<td>Uint16 WDTIM_RGET(REG)</td>
</tr>
<tr>
<td>WDTIM_RSET()</td>
<td>void WDTIM_RSET(REG, Uint16 regval)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>WDTIM_FGET()</td>
<td>Uint16 WDTIM_FGET(REG, FIELD)</td>
</tr>
<tr>
<td>WDTIM_FSET()</td>
<td>void WDTIM_FSET(REG, FIELD, Uint16 fieldval)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>WDTIM_REG_RMK()</td>
<td>Uint16 WDTIM_REG_RMK(fieldval_n,...,fieldval_0)</td>
</tr>
<tr>
<td></td>
<td>Note: * Start with field values with most significant field positions:</td>
</tr>
<tr>
<td></td>
<td>field_n: MSB field</td>
</tr>
<tr>
<td></td>
<td>field_0: LSB field</td>
</tr>
<tr>
<td></td>
<td>* only writable fields allowed</td>
</tr>
<tr>
<td>WDTIM_FMK()</td>
<td>Uint16 WDTIM_FMK(REG, FIELD, fieldval)</td>
</tr>
</tbody>
</table>

(d) Macros to read a register address

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>WDTIM_ADDR()</td>
<td>Uint16 WDTIM_ADDR(REG)</td>
</tr>
</tbody>
</table>

Notes:

1) REG indicates the registers: WDTCR, WDPDRD, 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)
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.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>21.1 Overview</td>
<td>21-2</td>
</tr>
<tr>
<td>21.2 Configuration Structures</td>
<td>21-3</td>
</tr>
<tr>
<td>21.3 Functions</td>
<td>21-4</td>
</tr>
</tbody>
</table>
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**

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
<th>See page</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPT_Config</td>
<td>Structure used to configure a GPT device</td>
<td>21-3</td>
</tr>
<tr>
<td>GPT_OPEN_RESET</td>
<td>GPT reset flag, used while opening the GPT device</td>
<td>21-3</td>
</tr>
</tbody>
</table>

**Table 21–2. GPT Functions**

<table>
<thead>
<tr>
<th>Structure</th>
<th>Purpose</th>
<th>See page</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPT_close</td>
<td>Closes previously opened GPT device</td>
<td>21-4</td>
</tr>
<tr>
<td>GPT_config</td>
<td>Configure GPT using configuration structure</td>
<td>21-4</td>
</tr>
<tr>
<td>GPT_getCnt</td>
<td>Gives the timer count values</td>
<td>21-5</td>
</tr>
<tr>
<td>GPT_getConfig</td>
<td>Reads the current GPT configuration values</td>
<td>21-5</td>
</tr>
<tr>
<td>GPT_getEventId</td>
<td>Returns event ID of the opened GPT device</td>
<td>21-6</td>
</tr>
<tr>
<td>GPT_getPID</td>
<td>Gets peripheral ID details</td>
<td>21-6</td>
</tr>
<tr>
<td>GPT_init64</td>
<td>Initialize the timer in 64 bit mode</td>
<td>21-7</td>
</tr>
<tr>
<td>GPT_initChained32</td>
<td>Initialize the timer in dual 32 bit chained mode</td>
<td>21-8</td>
</tr>
<tr>
<td>GPT_initDual32</td>
<td>Initialize the timer in dual 32 bit unchained mode</td>
<td>21-9</td>
</tr>
<tr>
<td>GPT_open</td>
<td>Opens a GPT device for use</td>
<td>21-10</td>
</tr>
<tr>
<td>GPT_reset</td>
<td>Resets a GPT</td>
<td>21-10</td>
</tr>
<tr>
<td>GPT_start</td>
<td>Starts all the timers</td>
<td>21-11</td>
</tr>
<tr>
<td>GPT_start12</td>
<td>Starts the 32 bit timer1 device</td>
<td>21-11</td>
</tr>
<tr>
<td>GPT_start34</td>
<td>Starts the 32 bit timer2 device</td>
<td>21-11</td>
</tr>
<tr>
<td>GPT_stop</td>
<td>Stops the timer if running</td>
<td>21-12</td>
</tr>
<tr>
<td>GPT_stop12</td>
<td>Stops the 32 bit timer1 device if running</td>
<td>21-12</td>
</tr>
<tr>
<td>GPT_stop34</td>
<td>Stops the 32 bit timer2 device if running</td>
<td>21-13</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Structure</th>
<th>GPT_Config</th>
</tr>
</thead>
</table>
| 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 initialize this structure before passing its address to the GPT_config function.

**GPT_OPEN_RESET GPT**

Reset flag, used while opening the GPT device

<table>
<thead>
<tr>
<th>Constant</th>
<th>GPT_OPEN_RESET</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>This flag is used while opening a GPT device.</td>
</tr>
<tr>
<td>Example</td>
<td>See GPT_open</td>
</tr>
</tbody>
</table>
21.3 Functions

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

**GPT_close**

*Closes previously opened GPT device*

Function

```c
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

```c
GPT_Handle hGpt;
....
    GPT_close(hGpt);
```

**GPT_config**

*Configure GPT using configuration structure*

Function

```c
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.
**GPT_getConfig**

Example

```c
GPT_Handle hGpt;
GPT_Config MyConfig
...
GPT_config(hGpt, &MyConfig);
```

**GPT_getCnt** *Gives the timer count values*

Function

```c
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

```c
GPT_Handle hGpt;
Uint32 *tim12,*tim34;
...

GPT_getCnt(hGpt,tim34,tim12);
```

**GPT_getConfig** *Reads the current GPT configuration values*

Function

```c
void GPT_getConfig(
    GPT_Handle hGpt,
    GPT_Config *myConfig
)
```

Arguments

- `hGpt` Device Handle; see GPT_open
- `myConfig` Pointer to the configuration structure
GPT_getEventId

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**

*Initialize the timer in 64-bit mode*

**Function**

```c
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 initialize the timer in 64-bit mode. It allows to initialize the period and also provide arguments to setup the timer control registers.
GPT_initChained32

Example

```c
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**  
**Initialize the timer in dual 32-bit chained mode**

**Function**

```c
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 initialize two 32-bit timers in chained mode. It allows to initialize the period and also provide arguments to setup the timer control registers.

**Example**

```c
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**

*Initialize the timer in dual 32-bit unchained mode*

**Function**

```c
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 initialize 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 provides arguments to setup the timer control registers.

**Example**

```c
GPT_Handle hGpt;

......

GPT_initDual32(
    hGpt,
    0x3FE,   // ct11
    0x3FE,   // ct12
    0x005,   // prd1
    0x008,   // prd2
    0x0FF    // psc34
);
```
GPT_open

Opens a GPT device for use

Function

```c
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_Handle**: Device 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

```c
Handle hGpt;
...

hGpt = GPT_open(GPT_DEV0, GPT_OPEN_RESET);
```

GPT_reset

Resets a GPT

Function

```c
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**

```c
void GPT_start(
    GPT_Handle hGpt
);
```

**Arguments**

- `hGpt` Device Handle; see GPT_open

**Return Value**

- none

**Description**

Starts all the timers.

**Example**

```c
GPT_Handle hGpt;
......

GPT_start(hGpt);
```

---

**GPT_start12**

*Starts the 32-bit timer1 device*

**Function**

```c
void GPT_start12(
    GPT_Handle hGpt
);
```

**Arguments**

- `hGpt` Device Handle; see GPT_open

**Return Value**

- none

**Description**

Starts the 32-bit timer1 device.

**Example**

```c
GPT_Handle hGpt;
......

GPT_start12(hGpt);
```

---

**GPT_start34**

*Starts the 32-bit timer2 device*

**Function**

```c
void GPT_start34(
    GPT_Handle hGpt
);
```

**Arguments**

- `hGpt` Device Handle; see GPT_open
GPT_stop

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

A
ADC, registers 3-3
ADC functions
  ADC_config 3-5
  ADC_getConfig 3-5
  ADC_read 3-6
  ADC_setFreq 3-6
parameter-based functions 3-2
register-based functions 3-2
ADC module
  configuration structure 3-4
  examples 3-9
  functions 3-5
  include file 1-4
  macros 3-8
  module support symbol 1-4
ADC_Config 3-4
API modules, illustration of 1-2
architecture, of the CSL 1-2

B
build options
  defining a target device 2-8
  defining large memory model 2-10
  defining library paths 2-11

C
CHIP functions
  CHIP_getDieId_High32 4-3
  CHIP_getDieId_Low32 4-3
  CHIP_getRevid 4-3
CHIP module
  functions 4-2
  overview 4-2
CHIP module
  functions 4-3
  macros 4-4
chip module
  include file 1-4
  module support symbol 1-4
chip support library 1-2
constant values for fields 1-13
constant values for registers 1-13
CSL
  architecture 1-2
  benefits of 1-2
  data types 1-7
  functions 1-8
  generic macros, handle-based 1-12
  generic symbolic constants 1-13
  introduction to 1-2
  macros 1-11
  modules and include files 1-4
  naming conventions 1-6
CSL, generic functions
  CSL
  compiling and linking 2-7
  destination address 2-2
  directory structure 2-7
  See also compiling and linking with CSL
  how to use, overview 2-2
  source address 2-2
  transfer size 2-2
CSL bool. See data types
CSL device support 1-5
CSL_init 2-12
.csldata, allocation of 2-12
Index

D

DAT 5-2
  module support symbol 1-4

DAT functions
  DAT_close 5-3
  DAT_copy 5-3
  DAT_copy2D 5-4
  DAT_fill 5-5
  DAT_open 5-6
  DAT_wait 5-7

DAT module
  functions 5-2, 5-3
  include file 1-4
  overview 5-2

data types 1-7

device support 1-5
  device support symbols 1-5
  devices. See CSL device support

direct register initialization 1-8

directory structure 2-7
  documentation 2-7
  examples 2-7
  include files 2-7
  libraries 2-7
  source library 2-7

DMA configuration structures, DMA_Config 6-5

DMA functions
  DMA_close 6-6
  DMA_config 6-6
  DMA_getConfig 6-7
  DMA_getEventId 6-7
  DMA_open 6-8
  DMA_pause 6-9
  DMA_reset 6-9
  DMA_start 6-9
  DMA_stop 6-10

DMA macros
  DMA_ADDR 6-11
  DMA_ADDRH 6-11
  DMA_FGET 6-12
  DMA_FGETH 6-13
  DMA_FMK 6-14
  DMA_FSET 6-15
  DMA_FSETH 6-16
  DMA_REG_RMK 6-17
  DMA_RGET 6-18
  DMA_RGETH 6-19
  DMA_RST 6-19
  DMA_RSTETH 6-20

DMA module
  configuration structure 6-5
  functions 6-6
  include file 1-4
  macros 6-11
  using channel number 6-11
  module support symbol 1-4
  overview 6-2

DMA_AdrPtr. See data types

DMA_close 6-2
DMA_config 6-2
DMA_config(), using 2-2
DMA_open 6-2
DMA_reset 6-2
documentation. See directory structure

E

EMIF configuration structure, EMIF_Config 7-6

EMIF functions
  EMIF_config 7-8
  EMIF_enterselfRefresh 7-9
  EMIF_exitselfRefresh 7-10
  EMIF_getConfig 7-9
  EMIF_reset 7-10

EMIF macros, using port number 7-11

EMIF module
  configuration structures 7-6
  functions 7-8
  include file 1-4
  macros 7-11
  module support symbol 1-4
  overview 7-2

EMIF_config 7-2
  event ID 12-3
  See also IRQ module
  examples
    See also directory structure
    McBSP 13-26

F

FIELD 1-13
  explanation of 1-11
  fieldval, explanation of 1-11
  funcArg. See naming conventions
Index

function, naming conventions 1-6
function argument, naming conventions 1-6
function inlining, using 2-12
functional parameters, for use with peripheral initialization 1-10
functions 1-8
generic 1-9

generic CSL functions 1-9
GPIO configuration structure
  GPIO_Config 8-4
  GPIO_ConfigAll 8-4
GPIO functions
  GPIO_close 8-5
  GPIO_config 8-7
  GPIO_configAll 8-7
  GPIO_open 8-5
  GPIO_pinDirection 8-8
  GPIO_pinDisable 8-13
  GPIO_pinEnable 8-13
  GPIO_pinRead 8-14
  GPIO_pinReadAll 8-14
  GPIO_pinReset 8-16
  GPIO_pinWrite 8-15
  GPIO_pinWriteAll 8-15
GPIO module, configuration structures 8-4
GPIO module
  functions 8-5
  include file 1-4
  macros 8-17
  module support symbol 1-4
  Overview 8-2
GPT configuration structure
  GPT_Config 21-3
  GPT_OPEN_RESET_GPT 21-3
GPT module
  API reference
    configuration structure 21-4
    functions 21-2
    module support symbol 1-4
    overview 21-2
GPT_functions
  GPT_close 21-4
  GPT_config 21-4
  GPT_getCnt 21-5
  GPT_getConfig 21-5
  GPT_getPID 21-6
  GPT_init64 21-7
  GPT_initChained32 21-8
  GPT_initDual32 21-9
  GPT_open 21-10
  GPT_reset 21-10
  GPT_start 21-11
  GPT_start12 21-11
  GPT_start34 21-11
  GPT_stop 21-12
  GPT_stop12 21-12
  GPT_stop34 21-13

handles
  resource management 1-14
  use of 1-14
HPI module, functions 9-5
HPI Configuration Structures, HPI_Config 9-4
HPI functions
  HPI_config 9-5
  HPI_getConfig 9-5
HPI macros
  HPI_ADDR 9-6
  HPI_FGET 9-6
  HPI_FMKG 9-7
  HPI_FSET 9-7
  HPI_REG_RMK 9-8
  HPI_RGET 9-9
  HPI_RSET 9-9
HPI module
  HPI configuration structures 9-4
  include file 1-4
  macros 9-6
  module support symbol 1-4
  Overview 9-2
I
I2C Configuration Structures
  I2C_Config 10-5
  I2C_Setup 10-6
I2C Functions, I2C_sendStop 10-13
I2C functions
  I2C_config 10-7
Index

I2C_eventEnable 10-8
I2C_eventDisable 10-8
I2C_getConfig 10-8
I2C_getEventId 10-9
I2C_isrAddr 10-10
I2C_read 10-10
I2C_readByte 10-11
I2C_reset 10-12
I2C_rfull 10-12
I2C_rrdy 10-12
I2C_setCallback 10-13
I2C_setup 10-9
I2C_start 10-14
I2C_write 10-14
I2C_writeByte 10-15
I2C_xempty 10-16
I2C_xrdy 10-16

I2C module
Configuration Structures 10-5
examples 10-18
Functions 10-7
include file 1-4
macros 10-17
module support symbol 1-4
overview 10-2

ICACHE configuration structures
ICACHE_Config 11-3
ICACHE_Setup 11-4
ICACHE_Tagset 11-4

ICACHE functions
ICACHE_config 11-5
ICACHE_disable 11-5
ICACHE_enable 11-6
ICACHE_flush 11-6
ICACHE_freeze 11-6
ICACHE_setup 11-7
ICACHE_tagset 11-7
ICACHE_unfreeze 11-7

ICACHE macros
ICACHE_ADDR 11-8
ICACHE_FGET 11-8
ICACHE_FMK 11-8
ICACHE_FSET 11-8
ICACHE_REG_RMK 11-8
ICACHE_RGET 11-8
ICACHE_RST 11-8

ICACHE Module
include file 1-4
module support Symbol 1-4

ICACHE module
Configuration Structures 11-3
functions 11-5
macros 11-6
overview 11-2

include Files. See directory structure
include files, for CSL modules 1-4
Int16. See data types
Int32. See data types
IRQ configuration structure, IRQ_Config 12-2

IRQ functions
IRQ_clear 12-9
IRQ_config 12-9
IRQ_disable 12-10
IRQ_enable 12-10
IRQ_getArg 12-10
IRQ_getConfig 12-11
IRQ_globalDisable 12-11
IRQ_globalEnable 12-12
IRQ_globalRestore 12-12
IRQ_map 12-13
IRQ_plug 12-13
IRQ_restore 12-14
IRQ_setArg 12-14
IRQ_setVecs 12-15
IRQ_test 12-15

IRQ module
Configuration Structures 12-8
functions 12-9
include file 1-4
module support symbol 1-4
overview 12-2
using interrupts 12-7

IRQ_EVT_NNNN 12-4
events list 12-4

IRQ_EVT_WDTINT 12-6

L

large-model library. See CSL device support
large/small memory model selection, instructions 2-8
libraries
See also directory structure
linking to a project 2-10
linker command file
creating. See compiling and linking with CSL using 2-12
macro, naming conventions
  1-6
macros
  generic
    1-11
  handle-based
    1-12
generic description of
  FIELD
    1-11
  fieldval
    1-11
  PER
    1-11
  REG
    1-11
  REG#
    1-11
  regval
    1-11
McBSP
  13-23
McBSP example
  13-26
McBSP registers
  13-3
McBSP, configuration structure
  13-6
McBSP configuration structure, MCBSP_Config
  13-6
MCBSP Functions, MCBSP_channelStatus
  13-11
MCBSP functions
  MCBSP_channelDisable
    13-8
  MCBSP_channelEnable
    13-9
  MCBSP_close
    13-12
  MCBSP_config
    13-12
  MCBSP_getConfig
    13-14
  MCBSP_getPort
    13-14
  MCBSP_getRcvEventID
    13-15
  MCBSP_getXmtEventID
    13-15
  MCBSP_open
    13-16
  MCBSP_read16
    13-17
  MCBSP_read32
    13-17
  MCBSP_reset
    13-18
  MCBSP_rfull
    13-18
  MCBSP_rxdy
    13-19
  MCBSP_start
    13-19
  MCBSP_write16
    13-21
  MCBSP_write32
    13-21
  MCBSP_xempty
    13-22
  MCBSP_xrdy
    13-22
MCBSP Macros, MCBSP_FSET
  13-23
MCBSP macros
  MCBSP_ADDR
    13-24
  MCBSP_FGET
    13-23
  MCBSP_FMK
    13-23
  MCBSP_REG_RMK
    13-23
  MCBSP_REGGET
    13-23
  MCBSP_RSET
    13-23
  using handle
    13-24
  using port number
    13-23
McBSP module
  API reference
    13-8
  configuration structure
    13-2
  functions
    13-2
  include file
    1-4
  module support symbol
    1-4
  overview
    13-2
memberName. See naming conventions
MMC
  Configuration Structures, MMC_Config
    14-5
Data Structures
  MMC_CardIdobj
    14-9
  MMC_CardObj
    14-10
  MMC_CardXCsdoObj
    14-10
  MMC_CmdObj
    14-11
  MMC_MmcObj
    14-11
  MMC_NativeInitObj
    14-12
  MMC_RspObj
    14-12
Functions
  MMC_close
    14-13
  MMC_clrResponse
    14-13
  MMC_config
    14-14
  MMC_dispatch0
    14-14
  MMC_dispatch1
    14-14
  MMC_drdy
    14-15
  MMC_dxrdy
    14-15
  MMC_getCardCSD
    14-16
  MMC_getCardId
    14-16
  MMC_getConfig
    14-17
  MMC_getNumberOfCards
    14-17
  MMC_getStatus
    14-18
  MMC_open
    14-18
  MMC_readBlock
    14-19
  MMC_responseDone
    14-19
  MMC_saveStatus
    14-20
  MMC_selectCard
    14-20
  MMC_setAllCID
    14-21
  MMC_sendCmd
    14-22
  MMC_sendCSD
    14-22
  MMC_sendGoIdle
    14-23
  MMC_sendOpCond
    14-24
  MMC_setCallBack
    14-25
  MMC_setCardPtr
    14-23
  MMC_setRca
    14-25
  MMC_stop
    14-26
Index

MMC_setRca 14-25
MMC_writeBlock 14-27

MMC Data Structures
MMC_CallBackObj 14-6
MMC_CardCsdObj 14-7
SD_CardCsdObj 14-8

MMC Module
Configuration Structures 14-5
Data Structures 14-6
Functions 14-13
include file 1-4
module support Symbol 1-4
Overview 14-2

MMC_CardIdobj 14-9
MMC_CardObj 14-10
MMC_CardXcsdobj 14-10
MMC_close 14-13
MMC_clrResponse 14-13
MMC_CmdObj 14-11
MMC_Config 14-5
MMC_config, see also MMC_open 14-14
MMC_getCardId 14-16
MMC_getConfig 14-17
MMC_getNumberOfCards 14-17
MMC_MmcRegObj 14-11
MMC_NativeInitObj 14-12
MMC_open 14-18
MMC_readBlock 14-19
MMC_RspRegObj 14-12
MMC_selectCard 14-20
MMC_sendAllCID 14-21
MMC_sendCmd 14-22
MMC_setRca 14-25
MMC_writeBlock 14-27

module support symbols, for CSL modules 1-4

O
object types. See Naming Conventions

P
parameter-based configuration, ADC module 3-2
PER 1-13

explanation of 1-11
PER_ADDR 1-12
PER_close 1-9
PER_config 1-9

initialization of registers 1-9
PER_FGET 1-12
PER_FMK 1-12
PER_FSET 1-12
PER_funcName(). See naming conventions
PER_Handle. See data types
PER_MACRO_NAME. See naming conventions
PER_open 1-9
PER_REG_DEFAULT 1-13
PER_REG_FIELD_DEFAULT 1-13
PER_REG_FIELD_SYMVAL 1-13
PER_REG_RMK 1-11

for use with peripheral initialization 1-9
PER_reset 1-9
PER_RGET 1-11
PER_RSET 1-11
PER_setup 1-9
PER_setup(), example of use 1-10
PER_start 1-9
PER_Typename. See naming conventions
PER_varName(). See naming conventions

peripheral initialization via functional parameters 1-10

using PER_setup 1-10

peripheral initialization via registers 1-9
using PER_config 1-10

peripheral modules
descriptions of 1-4
include files 1-4
PLL configuration structure, PLL_Config 15-4
PLL functions
PLL_config 15-5
PLL_setFreq 15-6
PLL macros, using port number 15-7

N
naming conventions 1-6
PLL module
  API reference 15-5
  configuration structure 15-2
  functions 15-2
  include file 1-4
  macros 15-7
  module support symbol 1-4
  overview 15-2

PWR functions, PWR_powerDown 16-2
PWR macros 16-4
  PWR_ADDR 16-4
  PWR_FGET 16-4
  PWR_FMK 16-4
  PWR_FSET 16-4
  PWR_REG_RMK 16-4
  PWR_RGET 16-4
  PWR_RSET 16-4

PWR module
  API reference 16-3
  PWR_powerDown 16-3
  functions 16-2
  include file 1-4
  macros 16-4
  module support symbol 1-4
  overview 16-2

RTC_getConfig 17-10
RTC_getDate 17-11
RTC_getTime 17-11
RTC_isrDisable 17-10
RTC_isrDisphook 17-13
RTC_isrEnable 17-10
RTC_setAlarm 17-12
RTC_setDate 17-13
RTC_setPeriodicInterval 17-14
RTC_setTime 17-14

RTC ADDR 17-16
RTC_FGET 17-16
RTC_FMK 17-16
RTC_FSET 17-16
RTC_REG_RMK 17-16
RTC_RGET 17-17
RTC_RSET 17-17

RTC module
  API reference
  configuration structure
  functions
  include file
  macros
  module support symbol
  overview

RTC_bcdToDec, description of 17-3
RTC_decToBcd, description of 17-3

S
  scratch pad memory 2-12
  small-model library. See CSL device support
  source library. See directory structure
  static inline. See function inlining
  structure member, naming conventions 1-6
  symbolic constant values 1-13
  symbolic constants, generic 1-13
  SYMVAL 1-13

T
  target device, specifying. See compiling and linking
  with CSL
  TIMER configuration structure, TIMER_Config 18-3
  TIMER functions
    TIMER_close 18-4
## Index

<table>
<thead>
<tr>
<th>Function/Property</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMER_Config</td>
<td>18-4</td>
</tr>
<tr>
<td>TIMER_getConfig</td>
<td>18-5</td>
</tr>
<tr>
<td>TIMER_getEventID</td>
<td>18-5</td>
</tr>
<tr>
<td>TIMER_open</td>
<td>18-6</td>
</tr>
<tr>
<td>TIMER_reset</td>
<td>18-7</td>
</tr>
<tr>
<td>TIMER_start</td>
<td>18-7</td>
</tr>
<tr>
<td>TIMER_stop</td>
<td>18-7</td>
</tr>
<tr>
<td>TIMER_tintoutCfg</td>
<td>18-8</td>
</tr>
</tbody>
</table>

**TIMER macros**

<table>
<thead>
<tr>
<th>Macro</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMER_ADDR</td>
<td>18-9</td>
</tr>
<tr>
<td>TIMER_FGET</td>
<td>18-9</td>
</tr>
<tr>
<td>TIMER_FMK</td>
<td>18-9</td>
</tr>
<tr>
<td>TIMER_FSET</td>
<td>18-9</td>
</tr>
<tr>
<td>TIMER_REG_RMK</td>
<td>18-9</td>
</tr>
<tr>
<td>TIMER_RGET</td>
<td>18-9</td>
</tr>
<tr>
<td>TIMER_RSET</td>
<td>18-9</td>
</tr>
</tbody>
</table>

**TIMER module**

- API reference: 18-4
- configuration structure: 18-2
- configuration structures: 18-5
- configuration functions: 18-2
- include file: 1-4
- macros: 18-9
- module support symbol: 1-4
- overview: 18-2
- typedef, naming conventions: 1-6

**UART, Control Signal Macros**

- UART_Config: 19-5
- UART_Config: 19-5

**UART configuration structures**

<table>
<thead>
<tr>
<th>Function/Property</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>UART_Config</td>
<td>19-5</td>
</tr>
<tr>
<td>UART_Setup</td>
<td>19-5</td>
</tr>
</tbody>
</table>

**UART functions**

<table>
<thead>
<tr>
<th>Function/Property</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>UART_config</td>
<td>19-8</td>
</tr>
<tr>
<td>UART_eventDisable</td>
<td>19-8</td>
</tr>
<tr>
<td>UART_eventEnable</td>
<td>19-9</td>
</tr>
<tr>
<td>UART_fgetc</td>
<td>19-10</td>
</tr>
<tr>
<td>UART_fgets</td>
<td>19-10</td>
</tr>
<tr>
<td>UART_fputc</td>
<td>19-11</td>
</tr>
<tr>
<td>UART_fputs</td>
<td>19-11</td>
</tr>
<tr>
<td>UART_getConfig</td>
<td>19-11</td>
</tr>
<tr>
<td>UART_read</td>
<td>19-12</td>
</tr>
<tr>
<td>UART_setCallback</td>
<td>19-12</td>
</tr>
<tr>
<td>UART_setup</td>
<td>19-13</td>
</tr>
<tr>
<td>UART_write</td>
<td>19-13</td>
</tr>
</tbody>
</table>

**UART macros**

<table>
<thead>
<tr>
<th>Macro</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>UART_ctsOff</td>
<td>19-16</td>
</tr>
</tbody>
</table>

**UART module**

- configuration structure: 19-2
- configuration structures: 19-5
- configuration functions: 19-2, 19-8
- include file: 1-4
- macros: 19-14
- module support symbol: 1-4
- overview: 19-2

Uchar. See data types

Uint16. See data types

Uint32. See data types

**USB module**

- configuration information: 1-4
- include file: 1-4
- module support symbol: 1-4
- using functional parameters: 1-8

**V**

variable, naming conventions: 1-6

**W**

WDTIM configuration structures, WDTIM_Config: 20-3

WDTIM functions

<table>
<thead>
<tr>
<th>Function/Property</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>WDTIM_close</td>
<td>20-4</td>
</tr>
<tr>
<td>WDTIM_config</td>
<td>20-4</td>
</tr>
<tr>
<td>WDTIM_getCnt</td>
<td>20-5</td>
</tr>
<tr>
<td>WDTIM_getPID</td>
<td>20-6</td>
</tr>
<tr>
<td>WDTIM_init64</td>
<td>20-6</td>
</tr>
<tr>
<td>WDTIM_initChained32</td>
<td>20-7</td>
</tr>
<tr>
<td>WDTIM_initDual32</td>
<td>20-8</td>
</tr>
<tr>
<td>WDTIM_open</td>
<td>20-9</td>
</tr>
<tr>
<td>WDTIM_service</td>
<td>20-9</td>
</tr>
<tr>
<td>WDTIM_start</td>
<td>20-10</td>
</tr>
<tr>
<td>WDTIM_start12</td>
<td>20-11</td>
</tr>
<tr>
<td>WDTIM_start34</td>
<td>20-11</td>
</tr>
<tr>
<td>WDTIM_stop</td>
<td>20-12</td>
</tr>
<tr>
<td>WDTIM_stop12</td>
<td>20-12</td>
</tr>
<tr>
<td>WDTIM_stop34</td>
<td>20-12</td>
</tr>
<tr>
<td>WDTIM_stopStart</td>
<td>20-13</td>
</tr>
</tbody>
</table>

| WDTIM_FMK | 20-14 |
| WDTIM_FSET | 20-14 |
| WDTIM_REG_RMK | 20-14 |
| WDTIM_RGET | 20-14 |
| WDTIM_RSET | 20-14 |

**WDTIM module**
- API reference | 20-4 |
- APIs | 20-2 |
- include file | 1-4 |
- macros | 20-14 |
- module support symbol | 1-4 |
- overview | 20-2 |