IMPORTANT NOTICE

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

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

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

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

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

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

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Copyright © 2003, Texas Instruments Incorporated
About This Manual

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

How to Use This Manual

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

- Chapter 1, CSL Overview, provides an overview of the CSL, includes tables showing CSL module support for various C5000 devices, and lists the CSL modules.

- Chapter 2, How To Use CSL, provides basic examples of how to use CSL functions with or without using the DSP/BIOS™ Configuration Tool, and shows how to define Build options for the CSL in the Code Composer Studio™ environment.

- Chapters 3-16 provide basic examples, functions, macros, and CSL GUI configurations for the individual CSL modules.

- Appendix A provides examples of how to use CSL C5000 Registers.
Notational Conventions

This document uses the following conventions:

- Program listings, program examples, and interactive displays are shown in a special typeface.

- In syntax descriptions, the function or macro appears in a **bold typeface** and the parameters appear in plainface within parentheses. Portions of a syntax that are in **bold** should be entered as shown; portions of a syntax that are within parentheses describe the type of information that should be entered.

- Macro names are written in uppercase text; function names are written in lowercase.

- TMS320C54x™ DSP devices are referred to throughout this reference guide as C5401, C5402, etc.
The following books describe the TMS320C54x™ DSP and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477-8924. When ordering, please identify the book by its title and literature number. Many of these documents are located on the internet at http://www.ti.com.

**TMS320C54x Assembly Language Tools User's Guide** (literature number SPRU102) describes the assembly language tools (assembler, linker, and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the C54x generation of devices.

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

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

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

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

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

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

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

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

1 CSL Overview ................................................................. 1-1
   Overview of the features and architecture of the Chip Support Library.
   1.1 Introduction to the CSL .............................................. 1-2
      1.1.1 Benefits of the CSL .......................................... 1-2
      1.1.2 CSL Architecture ........................................... 1-2
   1.2 Naming Conventions ................................................ 1-6
   1.3 Data Types ........................................................... 1-7
   1.4 Functions ............................................................. 1-8
      1.4.1 Peripheral Initialization via Registers .................... 1-10
   1.5 Macros ............................................................... 1-11
   1.6 Symbolic Constant Values ....................................... 1-14
   1.7 Resource Management and the Use of CSL Handles .......... 1-15
      1.7.1 Using CSL Handles .......................................... 1-15
   1.8 Support for Device-Specific Features ......................... 1-17

2 How To Use CSL .............................................................. 2-1
   This chapter explains the features of the CSL GUI. Instructions and examples are included.
   2.1 Overview .............................................................. 2-2
   2.2 Using CSL with the CSL GUI ...................................... 2-3
      2.2.1 CSL GUI: The CSL Tree .................................... 2-3
      2.2.2 Steps to configuring a peripheral using CSL GUI ..... 2-7
      2.2.3 TIMER Configuration Example ............................. 2-9
      2.2.4 Files Generated by the CSL GUI ......................... 2-11
      2.2.5 Modifying Your Application Code (main.c) .......... 2-15
   2.3 Using CSL without the CSL GUI .................................. 2-17
      2.3.1 Using the DMA_config() function ....................... 2-18
      2.3.2 Compiling and Linking With CSL using Code Composer Studio (CCStudio) 2-19
   2.4 Rebuilding CSL ...................................................... 2-26

3 CHIP Module ................................................................. 3-1
   General description of the CSL CHIP module, its functions, and macros.
   3.1 Overview .............................................................. 3-2
   3.2 Functions ............................................................. 3-3
4 DAA Module ............................................. 4-1
This section include API functions for the DAA module.
4.1 Overview ............................................. 4-2
   4.1.1 Configuration Structures ......................... 4-2
   4.1.2 Functions ......................................... 4-3
   4.1.3 Macros ............................................. 4-3
4.2 Configuration Structures ................................ 4-4
4.3 Functions ............................................. 4-7
4.4 Macros ................................................. 4-11

5 DAT Module ............................................. 5-1
General description of the DAT module and its functions.
5.1 Overview ............................................. 5-2
5.2 Functions ............................................. 5-4

6 DMA Module ............................................. 6-1
General description of the DMA module, its configuration structures, functions, and macros.
6.1 Overview ............................................. 6-2
6.2 Configuration Structure ................................ 6-4
6.3 Functions ............................................. 6-7
   6.3.1 DMA Primary Functions ......................... 6-7
   6.3.2 DMA Global Register Function .................. 6-12
   6.3.3 DMA Auxiliary Functions ....................... 6-18
6.4 Macros ................................................. 6-20
6.5 Configuring the DMA Module Using CSL GUI ............ 6-32
   6.5.1 DMA Configuration Manager ..................... 6-32
   6.5.2 DMA Resource Manager ......................... 6-35
   6.5.3 C Code Generation for DMA Module .............. 6-37
6.6 Examples .............................................. 6-40

7 EBUS Module ............................................. 7-1
Features and description of the external bus interface (EBUS) module.
7.1 Overview ............................................. 7-2
7.2 Configuration Structure ................................ 7-3
7.3 Functions ............................................. 7-4
7.4 Macros ................................................. 7-6

8 GPIO Module ............................................. 8-1
General description of the GPIO module and its macros.
8.1 Overview ............................................. 8-2
8.2 Functions ............................................. 8-3
8.3 Macros ................................................. 8-5
8.4 Configuring the GPIO Module Using CSL GUI .......... 8-7
   8.4.1 Non-Multiplexed GPIO Configuration Manager .... 8-7
   8.4.2 C Code Generation for GPIO Module .............. 8-8
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>HPI Module</td>
<td>9-1</td>
</tr>
<tr>
<td></td>
<td><em>Descriptions for macros available in the HPI module.</em></td>
<td></td>
</tr>
<tr>
<td>9.1</td>
<td>Macros</td>
<td>9-2</td>
</tr>
<tr>
<td>10</td>
<td>IRQ Module</td>
<td>10-1</td>
</tr>
<tr>
<td></td>
<td>*Overview of the IRQ module, including the configuration structure,</td>
<td></td>
</tr>
<tr>
<td></td>
<td><em>functions, and examples of interrupts.</em></td>
<td></td>
</tr>
<tr>
<td>10.1</td>
<td>Overview</td>
<td>10-2</td>
</tr>
<tr>
<td>10.1.1</td>
<td>The Event ID Concept</td>
<td>10-4</td>
</tr>
<tr>
<td>10.2</td>
<td>Using Interrupts with CSL</td>
<td>10-7</td>
</tr>
<tr>
<td>10.3</td>
<td>Configuration Structure</td>
<td>10-8</td>
</tr>
<tr>
<td>10.4</td>
<td>Functions</td>
<td>10-9</td>
</tr>
<tr>
<td>11</td>
<td>McBSP Module</td>
<td>11-1</td>
</tr>
<tr>
<td></td>
<td>*General description of the McBSP Module, its configuration structure,</td>
<td></td>
</tr>
<tr>
<td></td>
<td><em>functions and macros.</em></td>
<td></td>
</tr>
<tr>
<td>11.1</td>
<td>Overview</td>
<td>11-2</td>
</tr>
<tr>
<td>11.2</td>
<td>Configuration Structure</td>
<td>11-4</td>
</tr>
<tr>
<td>11.3</td>
<td>Functions</td>
<td>11-6</td>
</tr>
<tr>
<td>11.4</td>
<td>Macros</td>
<td>11-24</td>
</tr>
<tr>
<td>11.5</td>
<td>Configuring the McBSP Module Using CSL GUI</td>
<td>11-42</td>
</tr>
<tr>
<td>11.5.1</td>
<td>Overview</td>
<td>11-42</td>
</tr>
<tr>
<td>11.5.2</td>
<td>McBSP Configuration Manager</td>
<td>11-42</td>
</tr>
<tr>
<td>11.5.3</td>
<td>McBSP Resource Manager</td>
<td>11-46</td>
</tr>
<tr>
<td>11.5.4</td>
<td>C Code Generation for McBSP Module</td>
<td>11-47</td>
</tr>
<tr>
<td>11.6</td>
<td>Examples</td>
<td>11-50</td>
</tr>
<tr>
<td>12</td>
<td>PLL Module</td>
<td>12-1</td>
</tr>
<tr>
<td></td>
<td><em>Describes the structure, functions, and macros of the PLL module.</em></td>
<td></td>
</tr>
<tr>
<td>12.1</td>
<td>Overview</td>
<td>12-2</td>
</tr>
<tr>
<td>12.2</td>
<td>Configuration Structure</td>
<td>12-3</td>
</tr>
<tr>
<td>12.3</td>
<td>Functions</td>
<td>12-4</td>
</tr>
<tr>
<td>12.4</td>
<td>Macros</td>
<td>12-6</td>
</tr>
<tr>
<td>12.5</td>
<td>Configuring the PLL Module Using CSL GUI</td>
<td>12-8</td>
</tr>
<tr>
<td>12.5.1</td>
<td>Overview</td>
<td>12-8</td>
</tr>
<tr>
<td>12.5.2</td>
<td>PLL Configuration Manager</td>
<td>12-8</td>
</tr>
<tr>
<td>12.5.3</td>
<td>PLL Resource Manager</td>
<td>12-10</td>
</tr>
<tr>
<td>12.5.4</td>
<td>C Code Generation for PLL Module</td>
<td>12-11</td>
</tr>
<tr>
<td>13</td>
<td>PWR Module</td>
<td>13-1</td>
</tr>
<tr>
<td></td>
<td><em>General description of the PWR module and its functions.</em></td>
<td></td>
</tr>
<tr>
<td>13.1</td>
<td>Overview</td>
<td>13-2</td>
</tr>
<tr>
<td>13.2</td>
<td>Functions</td>
<td>13-3</td>
</tr>
</tbody>
</table>
**14 TIMER Module**

*General description of the structure and functions for the TIMER module.*

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>14.1 Overview</td>
<td>14-2</td>
</tr>
<tr>
<td>14.2 Configuration Structure</td>
<td>14-3</td>
</tr>
<tr>
<td>14.3 Functions</td>
<td>14-4</td>
</tr>
<tr>
<td>14.4 Macros</td>
<td>14-9</td>
</tr>
<tr>
<td>14.5 Configuring the TIMER Module Using CSL GUI</td>
<td>14-12</td>
</tr>
<tr>
<td>14.5.1 Overview</td>
<td>14-12</td>
</tr>
<tr>
<td>14.5.2 TIMER Configuration Manager</td>
<td>14-12</td>
</tr>
<tr>
<td>14.5.3 TIMER Resource Manager</td>
<td>14-14</td>
</tr>
<tr>
<td>14.5.4 C Code Generation for TIMER</td>
<td>14-16</td>
</tr>
</tbody>
</table>

**15 UART Module**

*This chapter describes the UART module, lists the API structure, functions, macros, and instructions for configuring using CSL GUI.*

<table>
<thead>
<tr>
<th>Section</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>15-5</td>
</tr>
<tr>
<td>15.3 Functions</td>
<td>15-8</td>
</tr>
<tr>
<td>15.3.1 CSL Primary Functions</td>
<td>15-8</td>
</tr>
<tr>
<td>15.4 Macros</td>
<td>15-15</td>
</tr>
<tr>
<td>15.4.1 General Macros</td>
<td>15-15</td>
</tr>
<tr>
<td>15.4.2 UART Control Signal Macros</td>
<td>15-16</td>
</tr>
<tr>
<td>15.5 Configuring the UART Module Using CSL GUI</td>
<td>15-21</td>
</tr>
<tr>
<td>15.5.1 UART Configuration Manager</td>
<td>15-21</td>
</tr>
<tr>
<td>15.5.2 UART Resource Manager</td>
<td>15-23</td>
</tr>
<tr>
<td>15.5.3 C Code Generation for UART Module</td>
<td>15-24</td>
</tr>
</tbody>
</table>

**16 WDTIM Module**

*General description of the WDTIM module, its configuration structure, functions, and macros.*

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.1 Overview</td>
<td>16-2</td>
</tr>
<tr>
<td>16.2 Configuration Structure</td>
<td>16-3</td>
</tr>
<tr>
<td>16.3 Functions</td>
<td>16-4</td>
</tr>
<tr>
<td>16.4 Macros</td>
<td>16-6</td>
</tr>
<tr>
<td>16.5 Configuring the WATCHDOG TIMER Module Using CSL GUI</td>
<td>16-8</td>
</tr>
<tr>
<td>16.5.1 Overview</td>
<td>16-8</td>
</tr>
<tr>
<td>16.5.2 WATCHDOG TIMER Configuration Manager</td>
<td>16-8</td>
</tr>
<tr>
<td>16.5.3 WATCHDOG TIMER Resource Manager</td>
<td>16-10</td>
</tr>
<tr>
<td>16.5.4 C Code Generation for WATCHDOG TIMER</td>
<td>16-11</td>
</tr>
</tbody>
</table>
## Contents

**A Peripheral Registers**

*Provides symbolic constants for the peripheral registers.*

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1</td>
<td>DMA Registers</td>
<td>A-2</td>
</tr>
<tr>
<td>A.1.1</td>
<td>DMA Channel Priority and Enable Control Register (DMPREC)</td>
<td>A-2</td>
</tr>
<tr>
<td>A.1.2</td>
<td>DMA Channel n Sync Select and Frame Count Register (DMSFCn)</td>
<td>A-3</td>
</tr>
<tr>
<td>A.1.3</td>
<td>DMA Channel n Transfer Mode Control Register (DMMCRn)</td>
<td>A-6</td>
</tr>
<tr>
<td>A.1.4</td>
<td>DMA Channel n Source Address Register (DMSRCn)</td>
<td>A-9</td>
</tr>
<tr>
<td>A.1.5</td>
<td>DMA Global Source Address Reload Register (DMGSA)</td>
<td>A-9</td>
</tr>
<tr>
<td>A.1.6</td>
<td>DMA Source Program Page Address Register (DMSRPCP)</td>
<td>A-9</td>
</tr>
<tr>
<td>A.1.7</td>
<td>DMA Channel n Destination Address Register (DMDSN)</td>
<td>A-10</td>
</tr>
<tr>
<td>A.1.8</td>
<td>DMA Global Destination Address Reload Register (DMGDA)</td>
<td>A-10</td>
</tr>
<tr>
<td>A.1.9</td>
<td>DMA Destination Program Page Address Register (DMDSNTP)</td>
<td>A-11</td>
</tr>
<tr>
<td>A.1.10</td>
<td>DMA Channel n Element Count Register (DMCTRn)</td>
<td>A-11</td>
</tr>
<tr>
<td>A.1.11</td>
<td>DMA Global Element Count Reload Register (DMGCER)</td>
<td>A-12</td>
</tr>
<tr>
<td>A.1.12</td>
<td>DMA Global Frame Count Reload Register (DMGCR)</td>
<td>A-12</td>
</tr>
<tr>
<td>A.1.13</td>
<td>DMA Element Address Index Register 0 (DMIDX0)</td>
<td>A-13</td>
</tr>
<tr>
<td>A.1.14</td>
<td>DMA Element Address Index Register 1 (DMIDX1)</td>
<td>A-13</td>
</tr>
<tr>
<td>A.1.15</td>
<td>DMA Frame Address Index Register 0 (DMFRI0)</td>
<td>A-14</td>
</tr>
<tr>
<td>A.1.16</td>
<td>DMA Frame Address Index Register 1 (DMFRI1)</td>
<td>A-14</td>
</tr>
<tr>
<td>A.1.17</td>
<td>DMA Global Extended Source Data Page Register (DMSECDP)</td>
<td>A-15</td>
</tr>
<tr>
<td>A.1.18</td>
<td>DMA Global Extended Destination Data Page Register (DMSESTD)</td>
<td>A-15</td>
</tr>
<tr>
<td>A.2</td>
<td>EBUS Registers</td>
<td>A-16</td>
</tr>
<tr>
<td>A.2.1</td>
<td>Software Wait-State Register (SWWSR)</td>
<td>A-16</td>
</tr>
<tr>
<td>A.2.2</td>
<td>Software Wait-State Control Register (SWCR)</td>
<td>A-17</td>
</tr>
<tr>
<td>A.2.3</td>
<td>Bank-Switching Control Register (BSCR)</td>
<td>A-17</td>
</tr>
<tr>
<td>A.3</td>
<td>GPIO Registers (C5440 and C5441)</td>
<td>A-22</td>
</tr>
<tr>
<td>A.3.1</td>
<td>General Purpose I/O Register (GPIO)</td>
<td>A-22</td>
</tr>
<tr>
<td>A.4</td>
<td>HPI Registers</td>
<td>A-24</td>
</tr>
<tr>
<td>A.4.1</td>
<td>General Purpose I/O Control Register (GPIOCR)</td>
<td>A-24</td>
</tr>
<tr>
<td>A.4.2</td>
<td>General Purpose I/O Status Register (GPIOSR)</td>
<td>A-25</td>
</tr>
<tr>
<td>A.4.3</td>
<td>HPI Control Register (HPIC) (for 5401, 5402, 5409, and 5410 only)</td>
<td>A-25</td>
</tr>
<tr>
<td>A.5</td>
<td>Multichannel BSP (McBSP) Registers</td>
<td>A-26</td>
</tr>
<tr>
<td>A.5.1</td>
<td>McBSP Serial Port Control Register (SPCR1)</td>
<td>A-26</td>
</tr>
<tr>
<td>A.5.2</td>
<td>McBSP Serial Port Control Register 2 (SPCR2)</td>
<td>A-28</td>
</tr>
<tr>
<td>A.5.3</td>
<td>McBSP Pin Control Register (PCR)</td>
<td>A-30</td>
</tr>
<tr>
<td>A.5.4</td>
<td>Receive Control Register 1 (RCR1)</td>
<td>A-33</td>
</tr>
<tr>
<td>A.5.5</td>
<td>Receive Control Register 2 (RCR2)</td>
<td>A-34</td>
</tr>
<tr>
<td>A.5.6</td>
<td>Transmit Control Register 1 (XCR1)</td>
<td>A-35</td>
</tr>
<tr>
<td>A.5.7</td>
<td>Transmit Control Register 2 (XCR2)</td>
<td>A-36</td>
</tr>
<tr>
<td>A.5.8</td>
<td>Sample Rate Generator Register 1 (SRGR1)</td>
<td>A-38</td>
</tr>
<tr>
<td>A.5.9</td>
<td>Sample Rate Generator Register 2 (SRGR2)</td>
<td>A-38</td>
</tr>
<tr>
<td>A.5.10</td>
<td>Multichannel Control Register 1 (MCR1)</td>
<td>A-40</td>
</tr>
<tr>
<td>A.5.11</td>
<td>Multichannel Control Register 2 (MCR2)</td>
<td>A-41</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>A.5.12</td>
<td>Receive Channel Enable Register (RCERn)</td>
<td>A-43</td>
</tr>
<tr>
<td>A.5.13</td>
<td>Transmit Channel Enable Register (XCERn)</td>
<td>A-45</td>
</tr>
<tr>
<td>A.6</td>
<td>PLL Registers (CLKMD)</td>
<td>A-47</td>
</tr>
<tr>
<td>A.7</td>
<td>Timer Registers</td>
<td>A-49</td>
</tr>
<tr>
<td>A.7.1</td>
<td>Timer Control Register (TCR)</td>
<td>A-49</td>
</tr>
<tr>
<td>A.7.2</td>
<td>Timer Secondary Control Register (TSCR)</td>
<td>A-51</td>
</tr>
<tr>
<td>A.7.3</td>
<td>Timer Period Register (PRD)</td>
<td>A-51</td>
</tr>
<tr>
<td>A.8</td>
<td>Watchdog Timer Registers (C5441)</td>
<td>A-52</td>
</tr>
<tr>
<td>A.8.1</td>
<td>Watchdog Timer Control Register (WDTCR)</td>
<td>A-52</td>
</tr>
<tr>
<td>A.8.2</td>
<td>Watchdog Timer Secondary Control Register (WDTSCR)</td>
<td>A-54</td>
</tr>
<tr>
<td>A.8.3</td>
<td>Watchdog Timer Period Register (WDPRD)</td>
<td>A-55</td>
</tr>
</tbody>
</table>
# Figures

1–1 API Modules ................................................................. 1-3
2–1 CSL Tree (for 5402) ..................................................... 2-3
2–2 Expanded CSL Tree (for C5402) ................................. 2-4
2–3 Insert Configuration Object ................................. 2-5
2–4 Delete/Rename Options ................................................. 2-6
2–5 Show Dependency Option ............................................. 2-7
2–6 Code Composer Studio IDE Project Window ................. 2-8
2–7 Code Composer Studio IDE Project Window with .cdb Project File Added 2-8
2–8 CCStudio IDE Project View ........................................... 2-9
2–9 Configuring the TIMER1 Device .................................. 2-10
2–10 Header File mytimercfg.h ........................................... 2-12
2–11 Resource Manager Properties Page ......................... 2-14
2–12 Source File mytimercfg_c.c ......................................... 2-15
2–13 Example of main.c File Using Data Generated by the CSL GUI 2-16
2–14 Defining the Target Device in the Build Options Dialog 2-21
2–15 Defining Far Mode ....................................................... 2-22
2–16 Defining Large Memory Model .................................... 2-23
2–17 Defining Library Paths .................................................. 2-24
6–1 DMA Sections Menu ................................................... 6-32
6–2 DMA Properties Page ................................................ 6-34
6–3 DMA Resource Manager Menu .............................. 6-35
6–4 DMA Properties Page With Handle Object Accessible .... 6-36
6–5 DMA Channel Initialization Using DMA_config() .......... 6-41
8–1 GPIO Sections Menu ..................................................... 8-7
8–2 GPIO Properties Page .................................................. 8-7
11–1 McBSP Sections Menu ................................................. 11-42
11–2 McBSP Properties Page .............................................. 11-45
11–3 McBSP Resource Manager Menu .......................... 11-46
11–4 McBSP Properties Page With Handle Object Accessible 11-47
12–1 PLL Sections Menu ..................................................... 12-8
12–2 PLL Properties Page ................................................... 12-9
12–3 PLL Resource Manager Menu ....................................... 12-10
12–4 PLL Properties Page .................................................. 12-11
14–1 Timer Sections Menu .................................................. 14-12
14–2 TIMER Properties Page .............................................. 14-14
14–3 Timer Resource Manager Menu .................................. 14-14
14–4 Timer Properties Page With Handle Object Accessible .... 14-15
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>15–1</td>
<td>UART Sections Menu</td>
<td>15-21</td>
</tr>
<tr>
<td>15–2</td>
<td>UART Properties Page</td>
<td>15-23</td>
</tr>
<tr>
<td>15–3</td>
<td>UART Resource Manager Menu</td>
<td>15-23</td>
</tr>
<tr>
<td>15–4</td>
<td>UART Resource Manager Properties Page</td>
<td>15-24</td>
</tr>
<tr>
<td>16–1</td>
<td>WATCHDOG TIMER Sections Menu</td>
<td>16-8</td>
</tr>
<tr>
<td>16–2</td>
<td>WATCHDOG TIMER Properties Page</td>
<td>16-10</td>
</tr>
<tr>
<td>16–3</td>
<td>WATCHDOG TIMER Resource Manager Menu</td>
<td>16-10</td>
</tr>
<tr>
<td>16–4</td>
<td>WATCHDOG TIMER Properties Page</td>
<td>16-11</td>
</tr>
<tr>
<td>A–1</td>
<td>DMA Channel Priority and Enable Control Register (DMPREC)</td>
<td>A-2</td>
</tr>
<tr>
<td>A–2</td>
<td>DMA Channel n Sync Select and Frame Count Register (DMSFCn)</td>
<td>A-3</td>
</tr>
<tr>
<td>A–3</td>
<td>DMA Channel n Transfer Mode Control Register (DMMCRn)</td>
<td>A-6</td>
</tr>
<tr>
<td>A–4</td>
<td>DMA Channel n Source Address Register (DMSRCn)</td>
<td>A-9</td>
</tr>
<tr>
<td>A–5</td>
<td>DMA Global Source Address Reload Register (DMGSA)</td>
<td>A-9</td>
</tr>
<tr>
<td>A–6</td>
<td>DMA Source Program Page Address Register (DMSRCP)</td>
<td>A-9</td>
</tr>
<tr>
<td>A–7</td>
<td>DMA Channel n Destination Address Register (DMDSTn)</td>
<td>A-10</td>
</tr>
<tr>
<td>A–8</td>
<td>DMA Global Destination Address Reload Register (DMGDA)</td>
<td>A-10</td>
</tr>
<tr>
<td>A–9</td>
<td>DMA Destination Program Page Address Register (DMDSTP)</td>
<td>A-11</td>
</tr>
<tr>
<td>A–10</td>
<td>DMA Channel n Element Count Register (DMCTRn)</td>
<td>A-11</td>
</tr>
<tr>
<td>A–11</td>
<td>DMA Global Element Count Reload Register (DMGCR)</td>
<td>A-12</td>
</tr>
<tr>
<td>A–12</td>
<td>DMA Global Frame Count Reload Register (DMGFR)</td>
<td>A-12</td>
</tr>
<tr>
<td>A–13</td>
<td>DMA Element Address Index Register 0 (DMIDX0)</td>
<td>A-13</td>
</tr>
<tr>
<td>A–14</td>
<td>DMA Element Address Index Register 1 (DMIDX1)</td>
<td>A-13</td>
</tr>
<tr>
<td>A–15</td>
<td>DMA Frame Address Index Register 0 (DMFRI0)</td>
<td>A-14</td>
</tr>
<tr>
<td>A–16</td>
<td>DMA Frame Address Index Register 1 (DMFRI1)</td>
<td>A-14</td>
</tr>
<tr>
<td>A–17</td>
<td>DMA Global Extended Source Data Page Register (DMSRCDP)</td>
<td>A-15</td>
</tr>
<tr>
<td>A–18</td>
<td>DMA Global Extended Destination Data Page Register (DMDSTDP)</td>
<td>A-15</td>
</tr>
<tr>
<td>A–19</td>
<td>Software Wait-State Register(SWWSR)-All devices except C5440 and C5441</td>
<td>A-16</td>
</tr>
<tr>
<td>A–20</td>
<td>Software Wait-State Control Register (SWCR)-All devices except C5440 and C5441</td>
<td>A-16</td>
</tr>
<tr>
<td>A–21</td>
<td>Bank-Switching Control Register (BCR) — C5401, C5402, C5409, C5420, C5421, and 5471</td>
<td>A-17</td>
</tr>
<tr>
<td>A–22</td>
<td>Bank-Switching Control Register (BCR) — C5410, C5410A, and C5416</td>
<td>A-19</td>
</tr>
<tr>
<td>A–23</td>
<td>Bank-Switching Control Register (BCR) — C5440 and C5441</td>
<td>A-21</td>
</tr>
<tr>
<td>A–24</td>
<td>General Purpose I/O Register (GPIO)</td>
<td>A-22</td>
</tr>
<tr>
<td>A–25</td>
<td>General Purpose I/O Control Register (GPIOCR)</td>
<td>A-24</td>
</tr>
<tr>
<td>A–26</td>
<td>General Purpose Status Register (GPIOSR)</td>
<td>A-25</td>
</tr>
<tr>
<td>A–27</td>
<td>HPI Control Register (HPIC) (for 5410)</td>
<td>A-25</td>
</tr>
<tr>
<td>A–28</td>
<td>McBSP Serial Port Control Register 1 (SPCR1)</td>
<td>A-26</td>
</tr>
<tr>
<td>A–29</td>
<td>McBSP Serial Port Control Register 2 (SPCR2)</td>
<td>A-28</td>
</tr>
<tr>
<td>A–30</td>
<td>McBSP Pin Control Register (PCR)</td>
<td>A-30</td>
</tr>
<tr>
<td>A–31</td>
<td>Receive Control Register 1 (RCR1)</td>
<td>A-33</td>
</tr>
<tr>
<td>A–32</td>
<td>Receive Control Register 2 (RCR2)</td>
<td>A-34</td>
</tr>
<tr>
<td>A–33</td>
<td>Transmit Control Register 1 (XCR1)</td>
<td>A-35</td>
</tr>
<tr>
<td>A–34</td>
<td>Transmit Control Register 2 (XCR2)</td>
<td>A-36</td>
</tr>
</tbody>
</table>
Figures

A–35 Sample Rate Generator Register 1 (SRGR1) ................................................................. A-38
A–36 Sample Rate Generator Register 2 (SRGR2) ................................................................. A-38
A–37 Multichannel Control Register 1 (MCR1) ................................................................. A-40
A–38 Multichannel Control Register 2 (MCR2) ................................................................. A-41
A–39 Receive Channel Enable Register (RCERn) ............................................................... A-43
A–40 Transmit Channel Enable Register (XCERn) ............................................................ A-45
A–41 Clock Mode Register (CLKMD) ................................................................. A-47
A–42 Timer Control Register (TCR) ................................................................. A-49
A–43 Timer Secondary Control Register (TSCR) — C5440, C5441, and C5471 ............ A-51
A–44 Timer Period Register (PRD) ................................................................. A-51
A–45 Watchdog Timer Control Register (WDTCR) .......................................................... A-52
A–46 Watchdog Timer Secondary Control Register (WDTSCR) ........................................ A-54
A–47 Watchdog Timer Period Register (WDPRD) .......................................................... A-55
<table>
<thead>
<tr>
<th>Page</th>
<th>Description</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>1-9</td>
<td>Device-Specific Features Support</td>
</tr>
<tr>
<td>2-1</td>
<td>CSL Directory Structure</td>
</tr>
<tr>
<td>3-1</td>
<td>CHIP Functions</td>
</tr>
<tr>
<td>4-1</td>
<td>DAA Configuration Structures</td>
</tr>
<tr>
<td>4-2</td>
<td>DAA Functions</td>
</tr>
<tr>
<td>4-3</td>
<td>DAA 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 CSL Macros (using channel number)</td>
</tr>
<tr>
<td>6-4</td>
<td>DMA CSL Macros (using handles)</td>
</tr>
<tr>
<td>7-1</td>
<td>EBUS Configuration Structure</td>
</tr>
<tr>
<td>7-2</td>
<td>EBUS Functions</td>
</tr>
<tr>
<td>7-3</td>
<td>EBUS Macros</td>
</tr>
<tr>
<td>8-1</td>
<td>GPIO Functions</td>
</tr>
<tr>
<td>8-2</td>
<td>GPIO Macros (C544x devices only)</td>
</tr>
<tr>
<td>9-1</td>
<td>HPI Macros</td>
</tr>
<tr>
<td>10-1</td>
<td>IRQ Configuration Structure</td>
</tr>
<tr>
<td>10-2</td>
<td>IRQ Functions</td>
</tr>
<tr>
<td>10-3</td>
<td>IRQ_EVT_NNNN Event List</td>
</tr>
<tr>
<td>11-1</td>
<td>McBSP Configuration Structure</td>
</tr>
<tr>
<td>11-2</td>
<td>McBSP Functions</td>
</tr>
<tr>
<td>11-3</td>
<td>MCBSP CSL Macros (using port number)</td>
</tr>
<tr>
<td>11-4</td>
<td>MCBSP CSL Macros (using handle)</td>
</tr>
<tr>
<td>12-1</td>
<td>PLL Configuration Structure</td>
</tr>
<tr>
<td>12-2</td>
<td>PLL Functions</td>
</tr>
<tr>
<td>12-3</td>
<td>PLL CSL Macros</td>
</tr>
<tr>
<td>13-1</td>
<td>PWR Functions</td>
</tr>
</tbody>
</table>

xvi
14–1 TIMER Configuration Structure ........................................ 14-2
14–2 TIMER Functions ...................................................... 14-2
14–3 TIMER CSL Macros Using Timer Port Number ................... 14-10
14–4 TIMER CSL Macros Using Handle .................................. 14-11
15–1 UART APIs ............................................................... 15-2
15–2 UART CSL Macros ...................................................... 15-15
16–1 WDTIM Configuration Structure ...................................... 16-2
16–2 WDTIM Functions ...................................................... 16-2
16–3 WDTIM CSL Macros Using Timer Port Number ................... 16-7
A–1 DMA Channel Priority and Enable Control Register (DMPREC) Field Values (DMA_DMPREC_field_symval) A-2
A–2 DMA Channel n Sync Select and Frame Count Register (DMSFCn) Field Values (DMA_DMSFC_field_symval) A-3
A–3 DMA Channel n Transfer Mode Control Register (DMMCRn) Field Values (DMA_DMMCR_field_symval) A-7
A–4 DMA Channel n Source Address Register (DMSRCn) Field Values (DMA_DMSRC_field_symval) A-9
A–5 DMA Global Source Address Reload Register (DMGSA) Field Values (DMA_DMGSA_field_symval) A-9
A–6 DMA Source Program Page Address Register (DMSRCP) Field Values (DMA_DMSRCP_field_symval) A-10
A–7 DMA Channel n Destination Address Register (DMDSTn) Field Values (DMA_DMDST_field_symval) A-10
A–8 DMA Global Destination Address Reload Register (DMGDA) Field Values (DMA_DMGDA_field_symval) A-10
A–9 DMA Destination Program Page Address Register (DMDSTP) Field Values (DMA_DMDSTP_field_symval) A-10
A–10 DMA Channel n Element Count Register (DMCTRn) Field Values (DMA_DMCTR_field_symval) A-11
A–11 DMA Global Element Count Reload Register (DMGCR) Field Values (DMA_DMGCR_field_symval) A-12
A–12 DMA Global Frame Count Reload Register (DMGFR) Field Values (DMA_DMGFR_field_symval) A-12
A–13 DMA Element Address Index Register 0 (DMIDX0) Field Values (DMA_DMIDX0_field_symval) A-13
A–14 DMA Element Address Index Register 1 (DMIDX1) Field Values (DMA_DMIDX1_field_symval) A-13
A–15 DMA Frame Address Index Register 0 (DMFRI0) Field Values (DMA_DMFRI0_field_symval) A-14
A–16 DMA Frame Address Index Register 1 (DMFRI1) Field Values (DMA_DMFRI1_field_symval) A-14
A–17 DMA Global Extended Source Data Page Register (DMSRCDP) Field Values (DMA_DMSRCDP_field_symval) A-15
A–18 DMA Global Extended Destination Data Page Register (DMDSTDP) Field Values (DMA_DMDSTDP_field_symval) A-15
A–19 Software Wait-State Register (SWWSR) Field Values (EBUS_SWWSR_field_symval) A-16
A–20 Software Wait-State Control Register (SWCR) Field Values (EBUS_SWWCR_field_symval) A-17
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-21</td>
<td>Bank-Switching Control Register (BSCR) Field Values — C5401, C5402, C5409, and C5420, and C5471 (EBUS_BSCR_field_symval)</td>
</tr>
<tr>
<td>A-22</td>
<td>Bank-Switching Control Register (BSCR) Field Values — C5410, and C5416 (EBUS_BSCR_field_symval)</td>
</tr>
<tr>
<td>A-23</td>
<td>Bank-Switching Control Register (BSCR) Field Values — C5440 and C5441 (EBUS_BSCR_field_symval)</td>
</tr>
<tr>
<td>A-24</td>
<td>General Purpose I/O Register (GPIO) Field Values (GPIO_GPIO_field_symval)</td>
</tr>
<tr>
<td>A-25</td>
<td>General Purpose I/O Control Register (GPIOCR) Field Values (HPI_GPIOCR_field_symval)</td>
</tr>
<tr>
<td>A-26</td>
<td>General Purpose I/O Status Register (GPIOSR) Field Values (HPI_GPIOSR_field_symval)</td>
</tr>
<tr>
<td>A-27</td>
<td>McBSP Serial Port Control Register 1 (SPCR1) Field Values (MCBSP_SPCR1_field_symval)</td>
</tr>
<tr>
<td>A-28</td>
<td>McBSP Serial Port Control Register 2 (SPCR2) Field Values (MCBSP_SPCR2_field_symval)</td>
</tr>
<tr>
<td>A-29</td>
<td>McBSP Pin Control Register (PCR) Field Values (MCBSP_PCR_field_symval)</td>
</tr>
<tr>
<td>A-30</td>
<td>Receive Control Register 1 (RCR1) Field Values (MCBSP_RCR1_field_symval)</td>
</tr>
<tr>
<td>A-31</td>
<td>Receive Control Register 2 (RCR2) Field Values (MCBSP_RCR2_field_symval)</td>
</tr>
<tr>
<td>A-32</td>
<td>Transmit Control Register 1 (XCR1) Field Values (MCBSP_XCR1_field_symval)</td>
</tr>
<tr>
<td>A-33</td>
<td>Transmit Control Register 2 (XCR2) Field Values (MCBSP_XCR2_field_symval)</td>
</tr>
<tr>
<td>A-34</td>
<td>Sample Rate Generator Register 1 (SRGR1) Field Values (MCBSP_SRGR1_field_symval)</td>
</tr>
<tr>
<td>A-35</td>
<td>Sample Rate Generator Register 2 (SRGR2) Field Values (MCBSP_SRGR2_field_symval)</td>
</tr>
<tr>
<td>A-36</td>
<td>Multichannel Control Register 1 (MCR1) Field Values (MCBSP_MCR1_field_symval)</td>
</tr>
<tr>
<td>A-37</td>
<td>Multichannel Control Register 2 (MCR2) Field Values (MCBSP_MCR2_field_symval)</td>
</tr>
<tr>
<td>A-38</td>
<td>Receive Channel Enable Register (RCERn) Field Values (MCBSP_RCERn_field_symval)</td>
</tr>
<tr>
<td>A-39</td>
<td>Transmit Channel Enable Register (XCERn) Field Values (MCBSP_XCERn_field_symval)</td>
</tr>
<tr>
<td>A-40</td>
<td>Clock Mode Register (CLKMD) Field Values (PLL_CLKMD_field_symval)</td>
</tr>
<tr>
<td>A-41</td>
<td>Timer Control Register (TCR) Field Values (TIMER_TCR_field_symval)</td>
</tr>
<tr>
<td>A-42</td>
<td>Timer Secondary Control Register (TSCR) Field Values (TIMER_TSCR_field_symval)</td>
</tr>
<tr>
<td>A-43</td>
<td>Timer Period Register (PRD)</td>
</tr>
<tr>
<td>A-44</td>
<td>Watchdog Timer Control Register (WDTCR) Field Values (WDTIM_WDTCR_field_symval)</td>
</tr>
<tr>
<td>A-45</td>
<td>Watchdog Timer Secondary Control Register (WDTSCR) Field Values (WDTIM_WDTSCR_field_symval)</td>
</tr>
<tr>
<td>A-46</td>
<td>Watchdog Timer Period Register (WDPRD)</td>
</tr>
</tbody>
</table>
### Contents

1–1 Using **PER_config** or **PER_configArgs** .......................................................... 1-10
2–1 Using a **Linker Command File** ............................................................................. 2-25
6–1 **DMA Header File** .............................................................................................. 6-37
6–2 **DMA Source File** (Declaration Section) ................................................................. 6-38
6–3 **DMA Source File** (Body Section) .......................................................................... 6-39
8–1 **GPIO Source File** (Body Section) .......................................................................... 8-8
10–1 Manual Setting Outside **DSPBIOS HWIs** ............................................................... 10-7
11–1 **McBSP Header File** ........................................................................................... 11-48
11–2 **McBSP Source File** (Declaration Section) ............................................................. 11-48
11–3 **McBSP Source File** (Body Section) ..................................................................... 11-49
11–4 McBSP Port Initialization using **MCBSP_config** .................................................. 11-50
12–1 **PLL Header File** ................................................................................................ 12-11
12–2 **PLL Source File** (Declaration Section) ................................................................. 12-11
12–3 **PLL Source File** (Body Section) ......................................................................... 12-12
14–1 **Timer Header File** .............................................................................................. 14-16
14–2 **Timer Source File** (Declaration Section) ............................................................... 14-16
14–3 **Timer Source File** (Body Section) ....................................................................... 14-17
15–1 **UART Header File** ............................................................................................. 15-24
15–2 **UART Source File** (Declaration Section) .............................................................. 15-25
15–3 **UART Source File** (Body Section) ...................................................................... 15-25
16–1 **WATCHDOG TIMER Header File** ...................................................................... 16-12
16–2 **WATCHDOG TIMER Source File** (Declaration Section) .................................... 16-12
16–3 **WATCHDOG TIMER Source File** (Body Section) .............................................. 16-12
This chapter introduces the Chip Support Library (CSL), briefly describes its architecture, and provides a generic overview of the collection of functions, macros, and constants that are needed to program DSP peripherals.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Introduction to the CSL</td>
<td>1-2</td>
</tr>
<tr>
<td>1.2 Naming Conventions</td>
<td>1-6</td>
</tr>
<tr>
<td>1.3 Data Types</td>
<td>1-7</td>
</tr>
<tr>
<td>1.4 Functions</td>
<td>1-8</td>
</tr>
<tr>
<td>1.5 Macros</td>
<td>1-11</td>
</tr>
<tr>
<td>1.6 Symbolic Constant Values</td>
<td>1-14</td>
</tr>
<tr>
<td>1.7 Resource Management and the Use of CSL Handles</td>
<td>1-15</td>
</tr>
<tr>
<td>1.8 Support for Device-Specific Features</td>
<td>1-17</td>
</tr>
</tbody>
</table>
1.1 Introduction to the CSL

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

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

1.1.1 Benefits of the CSL

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

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

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

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

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

1.1.2 CSL Architecture

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

Figure 1–1 illustrates the individual CSL modules. This architecture allows for future expansion because new modules can be added as new peripherals emerge.
Although each CSL module provides a unique set of functions, some interdependency exists between the modules. For example, the DMA module depends on the IRQ module because of DMA interrupts; as a result, when you link code that uses the DMA module, a portion of the IRQ module is linked automatically.

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

Table 1–1 lists general and peripheral modules with their associated include file, the module support symbol, and a column that specifies which modules are/are not supported by the CSL GUI. These components must be included in your application.
### Introduction to the CSL

**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>
<th>CSL GUI Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHIP</td>
<td>General device module</td>
<td>csl_chip.h</td>
<td>_CHIP_SUPPORT</td>
<td>NO</td>
</tr>
<tr>
<td>DAA</td>
<td>Digital Access Arrangement</td>
<td>csl_daa.h</td>
<td>_DAA_SUPPORT</td>
<td>NO</td>
</tr>
<tr>
<td>DAT</td>
<td>A data copy/fill module based on the DMA</td>
<td>csl_dat.h</td>
<td>_DAT_SUPPORT</td>
<td>NO</td>
</tr>
<tr>
<td>DMA</td>
<td>DMA Peripheral</td>
<td>csl_dma.h</td>
<td>_DMA_SUPPORT</td>
<td>YES</td>
</tr>
<tr>
<td>EBUS</td>
<td>External bus interface</td>
<td>csl_ebus.h</td>
<td>_EBUS_SUPPORT</td>
<td>YES</td>
</tr>
<tr>
<td>GPIO</td>
<td>Non-multiplexed general purpose I/O</td>
<td>csl_gpio.h</td>
<td>_GPIO_SUPPORT</td>
<td>YES</td>
</tr>
<tr>
<td>HPI</td>
<td>HPI peripheral</td>
<td>csl_hpi.h</td>
<td>_HPI_SUPPORT</td>
<td>NO</td>
</tr>
<tr>
<td>IRQ</td>
<td>Interrupt controller</td>
<td>csl_irq.h</td>
<td>_IRQ_SUPPORT</td>
<td>NO</td>
</tr>
<tr>
<td>MCBSP</td>
<td>Multi-channel buffered serial port</td>
<td>csl_mcbsp.h</td>
<td>_MCBSP_SUPPORT</td>
<td>YES</td>
</tr>
<tr>
<td>PLL</td>
<td>PLL</td>
<td>csl_pll.h</td>
<td>_PLL_SUPPORT</td>
<td>YES</td>
</tr>
<tr>
<td>PWR</td>
<td>Power savings control</td>
<td>csl_pwr.h</td>
<td>_PWR_SUPPORT</td>
<td>NO</td>
</tr>
<tr>
<td>TIMER</td>
<td>Timer peripheral</td>
<td>csl_timer.h</td>
<td>_TIMER_SUPPORT</td>
<td>YES</td>
</tr>
<tr>
<td>UART</td>
<td>Universal Asynchronous Receiver/Transmitter</td>
<td>csl_uart.h</td>
<td>_UART_SUPPORT</td>
<td>YES</td>
</tr>
<tr>
<td>WDTIM</td>
<td>Watchdog Timer</td>
<td>csl_wdtim.h</td>
<td>_WDT_SUPPORT</td>
<td>YES</td>
</tr>
</tbody>
</table>

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

**Note:**

Devices C541 to C549 are NOT supported by CSL.
Table 1-2. CSL Device Support

<table>
<thead>
<tr>
<th>Device</th>
<th>Near-Mode Library</th>
<th>Far-Mode Library</th>
<th>Device Support Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>C5401</td>
<td>csl5401.lib</td>
<td>csl5401x.lib</td>
<td>CHIP_5401</td>
</tr>
<tr>
<td>C5402</td>
<td>csl5402.lib</td>
<td>csl5402x.lib</td>
<td>CHIP_5402</td>
</tr>
<tr>
<td>C5404</td>
<td>csl5404.lib</td>
<td>csl5404x.lib</td>
<td>CHIP_5404</td>
</tr>
<tr>
<td>C5407</td>
<td>csl5407.lib</td>
<td>csl5407x.lib</td>
<td>CHIP_5407</td>
</tr>
<tr>
<td>C5409</td>
<td>csl5409.lib</td>
<td>csl5409x.lib</td>
<td>CHIP_5409</td>
</tr>
<tr>
<td>C5409A</td>
<td>csl5409A.lib</td>
<td>csl5409Ax.lib</td>
<td>CHIP_5409A</td>
</tr>
<tr>
<td>C5410</td>
<td>csl5410.lib</td>
<td>csl5410x.lib</td>
<td>CHIP_5410</td>
</tr>
<tr>
<td>C5410A</td>
<td>csl5410A.lib</td>
<td>csl5410Ax.lib</td>
<td>CHIP_5410A</td>
</tr>
<tr>
<td>C5416</td>
<td>csl5416.lib</td>
<td>csl5416x.lib</td>
<td>CHIP_5416</td>
</tr>
<tr>
<td>C5420</td>
<td>csl5420.lib</td>
<td>csl5420x.lib</td>
<td>CHIP_5420</td>
</tr>
<tr>
<td>C5421</td>
<td>csl5421.lib</td>
<td>csl5421x.lib</td>
<td>CHIP_5421</td>
</tr>
<tr>
<td>C5440</td>
<td>csl5440.lib</td>
<td>csl5440x.lib</td>
<td>CHIP_55440</td>
</tr>
<tr>
<td>C5441</td>
<td>csl5441.lib</td>
<td>csl5441x.lib</td>
<td>CHIP_5441</td>
</tr>
<tr>
<td>C5471</td>
<td>csl5471.lib</td>
<td>csl5471x.lib</td>
<td>CHIP_5471</td>
</tr>
<tr>
<td>C54CST</td>
<td>cslcst.lib</td>
<td>cslcstx.lib</td>
<td>CHIP_54CST</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—all in capital letters—is the Peripheral module name listed in Table 1–1).
- Function names use all small letters. Capital letters are used only if the function name consists of two separate words (e.g., PER_getConfig()).
- Macro names use all capital letters (e.g., DMA_DMPREC_RMK).
- Data types begin with a capital letter, followed by small letters (e.g., DMA_Handle).
1.3 Data Types

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

Table 1-4. CSL Data Types

<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 Functions

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

\begin{table}
\begin{center}
\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{center}
\end{table}

The following conventions are used in Table 1–5:

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

CSL functions provide a way to program peripherals by \textit{direct register initialization} using the \texttt{PER} \texttt{config()} or \texttt{PER} \texttt{configArgs()} functions (see section 1.4.1). This method is used by the CSL GUI.
### 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 that is used 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. You can initialize the configuration structure with:  
  - Integer constants  
  - Integer variables  
  - CSL symbolic constants, `PER_REG_DEFAULT` (see section 1.6, CSL Symbolic Constant Values)  
  - Merged field values created with the `PER_REG_RMK` macro |
| `PER_configArgs(handle,) regval_1, . regval_n` | Writes the individual values (`regval_n`) to the peripheral registers. These values can be any of the following:  
  - Integer constants  
  - Integer variables  
  - CSL symbolic constants, `PER_REG_DEFAULT`  
  - Merged field values created with the `PER_REG_RMK` macro |
| `PER_start(handle,) [txrx,] [delay]` | Starts the peripheral after using `PER_config()` or `PER_configArgs()`. `[txrx]` and `[delay]` apply only to MCBSP. |
| `PER_reset(handle)` | Resets the peripheral to its power-on default values. |
| `PER_close(handle)` | Closes a peripheral channel previously opened with `PER_open()`. The registers for the channel are set to their power-on defaults, and any pending interrupt is cleared. |
1.4.1 Peripheral Initialization via Registers

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

- `PER_config` allows you to initialize a configuration structure with the appropriate register values and pass the address of that structure to the function, which then writes the values to the register. The CSL GUI uses this function to initialize peripherals. Example 1–1 shows an example of this method.

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

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

**Example 1–1. Using PER_config or PER_configArgs**

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

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

- \textit{PER} indicates a peripheral module as listed in Table 1–1.
- \textit{REG} indicates a register name (without the channel number).
- \textit{REG#} indicates, if applicable, a register with the channel number. (e.g., DMPREC, DMSRC1, ...)
- \textit{FIELD} indicates a field in a register.
- \textit{regval} indicates an integer constant, an integer variable, a symbolic constant (\textit{PER\_REG\_DEFAULT}), or a merged field value created with the \textit{PER\_REG\_RMK()} macro.
- \textit{fieldval} indicates an integer constant, integer variable, macro, or symbolic constant (\textit{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 \textit{REG#} to identify which channel the register belongs to, it uses the Handle value. The Handle value is returned by the \textit{PER\_open()} function. The equivalent macros are shown in Table 1–7. Please note that \textit{REG} is the register name without the channel number.
### Table 1–6. Generic CSL Macros

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
</table>
| **PER_REG_RMK**  
  fieldval_15,  
  .  
  .  
  .  
  fieldval_0  
) | Creates a value to store in the peripheral register; _RMK macros make it easier to construct register values based on field values.  
  The following rules apply to the _RMK macros:  
  - Define only when more than one field exists in a register  
  - Include only fields that are writable.  
  - Specify field arguments as most-significant bit first.  
  - Whether or not they are used, all writable field values must be included.  
  - If you pass a field value exceeding the number of bits allowed for that particular field, the _RMK macro truncates that field value. |
| **PER_RGET**  
  (REG#) | Returns the value in the peripheral register. |
| **PER_RSET**  
  (REG#,  
  regval  
) | Writes the value to the peripheral register. |
| **PER_FMK**  
  (REG,  
  FIELD,  
  fieldval  
) | Creates a shifted version of fieldval that you could OR with the result of other _FMK macros to initialize register REG. This allows you to initialize few fields in REG as an alternative to the _RMK macro that requires that ALL the fields in the register be initialized. |
| **PER_FGET**  
  (REG#,  
  FIELD  
) | Returns the value of the specified FIELD in the peripheral register. |
| **PER_FSET**  
  (REG#,  
  FIELD,  
  fieldval  
) | Writes fieldval to the specified FIELD in the peripheral register. |
| **PER_ADDR**  
  (REG#) | If applicable, retrieves the memory address (or sub-address) of the peripheral register REG#. |
### Table 1-7. Generic CSL Macros (Handle-based)

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

1.6 Symbolic Constant Values

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

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

Table 1–8. Generic CSL Symbolic Constants

(a) Constant Values for Registers

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{PER_REG_DEFAULT}</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>\textit{PER_REG_FIELD_SYMVAL}</td>
<td>Symbolic constant to specify values for individual fields in the indicated peripheral register. See Appendix A for the symbolic values.</td>
</tr>
<tr>
<td>\textit{PER_REG_FIELD_DEFAULT}</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. It then returns a pointer to a Handle structure that contains information about which channel (DMA) or port (MCBSP) was opened.

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

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

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

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

1.7.1 Using CSL Handles

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

For example:

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

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

```
myDma = DMA_open(DMA_CHA0, DMA_OPEN_RESET);
/* Open DMA channel 0 */
```
The call to DMA_open initializes the handle, myDma. This handle can then be used in calls to other API functions:

```c
DMA_start(myDma);    /* Begin transfer */
.
.
DMA_close(myDma);    /* Free DMA channel */
```
1.8 Support for Device-Specific Features

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

Table 1–9. Device-Specific Features Support

(a) DMA Module-Channel Reload

<table>
<thead>
<tr>
<th>Individual Channel Register Reload Support</th>
<th>Global Channel Register Reload Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>5416, 5421, 5409a, 5410a, 5440, 5441</td>
<td>All other C54x supported devices</td>
</tr>
</tbody>
</table>

(b) DMA Module-Extended Data Reach

<table>
<thead>
<tr>
<th>Individual Channel Extended Data Memory Support</th>
<th>Global Extended Data Memory Support</th>
<th>No Extended Data Memory Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>5440, 5441</td>
<td>5409, 5416, 5421, 5409a, 5410a</td>
<td>5402, 5404, 5407, 5410, 5420, 5455</td>
</tr>
</tbody>
</table>

(c) MCBSP Module-Channel Support

<table>
<thead>
<tr>
<th>MCBSP 128–Channel Support</th>
<th>MCBSP 32-Channel Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>5416, 5421, 5440, 5409a, 5410a, 5441</td>
<td>All other C54x supported devices</td>
</tr>
</tbody>
</table>

(d) Watchdog Module

<table>
<thead>
<tr>
<th>Watchdog Timer Support</th>
<th>No Watchdog Timer Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>5440, 5441</td>
<td>All other C54x supported devices</td>
</tr>
</tbody>
</table>

(e) Timer Module

<table>
<thead>
<tr>
<th>Timer Extended Pre-Scaler Support</th>
<th>No Timer Extended Pre-Scaler Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>5471, 5441</td>
<td>All other C54x supported devices</td>
</tr>
</tbody>
</table>

(f) Chip Module

<table>
<thead>
<tr>
<th>Device ID Support</th>
<th>No Device ID Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>5416, 5409A, 5410A, 5441, 5421</td>
<td>All other C54x supported devices</td>
</tr>
</tbody>
</table>
Chapter 2

How To Use CSL

This chapter provides instructions on how to:

☐ Use the TMS320C54x CSL Graphic User Interface (GUI) integrated into the DSP/BIOS Configuration Tool. It describes a detailed work-flow for setting and accessing the CSL GUI data via the generation of C code files. The examples shown in this chapter include a complete procedure for creating configuration objects and generating peripheral pre-initialization.

☐ Use the CSL library without the use of the CSL GUI (see section 2.3).

<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 CSL with the CSL GUI</td>
<td>2-3</td>
</tr>
<tr>
<td>2.3 Using CSL without the CSL GUI</td>
<td>2-17</td>
</tr>
<tr>
<td>2.4 Rebuilding CSL</td>
<td>2-26</td>
</tr>
</tbody>
</table>
2.1 Overview

There are two methods to configure peripherals when using CSL:

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

- Configure peripherals by using the CSL GUI.

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

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

2.2.1 CSL GUI: The CSL Tree

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

Figure 2–1. CSL Tree (for 5402)

Figure 2–2 shows an example of the expanded CSL Tree.
Each peripheral is organized into two sections (see Figure 2–2):

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

- **PERIPHERAL Resource Manager** allows you to bind a peripheral configuration to a specific on-chip device, i.e., a DMA channel, a MCBSP port, or a TIMER device. The devices are displayed as pre-defined objects and cannot be deleted or renamed. However, the Handles to these objects can be renamed.
Figure 2–3. Insert Configuration Object

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

Figure 2–4. Delete/Rename Options
The Delete option allows you to delete a configuration object. The Delete option is not accessible if the configuration is used by a physical peripheral of the associated Resource manager.

Figure 2–5. Show Dependency Option

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

Note:
A configuration object cannot be deleted if there is a dependency present.
2.2.2 Steps to configuring a peripheral using CSL GUI

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

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

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

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

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

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

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

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

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

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

**Step 6:** Modify your application code (main.c) (see section 2.2.5)
Using CSL with the CSL GUI

Figure 2–6. Code Composer Studio IDE Project Window

Figure 2–7. Code Composer Studio IDE Project Window with .cdb Project File Added
2.2.3 TIMER Configuration Example

Because the default values (device power-on reset) are different from one chip to another, it is recommended that you delete any existing config objects before changing the chip type under the Global Setting (system). This ensures that you get the right reset value when you want to use the default setting of the registers.

The configuration file mytimer.cdb is assumed to be created previously and opened (see section 2.2.2).

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

Figure 2–8. CCStudio IDE Project View

Using Figure 2–9 follow these steps:

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

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

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

- Check the Open Handle to Timer and Enable pre-initialization
From the pre-initialize drop-down list, select the configuration, timerCfg0.

**Figure 2–9. Configuring the TIMER1 Device**
2.2.4 Files Generated by the CSL GUI

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

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

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

- Header file: `mytimercfg.h`
- Source file: `mytimercfg_c.c`

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

2.2.4.1 Header File: `mytimercfg.h`

This header file must be added to one of the user’s source files in order to access the CSL functions and/or objects predefined in the DSP/BIOS configuration tool.

This header file contains several elements:

- The definition of the chip:
  ```
  #define CHIP_5402 1
  ```
- The csl header files used by the CSL tree
- The declaration list of the CSL handle objects and configuration names defined in the `mytimer_c.c`

  These are declared external, and can be applied in the user’s code:
  ```
  extern TIMER_Config timerCfg1;
  extern TIMER_Handle hTimer1;
  #include <projectcfg.h>
  ```

Figure 2–10 illustrates the setup of `mytimercfg_c.c`
**Figure 2–10. Header File mytimercfg.h**

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

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

#ifdef __cplusplus
extern "C" {
#endif
extern far HST_Obj RTA_fromHost;
extern far HST_Obj RTA_toHost;
extern far SWI_Obj KNL_swic;
extern far TSK_Obj TSK_idle;
extern far LOG_Obj LOG_system;
extern far STS_Obj IDL_busyObj;
extern far TIMER_Config timerCfg0;
extern far TIMER_Handle htimer1;
extern far void CSL_cfgInit();

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

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

**2.2.4.2 Source File: mytimercfg_c.c**

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

- **Include section**

  This section defines the project header file. This allows `mytimercfg_c.c` access to the data declared in the header file.

  ```c
  #include <mytimercfg.h>
  ```
Note:
This line is added before the other CSL header files (csl_emif.h, csl_timer.h, ...). This is done so that you are not required to specify the device number under the Project option (that –dCHIP_54xx is not required).

- Declaration section

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

- Code section

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

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

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

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

In the example shown in Figure 2–11,
- If Enable pre-initialization is checked, a call to the TIMER_config() function is generated.
- If Enable Pre-initialization is unchecked, a call to the TIMER_config() is not generated, but the configuration structure timerCfg1 is created and available for you to use at a later time.
Figure 2–11. Resource Manager Properties Page

![Resource Manager Properties Page](image)

Figure 2–12 illustrates the mytimercfg_c.c file.

```c
void CSL_cfgInit()
{
    hTimer1 = TIMER_open(TIMER_DEV1, TIMER_OPEN_RESET);
    ...
    TIMER_config(hTimer1, &timerCfg1);
}
```
Figure 2–12. Source File mytimercfg_c.c

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

/* INPUT mytimer.cdb */

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

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

/* Handles */
TIMER_Handle hTimer1;

/* ========= CSL_cfgInit() ======== */
void CSL_cfgInit()
{
  CSL_init();
  hTimer1 = TIMER_open(TIMER_DEV1, TIMER_OPEN_RESET);
  TIMER_config(hTimer1, &timerCfg0);
}
```

**2.2.5 Modifying Your Application Code (main.c)**

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

```
#include "mytimercfg.h"
```

This includes CSL handle and configuration declarations to your code.

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

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

Figure 2–13 illustrates how to access the files generated by the CSL GUI in your application code (main.c)
Figure 2–13. Example of main.c File Using Data Generated by the CSL GUI

```c
#include <mytimercfg.h>
static Uint32 TIMEREventId1;
void main() {

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

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

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

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

    /* Close TIMER */
    TIMER_close(hTimer1);
}
```

This line allows the objects created with the CSL GUI tool to be referenced here. The header file must be included before any other CSL header files; otherwise, the -dCHIP_XXXX option must be set in the compiler option window.

Handle object “hTimer1” is used directly by the TIMER CSL APIs.
2.3 Using CSL without the CSL GUI

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

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

There are two ways to program peripherals using CSL:

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

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

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

Source address:   2000h in data space  
Destination address:   3000h in data space  
Transfer size:       Sixteen 16-bit single words
Using CSL without the CSL GUI

2.3.1 Using the DMA_config() function

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

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

```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 */
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.

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

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

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

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

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

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

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

Step 8: Close DMA channel

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

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

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

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

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

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

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

The remaining sections in this chapter will provide more details and explanations for the steps above.
Note:
CCStudio will automatically define the search paths for include files and libraries as defined in Table 2–1. You are not required to set the –i option.

Table 2–1. CSL Directory Structure

<table>
<thead>
<tr>
<th>This CSL component...</th>
<th>Is located in this directory...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Libraries</td>
<td>c:\ti\c5400\bios\lib</td>
</tr>
<tr>
<td>Source Library</td>
<td>c:\ti\c5400\bios\lib</td>
</tr>
<tr>
<td>Include files</td>
<td>c:\ti\c5400\bios\include</td>
</tr>
<tr>
<td>Examples</td>
<td>c:\ti\examples&lt;target&gt;\csl</td>
</tr>
<tr>
<td>Documentation</td>
<td>c:\ti\docs</td>
</tr>
</tbody>
</table>

2.3.2.1 Specifying Your Target Device

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

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

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

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

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

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

Step 5: Click OK.
2.3.2.2 Selecting Far/Near Mode

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

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

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

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

**Step 4:** Select Use Far Calls.

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

**Step 6:** Click OK.
**Using CSL without the CSL GUI**

**Figure 2–15. Defining Far Mode**

![Defining Far Mode](image)

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

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

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

**Step 10:** In the Category list, highlight Basic.

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

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

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

**Step 12:** Click OK.
2.3.2.3 Large/Small Memory Model Selection

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

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

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

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

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

**Step 5:** Click OK.

*Figure 2–16. Defining Large Memory Model*
Then, you must specify which CSL and RTS libraries will be linked in your project.

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

Figure 2–17. Defining Library Paths
2.3.2.4 Creating a Linker Command File

The CSL has two requirements for the linker command file:

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

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

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

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

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

**Example 2–1. Using a Linker Command File**

```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
}
```

2.3.2.5 Using Function Inlining

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

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

```
mk500  csl54xx.src  -dCHIP_5402 -v548 -mf
```
The CSL CHIP module offers general CPU functions and macros for C54x register accesses. The CHIP module is not handle-based.
3.1 Overview

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

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

Table 3–1. CHIP Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
<th>See page ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHIP_getCpuId</td>
<td>Returns the CPU ID field of the CSIDR register.</td>
<td>3-3</td>
</tr>
<tr>
<td>CHIP_getMapMode</td>
<td>Returns the current MAP mode of the device.</td>
<td>3-3</td>
</tr>
<tr>
<td>CHIP_getRevId</td>
<td>Returns the CPU revision ID.</td>
<td>3-4</td>
</tr>
<tr>
<td>CHIP_getSubsysId</td>
<td>Returns sub-system ID (or core) for a multi-core device.</td>
<td>3-4</td>
</tr>
</tbody>
</table>
3.2 Functions

This section lists the functions in the CHIP module.

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Uint32 CHIP_getCpuId();</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Return Value</td>
<td>CPU ID Returns the CPU (CHIP) ID Field</td>
</tr>
<tr>
<td>Description</td>
<td>This function returns the CPU (CHIP) ID field of the CSIDR register.</td>
</tr>
<tr>
<td>Example</td>
<td>Uint32 CpuId;</td>
</tr>
<tr>
<td></td>
<td>CpuId = CHIP_getCpuId();</td>
</tr>
</tbody>
</table>

**CHIP_getMapMode**  
Reads the map mode bits

<table>
<thead>
<tr>
<th>Function</th>
<th>Uint16 CHIP_getMapMode();</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Return Value</td>
<td>map mode Returns current device MAP mode, which will be one of the following:</td>
</tr>
<tr>
<td></td>
<td>CHIP_MAP_0: MP/MC DROM and OVLY bits are OFF</td>
</tr>
<tr>
<td></td>
<td>CHIP_MAP_1: DROM bit is on</td>
</tr>
<tr>
<td></td>
<td>CHIP_MAP_2: OVLY bit is on</td>
</tr>
<tr>
<td></td>
<td>CHIP_MAP_3: Both DROM and OVLY Bits are on</td>
</tr>
<tr>
<td></td>
<td>CHIP_MAP_4: MP/MC bit is on</td>
</tr>
<tr>
<td></td>
<td>CHIP_MAP_5: MP/MC and DROM are on</td>
</tr>
<tr>
<td></td>
<td>CHIP_MAP_6: MP/MC and OVLY bits are on</td>
</tr>
<tr>
<td></td>
<td>CHIP_MAP_7: MP/MC, DROM, and OVLY bits are on</td>
</tr>
<tr>
<td>Description</td>
<td>Reads the map mode bits (OVLY, DROM, MPMC) from the device. In devices not supported by a specific map-mode bit, the value returned is invalid. See the specific device data sheet for the availability of map mode bits. This function is useful for debugging purposes.</td>
</tr>
<tr>
<td>Example</td>
<td>Uint16 MapMode;</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>MapMode = CHIP_getMapMode();</td>
</tr>
<tr>
<td></td>
<td>if (MapMode == CHIP_MAP_0) {</td>
</tr>
<tr>
<td></td>
<td>/* do map 0 tasks */</td>
</tr>
<tr>
<td></td>
<td>} else {</td>
</tr>
<tr>
<td></td>
<td>/* do map 1 tasks */</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>
CHIP_getRevId

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

**Function**  
Uint32 CHIP_getRevId();

**Arguments**  
None

**Return Value**  
Revision ID  
Returns CPU revision ID

**Description**  
This function returns the CPU revision ID as determined by the Revision ID field of the CSIDR register.

**Example**  
Uint32 RevId;  
RevId = CHIP_getRevId();

CHIP_getSubsysId

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

**Function**  
Uint32 CHIP_getSubsysId();

**Arguments**  
None

**Return Value**  
Subsysem ID

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

**Example**  
Uint32 RevId;  
RevId = CHIP_getSubsysId();
This chapter contains general descriptions for the DAA (Data Access Arrangement) configuration structures, functions, and macros.

<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 Configuration Structures</td>
<td>4-4</td>
</tr>
<tr>
<td>4.3 Functions</td>
<td>4-7</td>
</tr>
<tr>
<td>4.4 Macros</td>
<td>4-11</td>
</tr>
</tbody>
</table>
4.1 Overview

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

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

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

4.1.1 Configuration Structures

Table 4–1. DAA Configuration Structures

<table>
<thead>
<tr>
<th>Configuration Structure</th>
<th>Purpose</th>
<th>See page ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAA_CircBufCtrl</td>
<td>Circular buffer control structure</td>
<td>4-4</td>
</tr>
<tr>
<td>DAA_Params</td>
<td>Parameters structure</td>
<td>4-4</td>
</tr>
<tr>
<td>DAA_PrivateObject/DAA/</td>
<td>Private object/Handle structure</td>
<td>4-5</td>
</tr>
<tr>
<td>DAA_Handle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DAA_DevSetup</td>
<td>Device setup structure</td>
<td>4-6</td>
</tr>
<tr>
<td>DAA_Setup</td>
<td>DAA setup structure</td>
<td>4-6</td>
</tr>
</tbody>
</table>
4.1.2 Functions

Table 4–2. DAA Functions

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

4.1.3 Macros

Table 4–3. DAA Macros

<table>
<thead>
<tr>
<th>Macros</th>
<th>Purpose</th>
<th>See page ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAA_RGETH</td>
<td>Macro to read DAA register</td>
<td>4-11</td>
</tr>
<tr>
<td>DAA_RSETH</td>
<td>Macro to write a value to a DAA register</td>
<td>4-11</td>
</tr>
<tr>
<td>DAA_ADDR</td>
<td>Returns a DAA register address</td>
<td>4-12</td>
</tr>
<tr>
<td>DAA_FMK</td>
<td>Returns a 16-bit register value for REG with FIELD set to VAL</td>
<td>4-12</td>
</tr>
</tbody>
</table>
4.2 Configuration Structures

(Example of structures are included in the Functions section)

**DAA_CircBufCtrl**

**DAA Circular Buffer Control Structure**

**Members**

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

**Description**

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

**DAA_Params**

**DAA Parameters Structure**

**Members**

- `Uint16 txAttenuation /* analog transmit attenuation, valid attenuation values are: */`

  - `DAA_GCR_ATX_ATT_0db`
  - `DAA_GCR_ATX_ATT_3db`
  - `DAA_GCR_ATX_ATT_6db`
  - `DAA_GCR_ATX_ATT_9db`
  - `DAA_GCR_ATX_ATT_12db`

- `Uint16 rxGain /* cid receive attenuation/analog receive gain, valid */`  
  - `DAA_GCR_ARX_ATT_0db`
  - `DAA_GCR_ARX_ATT_1db`
  - `DAA_GCR_ARX_ATT_2.2db`
  - `DAA_GCR_ARX_ATT_3.5db`
  - `DAA_GCR_ARX_ATT_5db`
  - `DAA_GCR_ARX_GAIN_0db`
  - `DAA_GCR_ARX_GAIN_3db`
  - `DAA_GCR_ARX_GAIN_6db`
  - `DAA_GCR_ARX_GAIN_9db`
  - `DAA_GCR_ARX_GAIN_12db`

- `Uint16 sampleRateReg7 /* sample rate control - register 7, valid values are: */`

  - `DAA_SRCTRL_SRC_7200`
The user must create this DAA parameters structure, initialize all members, and include it in the DAA single device setup structure (DAA_DevSetup). One parameter structure may be used for multiple DAA devices.
The user must create this private object/handle structure, initialize all members, and include it in the DAA single device setup structure (DAA_DevSetup). One private object structure must be created for each DAA device. This private object/handle serves as the state machine for the DAA device.

DAA_DevSetup

**Members**

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

**Description**

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

DAA_Setup

**Members**

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

**Description**

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

**void DAA_setup**  
*Pointer to DAA setup structure*

<table>
<thead>
<tr>
<th>Arguments</th>
<th>DAA_Setup *</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return Value</td>
<td>None</td>
</tr>
</tbody>
</table>

**Description**

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

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

**Example**

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

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

<table>
<thead>
<tr>
<th><strong>Handle to DAA device</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arguments</strong></td>
</tr>
<tr>
<td><strong>Return Value</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
</tbody>
</table>
| **Example** | DAA_Handle handle;
                DAA_close(handle); |

### DAA_reset

<table>
<thead>
<tr>
<th><strong>DAA flag to select hardware</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arguments</strong></td>
</tr>
<tr>
<td><strong>Return Value</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><strong>Example</strong></td>
</tr>
</tbody>
</table>
DAA_readWrite

Copies size words from the data read/write buffer

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

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

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

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

DAA_availability

Returns available words in the circular buffer

Arguments
- DAA_Handle Handle to DAA device

Return Value
- Int16 Number of available words in circular buffer.

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

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

DAA_resetInternalBuffer

Resets the pointers of the internal circular buffer

Arguments
- DAA_Handle Handle to DAA device

Return Value
- None.

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

Example
```c
DAA_Handle handle;
DAA_resetInternalBuffer(handle);
```
DAA_delay

**DAA_delay**

*Implements a delay of count McBSP frame syncs*

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

**Return Value**
- `CSLBool` TRUE Indicates “Accepted” and FALSE “rejected”.

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

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

DAA_isr

**DAA_isr**

*DAA interrupt service routine*

**Arguments**
- `DAA_Handle` Handle to DAA device

**Return Value**
None.

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

**Example**
```c
DAA_Handle handle;
Interrupt void myDAAIsr(void)
{
    DAA_isr(handle);
}
```
4.4 Macros

**DAA_RGETH**  
*Macros used to read a DAA register*

**Arguments**
- `DAA_Handle` Handle to DAA device
- `Reg` DAA register name

**Return Type**
- `CSLBool` TRUE Indicates “Accepted” and FALSE “rejected”.

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

**Example**
```c
Void main(void)
{
    DAA_Handle handle;
    While(!DAA_RGETH(handle, DAACTRL1));
    ...
    ...
}

void ctrlCB(void* pID, Uint16 task, Uint16 arg)
{
    if (task & _DAA_REG_READ)
    {
        printf(“DAA control 1 register has a value of %d
”, &arg);
    }
    ...
}
```

**DAA_RSETH**  
*Macro used to write a value to a DAA register*

**Arguments**
- `DAA_Handle` Handle to DAA device
- `REG` DAA register name
- `VAL` Register value to be written

**Return Type**
- `CSLBool` TRUE Indicates “Accepted” and FALSE “rejected”.

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

Example

```c
Void main(void)
{
    DAA_Handle handle;
    Uint16 val = DAA_DAACTRL1_OH_OFFHOOK;
    /* Take DAA off-hook */
    While(!DAA_RSETH(handle, DAACTRL1, val));
    ...
    ...
}
```

```c
void ctrlCB(void* pID, Uint16 task, Uint16 arg)
{
    if (task & _DAA_REG_WRITE)
    {
        printf("DAA control 1 register write complete!\n");
    }
}
```

---

**DAA_ADDR**

**Macro used to return the address of a DAA register**

**Arguments**
- REG DAA register name e.g SRCTRL

**Return Type**
- REGADDR Register address

**Description**
This macro returns the address of a DAA register.

**Example**

```c
reg5Addr = DAA_ADDR(DAACTRL1);
```

---

**DAA_FMK**

**Macro used to return 16-bit register values**

**Arguments**
- REG DAA register name
- FIELD Field name
- VAL Field value

**Return Type**
- REGADDR Register address

**Description**
This macro returns a 16-bit register value for `REG` with `FIELD` set to `VAL`.

**Example**

```c
offHook = DAA_FMK(DAACTRL1, OH, 1);
```
The handle-based DAT (data) 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>5-4</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 DMA regardless of the type of DMA controller; therefore, any application code using the DAT module is compatible across all devices as long as the DMA supports the specific address reach and memory space.

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

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

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

### Note:

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

2) **Memory Spaces:**
   The DAT module contains functions to copy data from one location to another and to fill a region of memory in program, data, or I/O space valid for the specific device (Refer to the C54x data sheets). CSL does not perform any searches for invalid memory addresses.
### Table 5–1. DAT Functions

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

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

**DAT_close**

*Closes the DAT module*

**Function**

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

**Arguments**

- `hDat` Handle to a DAT channel (obtained via `DAT_open`)

**Return Value**

None

**Description**

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

**Example**

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

**DAT_copy**

*Copies linear block of data from src to dst*

**Function**

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

**Arguments**

- `hDat` Handle to a DAT channel (obtained via `DAT_open`)
- `src` Source address ORed with any of the following memory space symbols:
  - `DAT_PROGRAM_SPACE` (Not valid for devices 5402, 5471, and 5472)
  - `DAT_DATA_SPACE`
  - `DAT_IO_SPACE` (Not valid for devices 5402, 5471, and 5472)

For example:
  - `0x10000 | DAT_PROGRAM_SPACE` indicates address 0x10000 in program space
  - `0x10000 | DAT_DATA_SPACE` indicates address 0x10000 in data space
  - `0x100 | DAT_IO_SPACE` indicates address 0x100 in I/O space;
### DAT_copy

<table>
<thead>
<tr>
<th>dst</th>
<th>Destination address ORed with a memory space symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>ElemCnt</td>
<td>Number of 16-bit words to copy</td>
</tr>
</tbody>
</table>

#### Return Value

<table>
<thead>
<tr>
<th>DMA status</th>
<th>Returns status of data transfer at the moment of exiting the routine:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: transfer complete</td>
<td></td>
</tr>
<tr>
<td>1: on-going transfer</td>
<td></td>
</tr>
</tbody>
</table>

#### Description

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

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

#### Example

```c
#define DATA_SIZE 256 // number of 16-bit elements to transfer
Uint16 BuffA[DATA_SIZE];
Uint16 BuffB[DATA_SIZE];
DAT_Handle hDat;
main() {
    ...
    hDat = DAT_open(DAT_CHA_ANY,DAT_PRI_LOW,0);
    DAT_copy(
        hDat,
        (Uint32) (&BuffA) | DAT_DATA_SPACE,
        (Uint32) (&BuffB) | DAT_DATA_SPACE,
        DATA_SIZE
    );
    ...
}
```
DAT_copy2D

Copies data from src to dst

Function

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

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDat</td>
<td>Handle to a DAT channel (obtained via DAT_open)</td>
</tr>
<tr>
<td>Type</td>
<td>Type of 2D DMA transfer, must be one of the following:</td>
</tr>
<tr>
<td></td>
<td>☑ DAT_1D2D : 1D to 2D transfer</td>
</tr>
<tr>
<td></td>
<td>☑ DAT_2D1D : 2D to 1D transfer</td>
</tr>
<tr>
<td></td>
<td>☑ DAT_2D2D : 2D to 2D transfer</td>
</tr>
<tr>
<td>src</td>
<td>Pointer to source ORed with any of the following memory space symbols:</td>
</tr>
<tr>
<td></td>
<td>☑ DAT_PROGRAM_SPACE (Not valid for devices 5402, 5471, and 5472)</td>
</tr>
<tr>
<td></td>
<td>☑ DAT_DATA_SPACE</td>
</tr>
<tr>
<td></td>
<td>☑ DAT_IO_SPACE   (Not valid for devices 5402, 5471, and 5472)</td>
</tr>
<tr>
<td>dst</td>
<td>Pointer to destination address ORed with a memory space symbol</td>
</tr>
<tr>
<td>LineLen</td>
<td>Number of 16-bit words to copy for each line</td>
</tr>
<tr>
<td>LineCnt</td>
<td>Number of lines to copy</td>
</tr>
<tr>
<td>LinePitch</td>
<td>Pitch of each line, number of 16-bit words</td>
</tr>
</tbody>
</table>

Return Value

DMA status Returns status of data transfer at the moment of exiting the
DAT_copy2D

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

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

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

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

Fills linear block of memory with specified fill value

**Function**

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

**Arguments**

- **hDat** Handle to a DAT channel
- **dst** Destination address ORed with any of the following memory space symbols:
  - DAT_PROGRAM_SPACE
  - DAT_DATA_SPACE
  - DAT_IO_SPACE
  For example:
  - 0x10000 | DAT_PROGRAM_SPACE indicates address 0x1000 in program space;
  - 0x10000 | DAT_DATA_SPACE indicates address 0x10000 in data space;
  - 0x100 | DAT_IO_SPACE indicates address 0x100 in I/O space;
- **ElemCnt** Number of bytes to fill (must be power of 2)
- **Value** fill value

**Return Value**

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

**Description**

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

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

**Example**

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

Opens a DAT module

Function

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

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChaNum</td>
<td>Specifies which DMA channel to allocate; must be one of the following:</td>
</tr>
<tr>
<td></td>
<td>- DAT_CHA_ANY (allocates Channel 2 or 3)</td>
</tr>
<tr>
<td></td>
<td>- DAT_CHA2</td>
</tr>
<tr>
<td></td>
<td>- DAT_CHA3</td>
</tr>
<tr>
<td>Priority</td>
<td>Specifies the priority of the DMA channel, must be one of the following:</td>
</tr>
<tr>
<td></td>
<td>- DAT_PRI_LOW sets the DMA channel for low priority level</td>
</tr>
<tr>
<td></td>
<td>- DAT_PRI_HIGH sets the DMA channel for high priority level</td>
</tr>
<tr>
<td>Flags</td>
<td>Miscellaneous open flags (currently None available).</td>
</tr>
</tbody>
</table>

Return Value

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

Description

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

Example 1

To open a DAT channel using any available DMA channel (2 or 3 only) in low priority mode:
```c
DAT_Handle hdat;
hdat = DAT_open(DAT_CHA_ANY, DAT_PRI_LOW, 0);
```

Example 2

To open the DAT channel using DMA channel 2 in high priority mode:
```c
DAT_Handle hdat;
hdat = DAT_open(DAT_CHA2, DAT_PRI_HIGH, 0);
```
**DAT_wait**

*Waits for previous transfer to complete*

**Function**

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

**Arguments**

- **hDat**  Handle to a DAT channel

**Return Value**

None

**Description**

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

**Example**

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

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

<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 Structure</td>
<td>6-4</td>
</tr>
<tr>
<td>6.3 Functions</td>
<td>6-7</td>
</tr>
<tr>
<td>6.4 Macros</td>
<td>6-20</td>
</tr>
<tr>
<td>6.5 Configuring the DMA Module Using CSL GUI</td>
<td>6-32</td>
</tr>
<tr>
<td>6.6 Examples</td>
<td>6-40</td>
</tr>
</tbody>
</table>
6.1 Overview

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

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

- Individual channel register reload support
- Extended Data Memory Support

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

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

Table 6–1. DMA Configuration Structure

<table>
<thead>
<tr>
<th>Structure</th>
<th>Purpose</th>
<th>See page ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA_Config</td>
<td>DMA structure that contains all local registers required to set up a specific DMA channel.</td>
<td>6-4</td>
</tr>
<tr>
<td>DMA_GblConfig</td>
<td>Global DMA structure that contains all global registers that you may need to initialize a DMA channel</td>
<td>6-5</td>
</tr>
</tbody>
</table>

Table 6–2. DMA Functions

(a) DMA Primary Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
<th>See page ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA_close()</td>
<td>Closes a DMA channel</td>
<td>6-7</td>
</tr>
<tr>
<td>DMA_config()</td>
<td>Sets up the DMA channel using the configuration structure</td>
<td>6-7</td>
</tr>
<tr>
<td>DMA_configArgs()</td>
<td>Sets up the DMA channel using the register values passed in</td>
<td>6-8</td>
</tr>
<tr>
<td>DMA_open()</td>
<td>Opens a DMA channel</td>
<td>6-10</td>
</tr>
<tr>
<td>DMA_pause()</td>
<td>Pauses a DMA channel. Identical to DMA_stop().</td>
<td>6-11</td>
</tr>
<tr>
<td>DMA_reset()</td>
<td>Resets DMA channel register to their power-on reset value</td>
<td>6-11</td>
</tr>
<tr>
<td>DMA_start()</td>
<td>Starts a DMA channel</td>
<td>6-11</td>
</tr>
<tr>
<td>DMA_stop()</td>
<td>Disables a DMA channel</td>
<td>6-12</td>
</tr>
</tbody>
</table>
Table 6–2. DMA Functions (Continued)

(b) DMA Global Register Function

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

(c) DMA Auxiliary Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
<th>See page ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA_autostart()</td>
<td>Enables a DMA channel</td>
<td>6-18</td>
</tr>
<tr>
<td>DMA_getConfig()</td>
<td>Get DMA channel configuration</td>
<td>6-18</td>
</tr>
<tr>
<td>DMA_getEventId()</td>
<td>Returns the IRQ Event ID for the DMA completion interrupt</td>
<td>6-19</td>
</tr>
</tbody>
</table>
6.2 Configuration Structure

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

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

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

**DMA_Config**  
**DMA channel configuration structure**

<table>
<thead>
<tr>
<th>Structure</th>
<th>DMA_Config</th>
</tr>
</thead>
<tbody>
<tr>
<td>Members</td>
<td>Uint16 priority</td>
</tr>
</tbody>
</table>

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

| Uint16 autoix | Provided for compatibility with older versions of DMA |

For all devices add:

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

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

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

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

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

**Note:**

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

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

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

DMA_GblConfig

**DMA global configuration structure**

Structure: DMA_GblConfig

Members
- Uint16 free: run free under emulation control
- For devices supporting individual channel reload registers (see note) add:
  - Uint16 autoix: For compatibility with older versions of DMA
- For all devices add:
  - Uint16 gbldmsrcp: global program page for src
  - Uint16 gblmdstsp: global program page for dst
  - Uint16 gbldmdx0: global element index 0
  - Uint16 gbldmfri0: global frame index 0
  - Uint16 gbldmdx1: global element index 1
  - Uint16 gbldmfri1: global frame index 1
- For devices not offering global channel reload registers (see note), add:
  - DMA_AdrPtr gbldmsa: global src address reload
  - DMA_AdrPtr gbldmda: global dst address reload
  - Uint16 gbldmcr: global element count reload
  - Uint16 gbldmfr: global frame count reload
- For devices supporting global extended data memory addressing (see note), add:
  - Uint16 gbldmsrcdp: global data page for src
  - Uint16 gblmdstdp: global data page for dst

DMA Module 6-5
Note:
For more information concerning these devices, see section 1.8, Device-Specific Features Support.

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

Example 1

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

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

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

Example 2

```c
extern DMA_AdrPtr mySrc;
extern myDst;
DMA_gblConfig myGblConfig = {
    0,
    0,
    0,
    0x1,
    0x4
    0,
    0
};
```

```c
myGblConfig.gbldmsrcp = (((Uint32)(&mySrc)>>16)&0xFFFFu);
myGblConfig.gblmdstp = (((Uint32)(&myDst)>>16)&0xFFFFu);
```
6.3 Functions

This section describes the functions in the DMA CSL module.

6.3.1 DMA Primary Functions

**DMA_close**  *Closes DMA channel*

Function

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

Arguments

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

Return Value

None

Description

Closes a DMA channel previously opened with DMA_open(). The registers for the DMA channel are set to their power-on reset defaults, then the completion interrupt is disabled and cleared.

Example

```c
DMA_close(hDma);
```

**DMA_config**  *Sets up DMA channel using configuration structure*

Function

```c
void DMA_config(
    DMA_handle hDma,
    DMA_Config *Config
);
```

Arguments

- **hDma** Handle to DMA channel; see DMA_open().
- **Config** Pointer to an initialized configuration structure (See DMA_Config)

Return Value

None

Description

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

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

`DMA_config(hDma,&MyConfig);`

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

**DMA_configArgs**

*Sets up DMA channel with register values*

**Function**

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

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

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

**Note:**

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

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDma</td>
<td>Handle to DMA channel; see DMA_open()</td>
</tr>
<tr>
<td>priority</td>
<td>DMA channel priority</td>
</tr>
</tbody>
</table>

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

- Uint16 autoix (Provided for compatibility with older versions of DMA)
- dmmcr DMA transfer mode control register value
- dmsfc DMA sync select and frame count register value
- dmsrc DMA source address register value
- dmdst DMA destination address register value
- dmctr DMA element count register value

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

- dmsrcdpa Pointer to DMA source address reload value
- dmdstdpa Pointer to DMA destination address reload value
- dmsgpa Pointer to DMA element count reload value
- dmsgfra Pointer to DMA frame count reload value

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

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

Return Value

None

Description

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

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

Example

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

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

Opens DMA channel

**Function**

DMA_Handle DMA_open(
    int ChaNum,
    Uint32 Flags
);

**Arguments**

ChaNum DMA channel to open:
   - DMA_CHA_ANY
   - DMA_CHA0
   - DMA_CHA1
   - DMA_CHA2
   - DMA_CHA3
   - DMA_CHA4
   - DMA_CHA5

Flags Open flags (logical OR of any of the following):
   - DMA_OPEN_RESET

**Return Value**

Device handle Handle to newly opened device

**Description**

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

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

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

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

**Example**

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

*Pauses DMA channel*

**Function**

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

**Arguments**

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

**Return Value**

None

**Description**

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

**Example**

```c
DMA_pause(hDma);
```

**DMA_reset**

*Resets DMA channel*

**Function**

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

**Arguments**

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

**Return Value**

None

**Description**

Resets the DMA channel by setting its registers to the power-on defaults and disables and clears the channel interrupt. You can use INV as the device handle to reset all channels.

**Example**

```c
/* reset an open DMA channel */
DMA_reset(hDma);

/* reset all DMA channels */
DMA_reset(INV);
```

**DMA_start**

*Starts DMA channel*

**Function**

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

**Arguments**

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

**Return Value**

None

**Description**

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

**Example**

```c
DMA_start(hDma);
```
Disables DMA channel

Function
void DMA_stop(
    DMA_Handle hDma
);

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

Return Value
None

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

Example
DMA_stop(hDma);

6.3.2 DMA Global Register Function

Performs global register allocation

Function
Uint16 DMA_globalAlloc (  
    Uint16 RegMask
);

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

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

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

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

Description

Performs Global register allocation. This function returns a mask that indicates to the DMA_global Config/ConfigArgs functions which global registers are being allocated for the DMA channel. If you request via RegMask a global register that has been previously allocated the function returns a zero. The use of this function is considered optional. It can be used to prevent double allocation of registers to DMA channels. If not used, you can pass off the DMA_GBL_ALL (0xffff value) as the RegMaskAlloc parameter for the DMA_global Config/Args functions.

Example

```c
#define NOTUSED 0

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

```
mask = DMA_globalAlloc (DMA_GBL_DMX1|DMA_GBL_DMFRI1);
DMA_globalConfig (mask, &MyGblConfig);
```

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

Sets up DMA global registers using configuration structure

Function

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

Arguments

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

Return Value

None

Description

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

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

Example

```c
#define NOTUSED 0

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

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

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

Sets up DMA global registers using arguments

**Function**

```c
void DMA_globalConfigArgs(
    Uint16 RegMask,
    Uint16 free,
    
    For devices supporting individual channel reload registers (see note) add:
    Uint16 autoix, (Provided for compatibility with older versions of DMA)

    For all devices add:
    Uint16 intosel,
    Uint16 dmidx0,
    Uint16 dfri0,
    Uint16 dmidx1,
    Uint16 dfri1,

    For devices not supporting global channel reload registers, (see section 1.8) add:
    Uint16 dmgsa,
    Uint16 dmgda,
    Uint16 dmc,
    Uint16 dmgcr,
    Uint16 dfri,

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

    For devices supporting extended DMA data support, (see section 1.8) add:
    Uint16 dmsrcdp,
    Uint16 dmdstdp
```

**Arguments**

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

- **free**
  Response to emulation control

- **dmidx0**
  Global element index 0

- **dfri0**
  Global frame index 0

- **dmidx1**
  Global element index 1

- **dfri1**
  Global frame index 1
### DMA_globalFree

For devices supporting global channel reload registers, (see section 1.8):
- **dmgsa;** Pointer to global src address reload
- **dmgda;** Pointer to global dst address reload
- **dmgcr;** Global element count reload
- **dmgfr;** Global frame count reload

For all devices:
- **dmsrcp;** Global program page for src
- **dmdstp;** Global program page for dst

For devices supporting extended data addressing (see section 1.8):
- **dmsrcep;** Global data page for src
- **dmdstdp;** Global data page for dst

**Return Value**
None

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

**Example**
None

---

**Frees global DMA register that was previously allocated**

### Function
void DMA_globalFree(
    Uint16 regMask
);  

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

### Return Value
None

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

### Example
Uint16 RegMask;
...
    RegMask = DMA_globalAlloc(DMA_GBL_DMIDX0, DMA_GBL_DMIDX0);
...
    /* some time later on when you’re done with it */
    DMA_globalFree(RegMask);

---

6-16
DMA_resetGbl

**DMA_globalGetConfig**  Gets a DMA global configuration register

| Function | void DMA_globalGetConfig (  
|          |   Uint16 RegMaskAlloc,  
|          |   DMA_GblConfig *Config  
|          | );                     |
| Arguments| RegMaskAlloc Mask that indicates which global register to get. Refer to DMA_globalAlloc for valid values. DMA_GBL_ALL will get all global registers  
|          | Config Pointer to an un-initialized global configuration structure  |
| Return Value | None |
| Description | Specifies the current configuration for the DMA global registers specified by RegMask. This is accomplished by reading the actual DMA global registers and fields and storing them back in the config structure. |
| Example | DMA_GblConfig ConfigRead;  
|          | ...  
|          | DMA_globalGetConfig (DMA_GBL_ALL, &ConfigRead); |

DMA_resetGbl  Resets a DMA global register

| Function | void DMA_resetGbl(  
|          |   DMA_Handle hDma  
|          | );                        |
| Arguments | hDma Handle to DMA channel; see DMA_open(),  
|          | Or INV (-1) if you want to reset all DMA channel registers. |
| Return Value | None |
| Description | Resets the DMA global register by setting all global registers to the power-on defaults. You must use INV (-1) as the device handle to reset all the global registers. |
| Example | DMA_resetGbl(hDma);  
|          | /* or */  
|          | DMA_resetGbl(INV); |
6.3.3 DMA Auxiliary Functions

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

**Function**
```c
void DMA_autostart(
    DMA_Handle hDma
);
```

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

**Return Value**
None

**Description**
Enables the specified DMA channel and sets the AUTOINIT bit.

**Example**
```c
DMA_autostart(hDma);
```

**DMA_getConfig**  
*Gets a DMA channel configuration*

**Function**
```c
void DMA_getConfig(
    DMA_Handle hDma,
    DMA_Config *Config
);
```

**Arguments**
- `hDma`  Handle to DMA channel; see DMA_open().
- `Config`  Pointer to an un-initialized configuration structure (see DMA_Config)

**Return Value**
None

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

**Example**
```c
DMA_Config ConfigRead;

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

- **Function**: Uint16 DMA_getEventId(
  - DMA_Handle hDma
);  
- **Arguments**: hDma Handle to DMA channel; see DMA_open().  
- **Return Value**: Event ID IRQ Event ID for DMA Channel  
- **Description**: Returns the IRQ Event ID for the DMA completion interrupt. Use this ID to manage the event using the IRQ module.  
- **Example**

```c
EventId = DMA_getEventId(hDma);
IRQ_enable(EventId);
```

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

### DMA_getStatus

- **Function**: Uint16 DMA_getStatus (  
  - DMA_Handle hDma
);  
- **Arguments**: hDma  
- **Return Value**:  
  - 1: if DMA channel is still running  
  - 0: if DMA channel has stopped (transfer completed)  
- **Description**: Returns the status of the DMA channel used by handle. Use as an indication of transfer complete.  
- **Example**

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

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

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

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

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

(a) Macros to read/write DMA register values

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA_RGET()</td>
</tr>
<tr>
<td>DMA_RSET()</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA_FGET()</td>
</tr>
<tr>
<td>DMA_FSET()</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA_REG_RMK()</td>
</tr>
<tr>
<td>DMA_FMK()</td>
</tr>
</tbody>
</table>

(d) Macros to read a register address

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA_ADDR()</td>
</tr>
</tbody>
</table>

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

(a) Macros to read/write DMA register values

<table>
<thead>
<tr>
<th>Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA_RGETH()</td>
</tr>
<tr>
<td>DMA_RSETH()</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA_FGETH()</td>
</tr>
<tr>
<td>DMA_FSETH()</td>
</tr>
</tbody>
</table>

(c) Macros to read a register address

<table>
<thead>
<tr>
<th>Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA_ADDRH()</td>
</tr>
</tbody>
</table>
**DMA_RGET**

Get value of 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,
    - DMSRC#
    - DMDST#
    - DMCTR#
    - DMSFC#
    - DMMCR#
  - For devices supporting individual channel reload registers, add:
    - DMGSA#
    - DMGDA#
    - DMGCR#
    - DMGFR#
  - For devices supporting individual channel extended data memory space support, add:
    - DMSRCDP#
    - DMDSTDP#

- GLOBALREG  Global register name
  - DMPREC
  - DMSRCP
  - DMDSTP
  - For devices not supporting individual channel extended data memory space support, add:
    - DMSRCDP
    - DMDSTDP
  - For devices supporting global channel reload registers, add:
    - DMGSA
    - DMGDA
    - DMGCR
    - DMGFR
  - For devices supporting global extended data memory space support, add:
    - DMSRCDP#
    - DMDSTDP#
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example 1</th>
<th>Example 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA_RSET</td>
<td>Returns the DMA register value</td>
<td>For local registers:</td>
<td>For global registers:</td>
</tr>
<tr>
<td>Return Value</td>
<td>value of register</td>
<td>Uint16 myvar;</td>
<td>Uint16 myvar;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>myVar = DMA_RGET(DMSRC1); /* read DMSRC for channel 1 */</td>
<td>... myVar = DMA_RGET(DMPREC);</td>
</tr>
<tr>
<td>Example 1</td>
<td>For local registers:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**DMA_RSET**

*Set value of DMA register*

**Macro**

Void DMA_RSET (REG, Uint16 regval)

**Arguments**

REG   LOCALREG# or GLOBALREG, as listed in DMA_RGET()  
macro  
regval register value that wants to write to register REG

**Return Value**

value of register

**Description**

Set the DMA register REG value to regval

**Example 1**

For local registers:

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

**Example 2**

For global registers:

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

Creates register value based on individual field values

**Macro**

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

**Arguments**

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

For example:

DMSFC

DMMCR

DMPREC

fieldval Field values to be assigned to the writable register fields.

Rules to follow:

- Only writable fields are allowed
- Start from Most-significant field first
- Value should be a right-justified 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;
myregval = DMA_DMSFC_RMK (0,0,3); /* dsyn,dblw,framecount fields */
```

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

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

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

*Create register value based on individual field values*

**Macro**

Uint16 DMA_FMKE (REG, FIELD, fieldval)

**Arguments**

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

DMPREC
DMSFC
DMMCR

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

fieldval Field values to be assigned to the writable register fields.

Rules to follow:

- Only writable fields are allowed
- Start from Most-significant field first
- Value should be a right-justified 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_FMKE (DMSFC, DBLN, 1) | DMA_FMKE (DMSFC, DSYN, 2);
### DMA_FGET

**Get value of register field**

#### Macro

`Uint16 DMA_FGET (REG, FIELD)`

#### Arguments

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

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

#### Return Value

Value of register field

#### Description

Gets the DMA register field value

#### Example 1

For local registers:

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

#### Example 2

For global registers:

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

*Set value of register field*

**Macro**

Void DMA_FSET (REG, FIELD, fieldval)

**Arguments**

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

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

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

**Return Value**

None

**Description**

Set the DMA register value to regval

**Example 1**

For Local Registers:

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

**Example 2**

For global registers:

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

Get address of given register

**Macro**

Uint16 DMA_ADDR (REG)

**Arguments**

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

**Return Value**

Address of register LOCALREG and GLOBALREG

**Description**

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

**Example 1**

For local registers:

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

**Example 2**

For global registers:

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

**DMA_RGETH**

Get value of DMA register used in handle

**Macro**

Uint16 DMA_RGETH (DMA_Handle hDma, LOCALREG)

**Arguments**

hDma Handle to DMA channel that identifies the specific DMA channel used.

LOCALREG Same register as in DMA_RGET(), but without channel number (#). Example: DMSRC (instead of DMSRC#)

**Return Value**

Value of register

**Description**

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

**Example**

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

**Set value of DMA register**

**Macro**

void DMA_RSETH (DMA_Handle hDma, LOCALREG, Uint16 regval)

**Arguments**

- **hDma**
  - Handle to DMA channel that identifies the specific DMA channel used.

- **LOCALREG**
  - Same register as in DMA_RSET(), but without channel number (#). Example: DMSRC (instead of DMSRC#)

- **regval**
  - Value to write to register LOCALREG for the channel associated with handle.

**Return Value**

None

**Description**

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

**Example**

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

### DMA_FGETH

**Get value of register field**

**Macro**

Uint16 DMA_FGETH (DMA_Handle hDma, LOCALREG, FIELD)

**Arguments**

- **hDma**
  - Handle to DMA channel that identifies the specific DMA channel used.

- **LOCALREG**
  - Same register as in DMA_RSET(), but without channel number (#). Example: DMSRC (instead of DMSRC#)

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

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

**Return Value**

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

**Description**

Gets the DMA register field value.

**Example**

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

**Set value of register field**

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

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

**Return Value**
None

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

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

`Uint16 DMA_ADDR_H (DMA_Handle hDma, LOCALREG,)`

**Arguments**

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

**Return Value**

Address of register LOCALREG

**Description**

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

**Example**

```c
DMA_Handle myHandle;
Uint16  myVar;
...
...myVar = DMA_ADDRH (myHandle, DMMCR);
```
6.5 Configuring the DMA Module Using CSL GUI

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

The configuration manager allows creation of an object that contains the complete set of register values needed to configure a DMA channel. The resource manager associates a configuration object with a specific DMA channel.

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

Figure 6–1. DMA Sections Menu

The DMA includes the following sections:

- **DMA Configuration Manager** allows you to create configuration objects by setting the peripheral registers related to the DMA.

- **DMA Resource Manager** allows you to select a DMA channel and to associate a configuration object to this channel. The six channel handle objects are predefined.

6.5.1 DMA Configuration Manager

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

6.5.1.1 Creating/Inserting a Configuration

There is no predefined configuration object available.

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

**Step 2:** To insert a new configuration object, right-click on the DMA Configuration Manager and select insert dmaCfg from the drop-down menu. The configuration objects can be renamed. Their use depends on the on-chip device resources. Because six channels are available, a maximum of six configurations can be used simultaneously.
Note:
A maximum of six configurations may be inserted. This is due to the association that each configuration has with a pre-defined global configuration. The global configuration is dynamically updated with changes made to the associated DMA configuration. One DMA configuration (and its associated global configuration) can be used by more than one DMA channel.

6.5.1.2 Deleting/Renaming an Object

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

If a configuration object is used by one of the predefined handle objects of the DMA Resource Manager, the Delete and Rename options are grayed-out and non usable. The Show Dependency option is accessible and shows which device is using the configuration object (see Figure 2–1, The CSL Tree, on page 2-3).

6.5.1.3 Configuring the Object Properties

You can configure object properties through the Properties dialog box. (See Figure 6–2). To access the Properties dialog box, right-click on a configuration object and select Properties. By default, the General page of the Properties dialog box is displayed.

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

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

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

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

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

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

Figure 6–3. DMA Resource Manager Menu

6.5.2.1 Predefined Objects

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

- **DMA0** – Default handle name: hDma0
- **DMA1** – Default handle name: hDma1
- **DMA2** – Default handle name: hDma2
- **DMA3** – Default handle name: hDma3
- **DMA4** – Default handle name: hDma4
- **DMA5** – Default handle name: hDma5

6.5.2.2 Properties Page

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

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

The first time the Properties page appears, only the Open Handle to DMA check-box can be selected. Select this to open the DMA channel, allowing pre-initialization.
DMA NOTHING is used to indicate that there is no configuration object selected for this DMA.

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

If DMA NOTHING is selected, no configuration object is generated for the related DMA handle (see section 6.5.3, C Code Generation for DMA Module).

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

Figure 6–4. DMA Properties Page With Handle Object Accessible
6.5.3 C Code Generation for DMA Module

Two C files are generated from the configuration tool:

- Header file
- Source file

6.5.3.1 Header File

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

Example 6–1. DMA Header File

```c
extern DMA_Config dmaCfg0;
extern DMA_GblConfig gDMAConfig0;
extern DMA_Handle hDma1;
```
6.5.3.2 Source File

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

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

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

DMA_GblConfig gDMAConfig0 = {
    0x0, /* Breakpoint Emulation Behavior (FREE) */
    0x0000, /* Global Reload Register Usage in Autoinit Mode (AUTOIX : 0x0000 or 0x0001) */
    NULL, /* Source Program Page Address Register (DMSRCP) - Symbolic */
    NULL, /* Destination Program Page Address Register (DMDSTP) - Symbolic */
    0x0000 /* Element Address Index Register 0 (DMIDX0) */
    0x0000 /* Frame Address Index Register 0 (DMFR0) */
    0x0000 /* Element Address Index Register 1 (DMIDX1) */
    0x0000 /* Frame Address Index Register 1 (DMFR1) */
    NULL, /* Global Source Address Reload Register (DMGSA) - Symbolic */
    NULL, /* Global Destination Address Reload Register (DMGDA) - Symbolic */
    0x0000 /* Global Element Count Reload Register (DMGCR) */
    0x0000 /* Global Frame Count Reload Register B (DMGFR) */
};

/* Handles */
DMA_Handle hDma1;
```

The source file contains the Handle and Configuration Pre-Initialization using the CSL DMA API functions, DMA_open() and DMA_config() (see Example 6–3).
These two functions are encapsulated in a unique function, CSL_cfgInit(), which is called from your main C file. DMA_open() and DMA_config() are generated only if Open Handle to DMA and Enable pre-initialization (with a selected configuration other than DMA_NOTHING) are checked under the DMA Resource Manager Properties page.

**Example 6-3. DMA Source File (Body Section)**

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

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

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

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

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

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

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

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

\[
\text{DMA_DMMCR_RMK}(\text{autoinit, dinm, imod, ctmod, sind, dms, dind, dmd})
\]

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
\end{array}
\]

\[
\text{DMA_DMSFC_RMK}(\text{dsyn, dblw, framecount})
\]

\[
\begin{array}{cccc}
0 & 0 & 0 & (\text{single-frame, Nframes-1}) \\
\end{array}
\]

The settings needed for the DMMCR are:

\[
\text{DMMCR0} = 0x0145u
\]

\[
#0000000100000101b
\]
Examples

;0~---------------- (AUTOINIT)  Autoinitialization disabled
;~0~------------ (DINM)  Interrupts masked
;~~0~---------- (IMOD)  N/A
;~~~0~---------- (CTMOD)  Multi-frame mode
;~~~~0~~~~~~~~~ (Reserved)
;~~~~~001~~~~~ (SIND)  Post increment source address
;~~~~~~~~~~0~~ (DMS)  Source in data space
;~~~~~~~~~~~~0~~~~ Reserved
;~~~~~~~~~~~001~~ (DIND)  Post increment destination address
;~~~~~~~~~~~~~~01 (DMD)  Destination in data space

The settings needed for the DMSFC are:

DMSFC0 = 0x0000u

#include <csl_dma.h>

/**
 * @brief DMA Channel Initialization Using DMA_config()
 */

#define N 16

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

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

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

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

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

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

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

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

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

/* DMSFC0 = 0x0000u */
/* #0000000000000000b */
/* ;0000~~~~~~~~~ (DSYN) No sync event */
/* ;~~~~0~~~~~~~~~ (DBLW) Single-word mode */
/* ;~~~~~000~~~~~~ N/A */
/* ;~~~~~~~~~000000 (Frame Count) = 0 (one frame) */

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

6-42
Figure 6-5. DMA Channel Initialization Using DMA_config() (Continued)

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

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

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

void taskFunc() {

DMA Module
Examples

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

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

LOG_printf(&LogMain,"<DMA1A>”);

/* Open DMA channel 0, to use for this transfer */
myhDma = DMA_open(DMA_CHA0, 0);

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

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

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

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

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

LOG_printf(&LogMain,”%s”,err?”DMA Example 1A FAILED”:”DMA Example 1A PASSED”);
LOG_printf(&LogMain,”<DONE>”);
This chapter describes the configuration structure, functions, and macros used in the external bus interface (EBUS) module.

<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 Structure</td>
<td>7-3</td>
</tr>
<tr>
<td>7.3 Functions</td>
<td>7-4</td>
</tr>
<tr>
<td>7.4 Macros</td>
<td>7-6</td>
</tr>
</tbody>
</table>
7.1 Overview

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

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

Use the following guidelines for the EBUS functions:

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

Table 7–1. EBUS Configuration Structure

<table>
<thead>
<tr>
<th>Structure</th>
<th>Purpose</th>
<th>See page...</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBUS_Config</td>
<td>EBUS configuration structure used to setup the EBUS interface</td>
<td>7-3</td>
</tr>
</tbody>
</table>

Table 7–2. EBUS Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
<th>See page...</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBUS_config()</td>
<td>Sets up EBUS using configuration structure (EBUS_Config)</td>
<td>7-4</td>
</tr>
<tr>
<td>EBUS_configArgs()</td>
<td>Sets up EBUS using register values passed to the function</td>
<td>7-5</td>
</tr>
</tbody>
</table>
7.2 Configuration Structure

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

**EBUS_Config**

*EBUS configuration structure used to set up EBUS interface*

**Structure**

EBUS_Config

**Members**

For 544x devices:
Uint16 bscr  Bank-switching control register

For other C54x devices:
Uint16 swssr  Software wait-state register
Uint16 bscr  Bank-switching control register
Uint16 swcr  Software wait-state control register

**Description**

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

**Example**

```c
EBUS_Config Config1 = {
  0x7FFF, /* swssr */
  0xF800, /* bscr */
  0x0000 /* swcr */
};
```
This section describes the EBUS API functions.

### EBUS_config

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

<table>
<thead>
<tr>
<th>Function</th>
<th>void EBUS_config(EBUS_Config *Config)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arguments</strong></td>
<td>Config Pointer to an initialized configuration structure</td>
</tr>
<tr>
<td><strong>Return Value</strong></td>
<td>None</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Writes a value to set up the EBUS using the configuration structure. The values of the structure are written to the port registers (see also EBUS_configArgs() and EBUS_Config).</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td>EBUS_Config MyConfig = {\n      0x7FFF, /* swsr <em>/\n      0xF800, /</em> bscr <em>/\n      0x0000 /</em> swcr */\n    };\n    ...\n    EBUS_config(&amp;MyConfig);</td>
</tr>
</tbody>
</table>
EBUS_configArgs

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

Function

For C54x devices:
EBUS_configArgs (Uint16 bscr);

For other C54x devices:
void EBUS_configArgs(
    Uint16 swsrs,
    Uint16 bscr,
    Uint16 swcr
);

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>swsrs</td>
<td>Software wait-state register</td>
</tr>
<tr>
<td>bscr</td>
<td>Bank-switching control register</td>
</tr>
<tr>
<td>swcr</td>
<td>Software wait-state control register</td>
</tr>
</tbody>
</table>

Return Value

None

Description

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

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

Example

EBUS_configArgs {
    0x7FFF, /* swsrs */
    0xF800, /* bscr */
    0x0000 /* swcr */
};
7.4 Macros

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

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

Table 7–3. EBUS Macros

(a) Macros to read/write EBUS register values

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

(b) Macros to read/write EBUS 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>EBUS_FGET()</td>
<td>Uint16 EBUS_FGET(REG, FIELD)</td>
</tr>
<tr>
<td>EBUS_FSET()</td>
<td>Void EBUS_FSET(REG, FIELD, Uint16 fieldval)</td>
</tr>
</tbody>
</table>

(c) Macros to read/write EBUS 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>EBUS_REG_RMK()</td>
<td>Uint16 EBUS_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 field_0: LSB field * only writeable fields allowed</td>
</tr>
<tr>
<td>EBUS_FMK()</td>
<td>Uint16 EBUS_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>EBUS_ADDR()</td>
<td>Uint16 EBUS_ADDR(REG)</td>
</tr>
</tbody>
</table>
Macros

Where:

REG: SWWSR (except in C544x), SWCR (except in C544x), BSCR
FIELD: register field name as specified in Appendix A.
- For _FSET and _FMK, field should be a writable field.
- _FGET, this field should, at least, be readable.
regVal: value to write in register REG
fieldVal: value to write in field FIELD of register REG. Rules to follow:
- Only writable fields are allowed.
- Value should be a right-justified constant. If fieldval_n value exceeds the number of bits allowed for that field, fieldval_n is truncated accordingly.

For examples on how to use macros, refer to macro sections 6.4 (DMA) and 11.4 (McBSP).
The GPIO module is designed to allow central control of non-multiplexed GPIO pins available in the C54x devices. (C544x devices only)

<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 Functions</td>
<td>8-3</td>
</tr>
<tr>
<td>8.3 Macros</td>
<td>8-5</td>
</tr>
<tr>
<td>8.4 Configuring the GPIO Module Using CSL GUI</td>
<td>8-7</td>
</tr>
</tbody>
</table>
8.1 Overview

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

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

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

Table 8–1. GPIO Functions

<table>
<thead>
<tr>
<th>Function</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>8-3</td>
</tr>
<tr>
<td>GPIO_pinRead</td>
<td>Reads the GPIO pin value</td>
<td>8-3</td>
</tr>
<tr>
<td>GPIO_pinWrite</td>
<td>Writes a value to a GPIO pin</td>
<td>8-4</td>
</tr>
</tbody>
</table>
8.2 Functions

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

**GPIO_pinDirection** *Sets the GPIO pin as input or output*

*Function*

```c
int GPIO_pinDirection (Uint32 pinId, Uint16 direction);
```

*Arguments*

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

- **Direction**: pin direction
  - GPIO_INPUT //input
  - GPIO_OUTPUT //output

*Return Value*

None

*Description*

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

*Example*

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

**GPIO_pinRead** *Reads the GPIO pin value*

*Function*

```c
int GPIO_pinRead (Uint32 pinId);
```

*Arguments*

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

*Return Value*

Value Value read in GPIO pin (1 or 0)

*Description*

Reads the value in a general purpose input pin.

*Example*

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

**Function**

```
int GPIO_pinWrite (Uint32 pinId, Uint16 val);
```

**Arguments**

- **pinId**: IDs of the GPIO pins to enable. It will have one of the following values:
  - GPIO_PIN0
  - ...
  - GPIO_PIN3
  
  Value to write to the pinID.

**Return Value**

None

**Description**

Writes a value to a general purpose output pin.

**Example**

```
GPIO_pinWrite (GPIO_PIN1, 1); /* sets iopin1 to "1" */
```
8.3 Macros

As covered in section 1.3, CSL offers a collection of macros to gain individual access to a GPIO specific register (GPIO) in C544x devices. Table 8–2 contains a list of macros available for the GPIO module. To use them, include “csl_gpio.h.”

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

(a) Macros to read/write GPIO register values

<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 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_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: field_n: MSB field field_0: LSB field * 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>
Where:

**REG**: GPIO

**FIELD**: register field name as specified in Appendix A.
- For _FSET and _FMK, field should be a writable field.
- _FGET, this field should, at least, be readable.

**regVal**: value to write in register REG

**fieldVal**: value to write in field FIELD of register REG. Rules to follow:
- Only writable fields are allowed.
- Value should be a right-justified constant. If fieldval_n value exceeds the number of bits allowed for that field, fieldval_n is truncated accordingly.

For examples on how to use macros, refer to the macro sections 6.4 (DMA) and 11.4 (McBSP).
8.4 Configuring the GPIO Module Using CSL GUI

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

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

Figure 8–1. GPIO Sections Menu

The Non-Multiplexed GPIO includes the following section:

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

8.4.1 Non-Multiplexed GPIO Configuration Manager

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

8.4.1.1 Properties Pages of the Non-Multiplexed GPIO Configuration

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

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

Figure 8–2 depicts the Properties Page dialog box.

Figure 8–2. GPIO Properties Page
8.4.2 C Code Generation for GPIO Module

Two C files are generated from the configuration tool:

- Header file
- Source file

8.4.2.1 Header File

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

8.4.2.2 Source File

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

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

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

```c
void CSL_cfgInit()
{
    GPIO_RSET(GPIO, 3840);
}
```
This chapter contains descriptions for macros available in the HPI module.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1 Macros</td>
<td>9-2</td>
</tr>
</tbody>
</table>
As covered in section 1.3, the CSL offers a collection of macros to gain individual access to the peripheral registers and fields.

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

### Table 9–1. HPI Macros

(a) Macros to read/write HPI register values

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

(b) Macros to read/write HPI 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>HPI_FGET()</td>
<td>Uint16 HPI_FGET(REG, FIELD)</td>
</tr>
<tr>
<td>HPI_FSET()</td>
<td>Void HPI_FSET(REG, FIELD, Uint16 fieldval)</td>
</tr>
</tbody>
</table>

(c) Macros to read/write HPI 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>HPI_REG_RMK()</td>
<td>Uint16 HPI_REG_RMK(fieldval_n,...fieldval_0)</td>
</tr>
<tr>
<td></td>
<td>Note:</td>
</tr>
<tr>
<td></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>HPI_FMK()</td>
<td>Uint16 HPI_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>HPI_ADDR()</td>
<td>Uint16 HPI_ADDR(REG)</td>
</tr>
</tbody>
</table>
Where:

**REG:** include HPIC, GPIOCR, GPIOSR

**FIELD:** register field name as specified in Appendix A.
- For _FSET and _FMK, field should be a writable field.
- For _FGET, this field should, at least, be readable.

**regVal:** value to write in register REG

**fieldVal:** value to write in field FIELD of register REG. Rules to follow:
- Only writable fields are allowed
- Value should be a right-justified constant. If fieldval_n value exceeds the number of bits allowed for that field, fieldval_n is truncated accordingly.

For examples on how to use macros, refer to macro sections 6.4 (DMA) and 11.4 (McBSP).
The IRQ module provides an easy to use interface for enabling/disabling and managing interrupts.
10.1 Overview

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

- Masking an interrupt in the 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 10–1 lists the IRQ configuration structure.

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

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

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

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

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

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

Table 10–3 lists all IRQ logical interrupt events for this module.
### Table 10–1. IRQ Configuration Structure

<table>
<thead>
<tr>
<th>Structure</th>
<th>Purpose</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>10-8</td>
</tr>
</tbody>
</table>

### Table 10–2. IRQ Functions

#### (a) Primary Functions

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

#### (b) Auxiliary Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
<th>See page ...</th>
</tr>
</thead>
<tbody>
<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>10-14</td>
</tr>
</tbody>
</table>

**Caution:** Do not use this function when DSP/BIOS is present and the dispatcher is enabled.
Table 10–2. IRQ Functions (Continued)

(c) DSP/BIOS Dispatcher Communication Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
<th>See page</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRQ_config()</td>
<td>Updates the DSP/BIOS dispatch table with a new configuration for the specified event.</td>
<td>10-9</td>
</tr>
<tr>
<td>IRQ_configArgs()</td>
<td>Updates the DSP/BIOS dispatch table with a new configuration for the specified event.</td>
<td>10-10</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>10-11</td>
</tr>
<tr>
<td>IRQ_getConfig()</td>
<td>Returns current DSP/BIOS dispatch table entries for the specified event.</td>
<td>10-12</td>
</tr>
<tr>
<td>IRQ_map()</td>
<td>Maps a logical event to its physical interrupt.</td>
<td>10-14</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>10-15</td>
</tr>
</tbody>
</table>

10.1.1 The Event ID Concept

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

Table 10–3. IRQ_EVT_NNNN Event List

<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>
</tbody>
</table>
### Table 10–3. IRQ_EVT_NNNN Event List (Continued)

<table>
<thead>
<tr>
<th>Constant</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<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>
<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>
</tbody>
</table>


Table 10–3. IRQ_EVT_NNNN Event List (Continued)

<table>
<thead>
<tr>
<th>Constant</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<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_HPINT</td>
<td>Host Interrupt (HPI)</td>
</tr>
<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_WDTINT</td>
<td>Watchdog Timer Interrupt</td>
</tr>
<tr>
<td>IRQ_EVT_UART</td>
<td>UART Interrupt</td>
</tr>
<tr>
<td>IRQ_EVT_DAAARC</td>
<td>DAA Receive Interrupt</td>
</tr>
<tr>
<td>IRQ_EVT_DAAAXMT</td>
<td>DAA Transmit Interrupt</td>
</tr>
</tbody>
</table>
10.2 Using Interrupts with CSL

Interrupts within CSL can be managed in the following methods:

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

Example 10–1. Manual Setting Outside DSPBIOS HWIs

```c
#define NVECTORS 256
#pragma DATA_SECTION (myIvtTable, "myvec")
int myIvtTable[NVECTORS];
...
interrupt void myIsr();
...
main (){ ...
; ...
; Option 1: use Event IDs directly ...
; ...
IRQ_setVecs ((Uint16)myIvtTable);
IRQ_plug (IRQ_EVT_TINT0,&myIsr);
IRQ_enable(IRQ_EVT_TINT0);
IRQ_globalEnable();
; ...
; Option 2: Use the PER_getEventId() function (TIMER as an example) for a better abstraction ...
; ...
IRQ_setVecs ((Uint16)myIvtTable);
eventId = TIMER_getEventId (hTimer);
IRQ_plug (eventId,&myIsr);
IRQ_enable (eventId);
IRQ_globalEnable();
; ...
}
interrupt void myISR()
{
//....;
}
```
10.3 Configuration Structure

**IRQ_Config**

**IRQ configuration structure**

**Structure**

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

**Description**

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

**Example**

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

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

**IRQ_clear**  
*Clears event flag from IFR register*

**Function**

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

**Arguments**

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

**Return Value**: None

**Description**: Clears the event flag from the IFR register

**Example**

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

**IRQ_config**  
*Updates Entry in DSPBIOS 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, /* funcAddr */
    0x0000, /* funcArg */
    0x0300 /* ierMask */
};
...
IRQ_config(IRQ_EVT_TINT0,&MyConfig);
```
**IRQ_configArgs**

*Updates entry in DSPBIOS dispatch table*

**Function**

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

**Arguments**

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

**Return Value**

None

**Description**

Updates DSPBIOS dispatch table entry for the specified event.

**Example**

```c
IRQ_configArgs(EventId, funcAddr, funcArg, ierMask);
```

---

**IRQ_disable**

*Disables specified event*

**Function**

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

**Arguments**

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

**Return Value**

```c
intm
```

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

**Description**

Disables the specified event, by modifying the IMR register.

**Example**

```c
int intm;
intm = IRQ_disable(IRQ_EVT_TINT0);
```
**IRAQ_enable**

*Enables specified event*

**Function**

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

**Arguments**

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

**Return Value**

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

**Description**

Enables the specified event.

**Example**

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

**IRAQ_getArg**

*Gets value for specified event*

**Function**

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

**Arguments**

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

**Return Value**

Value of argument

**Description**

Gets the value for specified event.

**Example**

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

**Function**

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

**Arguments**
- `EventId`: Event ID, see IRQ_EVT_NNNN (Table 10–3) for a complete list of events. Or, use the PER_getEventId() function to get the EventId.
- `Config`: Pointer to configuration structure

**Return Value**
None

**Description**
Reads the current configuration for the IRQ event.

**Example**
```c
IRQ_Config ConfigRead;
...
IRQ_getConfig(IRQ_EVT_TINT0, &ConfigRead);
```

**IRQ_globalDisable**

**Globally Disables Interrupts**

**Function**

```c
int IRQ_globalDisable(
);
```

**Arguments**
None

**Return Value**
- `intm`: Returns the old INTM value

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

**Example**
```c
int oldgie;
oldgie = IRQ_globalDisable();
```
**IRQ_globalRestore**

**IRQ_globalEnable**  
*Globally enables all events*

**Function**

```c
int IRQ_globalEnable()
```

**Arguments**

None

**Return Value**

`intm` Returns the old INTM value

**Description**

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

**Example**

```c
Uint32 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**

`gie` Previously saved value

**Description**

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

**Example**

```c
int intm;
intm = IRQ_globalDisable();
...
IRQ_globalRestore (intm);
```
**IRQ_map**

*Maps Event To Physical Interrupt Number*

**Function**

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

**Arguments**

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

**Return Value**

None

**Description**

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

**Example**

```c
IRQ_map(IRQ_EVT_TINT0);
```

---

**IRQ_plug**

*Initializes An Interrupt Vector Table Vector*

**Function**

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

**Arguments**

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

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

**Return Value**

None

**Description**

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

**Example**

```c
IRQ_IsrPtr funcAddr;
.
.
.
IRQ_plug (IRQ_EVT_TINT0, funcAddr);
```
**IRQ_restore**  
Restores the status of the specified event

**Function**
void IRQ_restore(
    Uint16 EventId,
    Uint16 Val
);

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

**Return Value**
None

**Description**
Restores the status of the specified event.

**Example**
Uint16 intm = IRQ_disable(IRQ_EVT_TINT0);
    ....
    IRQ_restore(IRQ_EVT_TINT0, intm);

---

**IRQ_setArg**  
Sets value of argument for DSPBIOS dispatch entry

**Function**
void IRQ_setArg(
    Uint16 EventId,
    Uint32 val
);

**Arguments**
- **EventId**  
  Event ID, see IRQ_EVT_NNNN (Table 10–3) for a complete list of events. Or, use the PER_getEventId() function to get the EventID.
- **val**  
  Value of argument for DSPBIOS dispatch entry

**Return Value**
None

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

**Example**
Uint32 val;
    IRQ_setArg(IRQ_EVT_TINT0, val);
**IRQ_setVecs**

Sets the base address of the interrupt vectors

**Function**

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

**Arguments**

- `iptr`: IVPD pointer to the DSP interrupt vector table

**Return Value**

- `oldVecs`: Returns the old IVPD Pointer to the DSP interrupt Vector table

**Description**

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

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

**Example**

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

**IRQ_test**

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

**Function**

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

**Arguments**

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

**Return Value**

- `Event flag, 0 or 1`

**Description**

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

**Example**

```c
while (!IRQ_test(IRQ_EVT_TINT0);
```
The chapter describes the structure, functions, and macros of the McBSP module.

<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 Structure</td>
<td>11-4</td>
</tr>
<tr>
<td>11.3 Functions</td>
<td>11-6</td>
</tr>
<tr>
<td>11.4 Macros</td>
<td>11-24</td>
</tr>
<tr>
<td>11.5 Configuring the McBSP Module Using CSL GUI</td>
<td>11-42</td>
</tr>
<tr>
<td>11.6 Examples</td>
<td>11-50</td>
</tr>
</tbody>
</table>
11.1 Overview

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

Table 11–1. McBSP Configuration Structure

<table>
<thead>
<tr>
<th>Structure</th>
<th>Purpose</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>11-4</td>
</tr>
</tbody>
</table>

Table 11–2. McBSP Functions

(a) Primary Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
<th>See page...</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCBSP_close()</td>
<td>Closes a McBSP port</td>
<td>11-10</td>
</tr>
<tr>
<td>MCBSP_config()</td>
<td>Sets up the McBSP port using the configuration structure (MCBSP_Config)</td>
<td>11-10</td>
</tr>
<tr>
<td>MCBSP_configArgs()</td>
<td>Sets up the McBSP port using the register values passed to the function</td>
<td>11-12</td>
</tr>
<tr>
<td>MCBSP_open()</td>
<td>Opens a McBSP port</td>
<td>11-17</td>
</tr>
<tr>
<td>MCBSP_start()</td>
<td>Start a transmit and/or receive for a McBSP port</td>
<td>11-20</td>
</tr>
</tbody>
</table>
Table 11–2. McBSP Functions (Continued)

(b) Auxiliary Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
<th>See page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCBSP_getConfig()</td>
<td>Get McBSP channel configuration</td>
<td>11-15</td>
</tr>
<tr>
<td>MCBSP_getPort()</td>
<td>Get McBSP Port number used in given handle</td>
<td>11-15</td>
</tr>
<tr>
<td>MCBSP_read16()</td>
<td>Performs a direct 16-bit read from the data receive register DRR1</td>
<td>11-17</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>11-18</td>
</tr>
<tr>
<td>MCBSP_reset()</td>
<td>Resets the given serial port</td>
<td>11-18</td>
</tr>
<tr>
<td>MCBSP_ffull()</td>
<td>Reads the RFULL bit SPCR1 register</td>
<td>11-18</td>
</tr>
<tr>
<td>MCBSP_rrdy()</td>
<td>Reads the RRDY status bit of the SPCR1 register</td>
<td>11-19</td>
</tr>
<tr>
<td>MCBSP_write16()</td>
<td>Writes a 16-bit value to the serial port data transmit register, DXR1</td>
<td>11-22</td>
</tr>
<tr>
<td>MCBSP_write32()</td>
<td>Writes two 16-bit values to the two serial port data transmit registers, DXR2 (16-bit MSB) and DXR1 (16-bit LSB)</td>
<td>11-22</td>
</tr>
<tr>
<td>MCBSP_xempty()</td>
<td>Reads the XEMPTY bit from the SPCR2 register</td>
<td>11-23</td>
</tr>
<tr>
<td>MCBSP_xrdy()</td>
<td>Reads the XRDY status bit of the SPCR2 register</td>
<td>11-23</td>
</tr>
</tbody>
</table>

(c) Interrupt Control Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
<th>See page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCBSP_getRcvEventId()</td>
<td>Retrieves the receive event ID for the given port</td>
<td>11-16</td>
</tr>
<tr>
<td>MCBSP_getXmtEventId()</td>
<td>Retrieves the transmit event ID for the given port</td>
<td>11-16</td>
</tr>
</tbody>
</table>

(d) Multichannel Control Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
<th>See page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCBSP_channelDisable()</td>
<td>Disables one or several McBSP channels</td>
<td>11-6</td>
</tr>
<tr>
<td>MCBSP_channelEnable()</td>
<td>Enables one or several McBSP channels of the selected register</td>
<td>11-7</td>
</tr>
<tr>
<td>MCBSP_channelStatus()</td>
<td>Returns the channel status</td>
<td>11-9</td>
</tr>
</tbody>
</table>
11.2 Configuration Structure

This section lists the configuration structure for the McBSP module.

**McBSP_Config**

McBSP configuration structure used to setup McBSP port

<table>
<thead>
<tr>
<th>Structure</th>
<th>McBSP_Config</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Members</strong></td>
<td></td>
</tr>
<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>
</tbody>
</table>

For devices supporting 128 channels:

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

For devices that do not support 128 channels:

| Uint16 rcera       |          |
| Uint16 rcerb       |          |
| Uint16 xcerb       |          |
**Description**  McBSP configuration structure used to setup a McBSP port. You create and initialize this structure then pass its address to the MCBSP_config() function. You can use literal values or the MCBSP_RMK macros to create the structure member values.

**Example**

```c
MCBSP_Config MyConfig = {
  0x8001, /* spcr1 */
  0x0001, /* spcr2 */
  0x0000, /* rcr1 */
  0x0000, /* rcr2 */
  0x0000, /* xcr1 */
  0x0000, /* xcr2 */
  0x0001, /* srgr1 */
  0x0200, /* srgr2 */
  0x0000, /* mcr1 */
  0x0000, /* mcr2 */
  0x0000, /* pcr */
  0x0000, /* rcera */
  0x0000, /* rcerb */
  0x0000, /* xcera */
  0x0000, /* xcerb */
};
```

```c
hMcbsp = MCBSP_open(MCBSP_PORT0, MCBSP_OPEN_RESET)
...
MCBSP_config(hMcbsp, &MyConfig);
```
11.3 Functions

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

**MCBSP_channelDisable**  
_Disables one or several McBSP channels_

**Function**

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

**Arguments**

- **hMcbsp**  Handle to McBSP port obtained by MCBSP_open()
- **RegName**  Receive and Transmit Channel Enable Registers:
  - RCERA
  - RCERB
  - XCERA
  - XCERB
  - For devices supporting 128 channels (see section 1.8) add:
    - RCERC
    - RCERD
    - RCERE
    - RCERF
    - RCERG
    - RCERH
    - XCERC
    - XCERD
    - XCERE
    - XCERF
    - XCERG
    - XCERH
- **Channels**  Available values for the specific RegName are:
  - MCBSP_CHAN0
  - MCBSP_CHAN1
  - MCBSP_CHAN2
  - MCBSP_CHAN3
  - MCBSP_CHAN4
MCBSP_channelDisable

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

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

Example

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

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

MCBSP_channelEnable

Enables one or several McBSP channels of selected register

Function

void MCBSP_channelEnable(
    MCBSP_Handle hMbsp,
    Uint16 RegName,
    Uint16 Channels
);

Arguments

- hMbsp: Handle to McBSP port obtained by MCBSP_open()
- RegName: Receive and Transmit Channel Enable Registers:
  - RCERA
  - RCERB
  - XCERA
  - XCERB
### MCBSP_channelEnable

For devices supporting 128 channels (see section 1.8) add:
- RCERC
- RCERD
- RCERE
- RCERF
- RCERG
- RCERH
- XCERC
- XCRD
- XCRE
- XCERF
- XCERG
- XCRH

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

**Return Value**
None

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

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

**Example**

```c
/* 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 Module 11-9
MCBSP_close

| CHAN11 | CHAN12 | CHAN13 | CHAN14 | CHAN15 |

Return Value: Channel status
- 0 - Disabled
- 1 - Enabled

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

Example:
```c
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);
```

Closes McBSP port

Function: void MCBSP_close(MCBSP_Handle hMcbsp);

Arguments:
- hMcbsp: Handle to McBSP port obtained by MCBSP_open()

Return Value: None

Description: Closes a McBSP port previously opened via MCBSP_open(). The registers for the McBSP port are set to their power-on defaults and any associated interrupts are disabled and cleared.

Example:
```c
MCBSP_close(hMcbsp);
```

Sets up McBSP port using configuration structure

Function: void MCBSP_config(MCBSP_Handle hMcbsp, MCBSP_Config *Config);

Example:
```c
MCBSP_config(hMcbsp, Config);
```
**Arguments**

- **hMcbsp**  Handle to McBSP port obtained by MCBSP_open()
- **Config**  Pointer to an initialized configuration structure

**Return Value**

None

**Description**

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

**Note:**

If you want to configure all McBSP registers without starting the McBSP port, use MCBSP_config() without setting the SPCR2 (XRST, RRST, GRST, and FRST) fields. Then,

- Step 1: Start Transmit and receive
- Step 2: Write the first data valid to the DXR registers
- Step 3: Start Sample rate generator and framesync

This guarantees that the correct value is transmitted/received.

**Example**

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

For complete examples, refer to section 11.4.
MCBSP_configArgs

Sets up McBSP port using register values passed in

Function

void MCBSP_configArgs(
    MCBSP_Handle hMcbsp,
    Uint16 spcr1,
    Uint16 spcr2,
    Uint16 rcr1,
    Uint16 rcr2,
    Uint16 xcr1,
    Uint16 xcr2,
    Uint16 srgr1,
    Uint16 srgr2,
    Uint16 mcr1,
    Uint16 mcr2,
    Uint16 pcr,

    For devices that support 128 channels:
        Uint16 rcera,
        Uint16 rcerb,
        Uint16 rcerc,
        Uint16 rcerd,
        Uint16 rcere,
        Uint16 rcerf,
        Uint16 rcerg,
        Uint16 rcerh,
        Uint16 xcera,
        Uint16 xcerb,
        Uint16 xcerc,
        Uint16 xcerd,
        Uint16 xcere,
        Uint16 xcerf,
        Uint16 xcerg,
        Uint16 xcerh,

    For devices that do not support 128 channels:
        Uint16 rcera,
        Uint16 rcerb,
        Uint16 xcera,
        Uint16 xcerb
);
<table>
<thead>
<tr>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hMcbsp</td>
<td>Handle to McBSP port obtained by MCBSP_open()</td>
</tr>
<tr>
<td>spcr1</td>
<td>Serial port control register 1 value</td>
</tr>
<tr>
<td>spcr2</td>
<td>Serial port control register 2 value</td>
</tr>
<tr>
<td>rcr1</td>
<td>Receive control register 1 value</td>
</tr>
<tr>
<td>rcr2</td>
<td>Receive control register 2 value</td>
</tr>
<tr>
<td>xcr1</td>
<td>Transmit control register 1 value</td>
</tr>
<tr>
<td>xcr2</td>
<td>Transmit control register 2 value</td>
</tr>
<tr>
<td>srgr1</td>
<td>Sample rate generator register 1 value</td>
</tr>
<tr>
<td>srgr2</td>
<td>Sample rate generator register 2 value</td>
</tr>
<tr>
<td>mcr1</td>
<td>Multi-channel control register 1 value</td>
</tr>
<tr>
<td>mcr2</td>
<td>Multi-channel control register 2 value</td>
</tr>
<tr>
<td>pcr</td>
<td>Pin control register value</td>
</tr>
<tr>
<td>rcex</td>
<td>Receive channel enable register partition x value</td>
</tr>
<tr>
<td>xcrex</td>
<td>Transmit channel enable register partition x value</td>
</tr>
</tbody>
</table>

Where x = A, B, C, D, E, F, G, H

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>
Sets up the MCBSP port using the register values that are passed. The register values are written to the port registers.

**Note:**

If you want to configure all MCBSP registers without starting the MCBSP port, use `MCBSP_config()` without setting the SPCR2 (XRST, RRST, GRST, and FRST) fields. Then,

- Step 1: Start Transmit and receive
- Step 2: Write the first data valid to the DXR registers
- Step 3: Start Sample rate generator and framesync

This guarantees that the correct value is transmitted/received.

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

**Example**

```c
MCBSP_configArgs(hMcbsp,
    0x8001, /* spcr1 */
    0x0001, /* spcr2 */
    0x0000, /* rcr1 */
    0x0000, /* rcr2 */
    0x0000, /* xcr1 */
    0x0000, /* xcr2 */
    0x0000, /* srgr1 */
    0x2000, /* srgr2 */
    0x0000, /* mcrl */
    0x0000, /* mcr2 */
    0x0000 /* pcr */
    0x0000, /* rcera*/
    0x0000, /* rcerb*/
    0x0000, /* xcera*/
    0x0000 /* xcerb*/

);```
**MCBSP_getConfig**  
*Get MCBSP channel configuration*

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

**Arguments**
- **hMcbsp** Handle to McBSP port; (see MCBSP_open())
- **Config** Pointer to an initialized configuration structure (see MCBSP_Config)

**Return Value**
None

**Description**
Get the current configuration for the McBSP port used by handle. This is accomplished by reading the actual McBSP port registers and fields and storing them back in the Config structure.

**Example**
```c
MCBSP_Config  ConfigRead;
...
myHandle = MCBSP_open (MCBSP_PORT0, 0);
MCBSP_getConfig (myHandle, &ConfigRead);
```

**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 receive event ID for given port*

**Function**
Uint16 MCBSP_getRcvEventId(
    MCBSP_Handle hMcbsp
);

**Arguments**
hMcbsp  
Handle to McBSP port obtained by MCBSP_open()

**Return Value**
Receiver event ID

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

**Example**
Uint16 RecvEventId;
...
RecvEventId = MCBSP_getRcvEventId(hMcbsp);
IRQ_enable(RecvEventId);

**MCBSP_getXmtEventId**  
*Retrieves transmit event ID for given port*

**Function**
Uint16 MCBSP_getXmtEventId(
    MCBSP_Handle hMcbsp
);

**Arguments**
hMcbsp  
Handle to McBSP port obtained by MCBSP_open()

**Return Value**
Transmitter event ID

**Description**
Simple replace receive for transmit. Use this ID to manage the event using the IRQ module.

**Example**
Uint16 XmtEventId;
...
XmtEventId = MCBSP_getXmtEventId(hMcbsp);
IRQ_enable(XmtEventId);
**MCBSP_open**  
*Opens McBSD port*

**Function**

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

**Arguments**

- **devNum**  
  McBSP device (port) number:
  - MCBSP_PORT0
  - MCBSP_PORT1
  - MCBSP_PORT2 (if available in the device)
  - MCBSP_PORT_ANY

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

**Return Value**

Device Handle

**Description**

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

If the MCBSP_OPEN_RESET is specified, the McBSP port registers are set to their power-on defaults and any associated interrupts are disabled and cleared.

**Example**

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

**MCBSP_read16**  
*Performs 16-bit data read*

**Function**

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

**Arguments**

- **hMcbsp**  
  Handle to McBSP port obtained by MCBSP_open()

**Return Value**

Data read for McBSP receive port.

**Description**

Directly reads a 16-bit value from the McBSP data receive register DRR1. Depending on the receive word data length you have selected in the RCR1/RCR2 registers, the actual data could be 8, 12, or 16 bits long. This function does not verify if new valid data as been received. Use MCBSP_rrdy() prior to calling MCBSP_read32() for this purpose.

**Example**

```c
Uint16 Data;
...
Data = MCBSP_read16(hMcbsp);
```
MCBSP_read32

**Performs 32-bit data read**

**Function**

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

**Arguments**

- `hMcbsp`: Handle to McBSP port obtained by MCBSP_open()

**Return Value**

- Data (MSW-LSW ordering)

**Description**

A 32-bit read. First, the 16-bit MSW (Most significant word) is read from register DRR2. Then, the 16-bit LSW (least significant word) is read from register DRR1. Depending on the receive word data length you have selected in the RCR1/RCR2 register, the actual data could be 20, 24, or 32 bits. This function does not verify that new valid data has been received. Use MCBSP_rrdy() prior to calling MCBSP_read32() for this purpose.

**Example**

```c
Uint32 Data;
...
Data = MCBSP_read32(hMcbsp);
```

MCBSP_reset

**Resets given serial port**

**Function**

```c
void MCBSP_reset(
    MCBSP_Handle hMcbsp
);
```

**Arguments**

- `hMcbsp`: Handle to McBSP port obtained by MCBSP_open()

**Return Value**

- None

**Description**

Resets the given serial port. If you use INV (-1) for hMcbsp, all serial ports are reset. Actions Taken:

- All serial port registers are set to their power-on defaults.
- All associated interrupts are disabled and cleared.

**Example**

```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) or 1 (receive buffer full)

**Description**

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

**Example**

```c
if (MCBSP_rfull(hMcbsp)) {
    ...
}
```

---

**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) or 1 (new data has been received)

**Description**

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

**Example**

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

**MCBSP_start** *Starts 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:

  \[
  \text{SampleRateGenDelay} = \frac{2 \times \text{Sample Rate Generator Clock period}}{4 \times C54x\_Instruction\_Cycle}
  \]

A default value of:

- MCBSP_SAMPLE_RATE_DELAY_DEFAULT (0xFFFF value)

  can be used (maximum value).

**Return Value**

None
Description

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

Note:

If you want to configure all MCBSP registers without starting the MCBSP port, use MCBSP_config() without setting the SPCR2 (XRST, RRST, GRST, and FRST) fields. Then,

- Step 1: Start Transmit and receive
- Step 2: Write the first data valid to the DXR registers
- Step 3: Start Sample rate generator and framesync

This guarantees that the correct value is transmitted/received (see example 2).

Example 1

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

Example 2

```c
/* Step 1 */
MCBSP_config(hMcbsp, &MyConfig); 
MCBSP_start(hMcbsp, MCBSP_RCV_START|MCBSP_XMIT_START, 0x200); 

/* Step 2 */
MCBSP_write16(hMcbsp, 0x1234); 

/* Step 3 */
MCBSP_start(hMcbsp, MCBSP_SRGR_START|MCBSP_SRGR_FRAMESYNC, 
            0x200); 
```
**MCBSP_write16**

*Writes a 16-bit data value*

**Function**

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

**Arguments**

- **hMcbsp** Handle to McBSP port obtained by MCBSP_open()
- **Val** 16-bit data value to be written to McBSP transmit register.

**Return Value**

None

**Description**

Directly writes a 16-bit value to the serial port data transmit register; DXR1. Depending on the receive word data length you have selected in the RCR1/RCR2 registers, the actual data could be 8, 12, or 16 bits long. *This function does not check if the transmitter is ready.* Use MCBSP_xrdy() prior to calling MCBSP_write16() for this purpose.

**Example**

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

---

**MCBSP_write32**

*Writes a 32-bit data value*

**Function**

```c
Void MCBSP_write32(
    MCBSP_Handle hMcbsp,
    Uint32 Val
);
```

**Arguments**

- **hMcbsp** Handle to McBSP port obtained by MCBSP_open()
- **Val** 32-bit data value

**Return Value**

None

**Description**

Depending on the transmit word data length you have selected in the XCR1|XCR2 registers, the actual data could be 20, 24, or 32 bits long. *This function does not verify that all valid data has been transmitted.* Use MCBSP_xrdy(), prior to calling MCBSP_write32(), for this purpose.

**Example**

```c
MCBSP_write32(hMcbsp, 0x12345678);
```
**MCBSP_xempty**  
*Reads XEMPTY bit from SPCR2 register*

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

**Arguments**  
hMbsp  Handle to McBSP port obtained by MCBSP_open()

**Return Value**  
XEMPTY  Returns XEMPTY bit of SPCR2 register, 0 (transmit buffer empty) or 1 (transmit buffer full)

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

**Example**  
if (MCBSP_xempty(hMbsp)) {
    ...
}

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

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

**Arguments**  
hMbsp  Handle to McBSP port obtained by MCBSP_open()

**Return Value**  
XRDY  Returns XRDY status bit of SPCR2, 0 (not ready to transmit) and 1 (ready to transmit).

**Description**  
Reads the XRDY status bit of the SPCR2 register. A “1” indicates that the transmitter is ready to transmit a new word. A “0” indicates that the transmitter is not ready to transmit a new word.

**Example**  
if (MCBSP_xrdy(hMbsp)) {
    ...
    MCBSP_write16 (hMbsp, 0x1234);
    ...
}
11.4 Macros

As covered in section 1.5, the CSL offers a collection of macros to get individual access to the peripheral registers and fields.

The following are the list of macros available for the MCBSP. To use these macros, include “csl_mcbsp.h”.

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

Table 11–3 lists the macros available for a MCBSP channel using the channel number as part of the register name.

Table 11–4 lists the macros available for a MCBSP channel using its corresponding handle.

Table 11–3. MCBSP CSL Macros (using port number)

(a) Macros to read/write MCBSP register values

<table>
<thead>
<tr>
<th>Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCBSP_RGET()</td>
</tr>
<tr>
<td>MCBSP_RSET()</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>
</tr>
</thead>
<tbody>
<tr>
<td>MCBSP_FGET()</td>
</tr>
<tr>
<td>MCBSP_FSET()</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCBSP_REG_RMK()</td>
</tr>
<tr>
<td>MCBSP_FMK()</td>
</tr>
</tbody>
</table>

(d) Macros to read a register address

<table>
<thead>
<tr>
<th>Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCBSP_ADDR()</td>
</tr>
</tbody>
</table>
### Table 11–4. MCBSP CSL Macros (using handle)

(a) Macros to read/write MCBSP register values

<table>
<thead>
<tr>
<th>Macro</th>
<th>See page...</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCBSP_RGETH()</td>
<td>11-34</td>
</tr>
<tr>
<td>MCBSP_RSETH()</td>
<td>11-36</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Macro</th>
<th>See page...</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCBSP_FGETH()</td>
<td>11-37</td>
</tr>
<tr>
<td>MCBSP_FSETH()</td>
<td>11-38</td>
</tr>
</tbody>
</table>

(c) Macros to read a register address

<table>
<thead>
<tr>
<th>Macro</th>
<th>See page...</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCBSP_ADDRH()</td>
<td>11-40</td>
</tr>
</tbody>
</table>
**MCBSP_RGET**

*Get the value of a MCBSP register*

**Macro**

`Uint16 MCBSP_RGET (REG#)`

**Arguments**

- **REG#** Register name with channel number (#) where
  - `# = 0,1, (2: depending on the device)
  - `DRR1#`
  - `DRR2#`
  - `DXR1#`
  - `DXR2#`
  - `SPCR1#`
  - `SPCR2#`
  - `RCR1#`
  - `RCR2#`
  - `XCR1#`
  - `XCR2#`
  - `SRGR1#`
  - `SRGR2#`
  - `MCR1#`
  - `MCR2#`
  - `PCR#`
  - `RCERA#`
  - `RCERB#`
  - `XCERA#`
  - `XCELERB#

For devices supporting 128-channels, add:
- `RCERC#`
- `XCERC#`
- `RCERD#`
- `XCERD#`
- `RCERE#`
- `XCELERE#`
- `RCERF#`
- `XCELERF#`
- `RCERG#`
- `XCELERG#`
- `RCERH#`
- `XCELERH#`

**Return Value**

Value of register
**Description**
Returns the MCBSP register value

**Example**
```
Uint16 myVar;
...
myVar = MCBSP_RGET(RCR10); /*get register RCR1 of channel 0 */
```

---

**Set the value of a MCBSP register**

**Macro**
Void MCBSP_REG_SET (MCBSP_Handle hMcbsp, Uint16 RegVal)

**Arguments**

<table>
<thead>
<tr>
<th>REG#</th>
<th>Register name with channel number (#) where # = 0,1, (2: depending on the device)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRR1#</td>
<td>DRR1#</td>
</tr>
<tr>
<td>DRR2#</td>
<td>DRR2#</td>
</tr>
<tr>
<td>DXR1#</td>
<td>DXR1#</td>
</tr>
<tr>
<td>DXR2#</td>
<td>DXR2#</td>
</tr>
<tr>
<td>SPCR1#</td>
<td>SPCR1#</td>
</tr>
<tr>
<td>SPCR2#</td>
<td>SPCR2#</td>
</tr>
<tr>
<td>RCR1#</td>
<td>RCR1#</td>
</tr>
<tr>
<td>RCR2#</td>
<td>RCR2#</td>
</tr>
<tr>
<td>XCR1#</td>
<td>XCR1#</td>
</tr>
<tr>
<td>XCR2#</td>
<td>XCR2#</td>
</tr>
<tr>
<td>SRGR1#</td>
<td>SRGR1#</td>
</tr>
<tr>
<td>SRGR2#</td>
<td>SRGR2#</td>
</tr>
<tr>
<td>MCR1#</td>
<td>MCR1#</td>
</tr>
<tr>
<td>MCR2#</td>
<td>MCR2#</td>
</tr>
<tr>
<td>PCR#</td>
<td>PCR#</td>
</tr>
<tr>
<td>RCERA#</td>
<td>RCERA#</td>
</tr>
<tr>
<td>RCERB#</td>
<td>RCERB#</td>
</tr>
<tr>
<td>XCERA#</td>
<td>XCERA#</td>
</tr>
<tr>
<td>XCELER#</td>
<td>XCELER#</td>
</tr>
<tr>
<td>XCERF#</td>
<td>XCERF#</td>
</tr>
<tr>
<td>RCERG#</td>
<td>RCERG#</td>
</tr>
</tbody>
</table>

For devices supporting 128-channels, add:
- RCERC#
- XCELER#
- XCELERD#
- RCERD#
- XCELERE#
- RCERE#
- XCELERF#
- RCERF#
- XCELERG#
**MCBSP_REG_RMK**

regval Register value needed to write to register REG

**Return Value**
None

**Description**
Set the MCBSP register REG value to regval

**Example**
MCBSP_RSET(RCR10, 0x4); /* RCR1C for channel 0 = 0x4 */

---

**MCBSP_REG_RMK**  *Creates a register value based on individual field values*

**Macro**
Uint16 MCBSP_REG_RMK(fieldval_n,...,fieldval_0)

**Arguments**
REG

Only writable registers containing more than one field are supported by this macro. Please note that the channel number is not used as part of the register name.

SPCR1  
SPCR2  
RCR1  
RCR2  
XCR1  
XCR2  
SRGR1  
SRGR2  
MCR1  
MCR2  
PCR  
RCERA  
RCERB  
XCERA  
XCERB

For devices supporting 128-channels, add:
RCERC  
XCERC  
RCERD  
XCERD  
RCERE  
XCERE  
RCERF  
XCERF
fieldval_n  field values to be assigned to the register fields rules to follow:
- Only writable fields are allowed
- Start from Most-significat field first
- Value should be a right-justified constant. If fieldval_n value exceeds the number of bits allowed for that field, then fieldval_n is truncated accordingly.

Return Value  value of register that corresponds to the concatenation of values passed for the fields. (writable fields only)

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

Example  /* frame length, word length */
myVal = MCBSP_RCR1_RMK Uint16 myVal (4,3);

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

MCBSP_REG_RMK macros are typically used to initialize a MCBSP configuration structure used for the MCBSP_config() function. For more examples see section 11.6.

**MCBSP_FMK**  Creates a register value based on individual field values

**Macro**  Uint16 MCBSP_FMK (REG, FIELD, fieldval)

**Arguments**  
REG  Only writable registers containing more than one field are supported by this macro. Please note that the channel number is not used as part of the register name.
- SPCR1
- SPCR2
- RCR1
- RCR2
- XCR1
- XCR2
- SRGR1
MCBSP_FMCK

SRGR2
MCR1
MCR2
PCR
RCERA
RCERB
XCERA
XCERB

For devices supporting 128-channels, add:
RCERC
XCERC
RCERD
XCERD
RCERE
XCERE
RCERF
XCERF
RCERG
XCERG
RCERH
XCERH

FIELD  Symbolic name for field of register REG. Possible values are the
field names as listed in the C54x Register Reference Guide.  
**Only writable fields are allowed.**

fieldval  field values to be assigned to the register fields rules to follow:
- Only writable fields are allowed
- Start from Most-significant field first
- Value should be a right-justified constant. If fieldval_n value exceeds the number of bits allowed for that field, then
  fieldval_n is truncated accordingly.

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

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

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

**MCBSP_FGET**

Gets the value of a register field

**Macro**

Uint16 MCBSP_FGET (REG#, FIELD)

**Arguments**

REG# Register name with channel number (#) where
# = 0, 1, (2: depending on the device)
DRR1#
DRR2#
DXR1#
DXR2#
SPCR1#
SPCR2#
RCR1#
RCR2#
XCR1#
XCR2#
SRGR1#
SRGR2#
MCR1#
MCR2#
PCR#
RCERA#
RCERB#
XCERA#
XCERB#

For devices supporting 128-channels, add:
RCERC#
XCERC#
RCERD#
XCERD#
RCERE#
XCERE#
RCERF#
XCERF#
RCERG#
XCERG#
RCERH#
XCERH#
FIELD symbolic name for field of register REG. Possible values are the field names listed in the C54x Register Reference Guide (Appendix x) Only readable fields are allowed.

Return Value
Value of register field

Description
Gets the MCBSP register FIELD value

Example
Uint16 myVar;

... 
myVar = MCBSP_FGET(RCR20, RPHASE);

MCBSP_FSET

Sets the value of a register field

Macro
Void MCBSP_FSET (REG#, FIELD, fieldval)

Arguments
REG# Register name with channel number (#) where 
# = 0,1, (2: depending on the device) 
DRR1# DRR2# DXR1# DXR2# SPCR1# SPCR2# RCR1# RCR2# XCR1# XCR2# SRGR1# SRGR2# MCR1# MCR2# PCR# RCERA# RCERB# XCERA# XCERB#

For devices supporting 128-channels, add:
RCERC# XCERC# RCERD# XCERD#
FIELD  Symbolic name for field of register REG. Possible values: Field names as listed in the C54x Register Reference Guide. Only writable fields are allowed.

fieldval  field values to be assigned to the register fields rules to follow:
- Only writable fields are allowed
- Start from Most-significant field first
- Value should be a right-justified constant. If fieldval_n value exceeds the number of bits allowed for that field, then fieldval_n is truncated accordingly.

Return Value  None

Description  Set the MCBSP register value to regval

Example  For Registers:
MCBSP_FSET (RCR20, RPHASE, 2);

Get the address of a given register

Macro  Uint16 MCBSP_ADDR (REG#)

Arguments  REG#  Register name with channel number (#) where 
# = 0,1, (2: depending on the device)
  DRR1#
  DRR2#
  DXR1#
  DXR2#
  SPCR1#
  SPCR2#
  RCR1#
  RCR2#
  XCR1#
### MCBSP_RGETH

XCR2#
SRGR1#
SRGR2#
MCR1#
MCR2#
PCR#
RCERA#
RCERB#
XCERA#
XCERB#

For devices supporting 128-channels, add:
RCERC#
XCERC#
RCERD#
XCERD#
RCERE#
XCERE#
RCERF#
XCERF#
RCERG#
XCERG#
RCERH#
XCERH#

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Address of register REG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Get the address of a given MCBSP register.</td>
</tr>
<tr>
<td>Example</td>
<td>For Registers: myVar = MCBSP_ADDR(RCR10); /*get register RCR1 of channel 0 */</td>
</tr>
</tbody>
</table>

---

### Get the value of a MCBSP register used in a handle

**Macro**

Uint16 MCBSP_RGETH (MCBSP_Handle hMcbsp, REG)

**Arguments**

- **hMcbsp**: Handle to MCBSP channel that identifies the MCBSP channel used.
- **REG**: Similar to register in MCBSP_RGET(), but without channel number (#). DRR1
MCBSP_RGETH

DRR2
DXR1
DXR2
SPCR1
SPCR2
RCR1
RCR2
XCR1
XCR2
SRGR1
SRGR2
MCR1
MCR2
PCR
RCERA
RCERB
XCERA
XCERB

For devices supporting 128-channels, add:
RCERC
XCERC
RCERD
XCERD
RCERE
XCERE
RCERF
XCERF
RCERG
XCERG
RCERH
XCERH

Return Value
value of register

Description
Returns the MCBSP register value for register REG for the channel associated with handle.

Example
MCBSP_Handle myHandle;
Uint16    myVar;
...
myHandle = MCBSP_open (MCBSP_PORT0, MCBSP_OPEN_RESET);
...
myVar = MCBSP_RGETH(myHandle, RCR1);
**MCBSP_RSETH**

Set the value of a MCBSP register

**Macro**

Void MCBSP_RSETH (MCBSP_Handle hMcbsp, REG, Uint16 RegVal)

**Arguments**

- **hMcbsp** Handle to McBSP port that identifies specific McBSP port being used.
- **REG#** Similar to register in MCBSP_RGET(), but without channel number (#).
  - DRR1
  - DRR2
  - DXR1
  - DXR2
  - SPCR1
  - SPCR2
  - RCR1
  - RCR2
  - XCR1
  - XCR2
  - SRGR1
  - SRGR2
  - MCR1
  - MCR2
  - PCR
  - RCERA
  - RCERB
  - XCERA
  - XCELER

For devices supporting 128-channels, add:
- RCERC
- XCERC
- RCERD
- XCERD
- RCERE
- XCERE
- RCERF
- XCERF
- RCERG
- XCERG
- RCERH
- XCERH
regval  value to write to register REG for the channel associated with handle.

Return Value  None

Description  Set the MCBSP register REG for the channel associated with handle to the value regval.

Example  
```c
MCBSP_Handle myHandle;
...
myHandle = MCBSP_open (MCBSP_PORT0, MCBSP_OPEN_RESET);
...
MCBSP_RSETH(myHandle, RCR1, 0x4);
```

---

**MCBSP_FGETH**  
*Get the value of a register field*

**Macro**  
Uint16 MCBSP_FGETH  (MCBSP_Handle Hmcbsp, REG, FIELD)

**Arguments**  
- **hMcbsp**  Handle to McBSP port that identifies specific McBSP port being used.
- **REG**  Similar to register in MCBSP_RGET(), but without channel number (#).
  - DRR1
  - DRR2
  - DXR1
  - DXR2
  - SPCR1
  - SPCR2
  - RCR1
  - RCR2
  - XCR1
  - XCR2
  - SRGR1
  - SRGR2
  - MCR1
  - MCR2
  - PCR
  - RCERA
  - RCERB
  - XCERA
MCBSP_FGETH

XCB

For devices supporting 128-channels, add:
RCRC
XCCRC
RCRCD
XCRD
RCRCE
XCRCE
RCRFR
XCRFR
RCRFRG
XCRFRG
RCRFRH
XCRFRH

FIELD  symbolic name for field of register REG  Possible values:
Field names listed in the C54x Register Reference Guide
Only readable fields are allowed.

Return Value
Value of register field given by FIELD and of REG used by handle.

Description
Gets the MCBSP register FIELD value

Example
MCBSP_Handle myHandle;
Uint16 myVar;
...
myHandle = MCBSP_open (MCBSP_PORT0, MCBSP_OPEN_RESET);
...
myVar = MCBSP_FGETH(myHandle, RCR2, RPHASE);

MCBSP_FSETH  Set the value of a register field

Macro
Void MCBSP_FSETH  (MCBSP_Handle hMcbsp, REG, FIELD, fieldval)

Arguments
hMcbsp  Handle to McBSP port that identifies specific McBSP port
being used.

REG#  Similar to register in MCBSP_RGET(), but without channel
number (#).
  DRR1
  DRR2
  DXR1
DXR2
SPCR1
SPCR2
RCR1
RCR2
XCR1
XCR2
SRGR1
SRGR2
MCR1
MCR2
PCR
RCERA
RCERB
XCERA
XCERB

For devices supporting 128-channels, add:
RCERC
XCERC
RCERD
XCERD
RCERE
XCERE
RCERF
XCERF
RCERG
XCERG
RCERH
XCERH

FIELD Symbolic name for field of register REG. Possible values are the field names as listed the C54x Register Reference Guide. **Only writable fields are allowed.**

fieldval field values to be assigned to the register fields rules to follow:

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

**Return Value** None
MCBSP_ADDRH

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

Example
MCBSP_Handle myHandle;
...
myHandle = MCBSP_open (MCBSP_PORT0, MCBSP_OPEN_RESET);
...
MCBSP_FSETH(myHandle, RCR2, RPHASE,1);

MCBSP_ADDRH  Get the address of a given register

Macro
Uint16 MCBSP_ADDR (REG#)

Arguments
hMcbsp  Handle to MCBSP channel that identifies the MCBSP channel used. Use only for MCBSP channel registers. Registers are listed as part of the MCBSP_RGETH macro description.

REG  Similar to register in MCBSP_RGET(), but without channel number (#).
  DRR1
  DRR2
  DXR1
  DXR2
  SPCR1
  SPCR2
  RCR1
  RCR2
  XCR1
  XCR2
  SRGR1
  SRGR2
  MCR1
  MCR2
  PCR
  RCERA
  RCERB
  XCERA
  XCELER

For devices supporting 128-channels, add:
  RCERC
MCBSP_ADDRH

Return Value
Address of register REG

Description
Gets the address of the MCBSP register associated with handle hMCBSP

Example 1
MCBSP_Handle myHandle;
Uint16 myVar;
...
myVar = MCBSP_ADDRH(myHandle, RCR1);
11.5 Configuring the McBSP Module Using CSL GUI

11.5.1 Overview

The McBSP module facilitates configuration/control of the Multichannel Buffered Serial Port (McBSP). The module consists of a configuration manager and a resource manager. The configuration manager allows creation of one or more configuration objects. The configuration objects contain all of the data necessary to set the McBSP Control Registers. The resource manager associates a configuration object with a specified port.

Figure 11–1 illustrates the GPIO sections menu on the CSL graphical user interface (GUI).

Figure 11–1. McBSP Sections Menu

The McBSP includes the following two sections:

- **McBSP Configuration Manager** allows you to create configuration objects. No predefined configuration objects.

- **McBSP Resource Manager** allows you to select a device and to associate a configuration object to that device. Three handle objects are predefined.

11.5.2 McBSP Configuration Manager

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

11.5.2.1 Creating/Inserting a Configuration Object

There is no predefined configuration object available.

To configure a McBSP port through the peripheral registers, you must insert a new configuration object.

To insert a new configuration object, right-click on the McBSP Configuration Manager and select insert mcbspCfg from the drop-down menu. The configuration objects can be renamed. Their use depends upon the on-chip device resources.
11.5.2.2 Deleting/Renaming an Object

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

If a configuration object is used by one of the predefined handle objects of the McBSP Resource Manager, the Delete and Rename options are grayed out and non-usable. The Show Dependency option is accessible and shows which device is using the configuration object (see Figure 2–1, The CSL Tree, on page 2-3).

11.5.2.3 Configuring the Object Properties

The Properties pages allow you to set the Peripheral registers related to the McBSP Port (see Figure 11–2). To access the Properties dialog box, right-click on a configuration object and select Properties. By default, the General page of the Properties dialog box is displayed.

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

- **General:** Allows you to configure the Digital Loopback, ABIS Mode, Breakpoint Emulation.

- **Transmit Modes:** Allows you to configure the Transmit Interrupt mode, Frame Sync, Clock control.

- **Transmit Lengths:** Allows you to configure the Transmit Phase, elements-per-word, elements per frame.

- **Receive Modes:** Allows you to configure the Receive Interrupt mode, Frame Sync, Clock control.

- **Receive Lengths:** Allows you to configure the Receive Phase, elements-per-word, elements per frame.

- **Sample-Rate Generator:** Allows you to configure the Sample-Rate Generator (Frame Setup).
Receive Multichannel: Allows you to configure the Receive Element and Block partitioning.

Transmit Multichannel: Allows you to configure the Transmit Element and Block partitioning.

GPIO: General Purpose I/O pin configuration.

ABIS: Allows you to configure the McBSP ABIS mode.

Some fields are activated according to the setup of the Transmitter, Receiver, and Sample-rate generator options.

Advanced A and B: Summary of the previous pages. These pages contain the full hexadecimal register values and reflects the setting of the options done under the previous pages.

The full register values can be entered directly and the new options will be mirrored on the corresponding pages automatically.

Figure 11–2 depicts the Properties Page.
Each Tab page is composed of several options that are set to a default value (at device reset).
11.5.3 McBSP Resource Manager

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

Figure 11–3 illustrates the McBSP Resource Manager menu on the CSL graphical user interface (GUI).

Figure 11–3. McBSP Resource Manager Menu

11.5.3.1 Predefined Objects

Three handle objects are predefined and each of them is associated with a supported on-chip McBSP port.

- McBSP0 – Default handle name: hMcbsp0
- McBSP1 – Default handle name: hMcbsp1
- McBSP2 – Default handle name: hMcbsp2

Note:
The above objects cannot be deleted. They can be renamed only.

A configuration can be enabled if at least one configuration object was defined previously. See section 11.5.2, McBSP Configuration Manager.

11.5.3.2 Properties Page

You can generate the MCBSP_open() and MCBSP_config() CSL functions through the Properties page.

To access the Properties page, right-click on a predefined McBSP channel and select Properties from the drop-down menu (see Figure 11–4).

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

MCBSP NOTHING is used to indicate that there is no configuration object selected for this serial port.
To pre-initialize a McBSP port, check the Enable Pre-Initialization box. You can then select one of the available configuration objects (see section 11.5.2, McBSP Configuration Manager) for this channel through the pre-initialize drop-down list.

If MCBSP_NOTHING is selected, no configuration object is generated for the related McBSP handle. (see section 11.5.4, C Code Generation for McBSP Module).

In the example shown in Figure 11–4, the Open Handle to McBSP option is checked and the handle object hMcbsp1 is now accessible (The handle object can be renamed by typing the new name in the box provided). The MCBSP_open() function is now generated with hMcbsp0 containing the returned handle address.

11.5.4 C Code Generation for McBSP Module

Two C files are generated from the configuration tool:

- Header file
- Source file.

11.5.4.1 Header File

The header file includes all the csl header files of the modules and contains the McBSP handle and configuration objects defined from the configuration tool (see Example 11–1).
Example 11–1. McBSP Header File

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

11.5.4.2 Source File

The source file includes the declaration of the handle object and the configuration structures (see Example 11–2).

Example 11–2. McBSP Source File (Declaration Section)

```c
/* Config Structures */
MCBSP_Config mcbspCfg0 = {
    0x0000, /* Serial Port Control Register 1 */
    0x0000, /* Serial Port Control Register 2 */
    0x0000, /* Receive Control Register 1 */
    0x0000, /* Receive Control Register 2 */
    0x0000, /* Transmit Control Register 1 */
    0x0000, /* Transmit Control Register 2 */
    0x0000, /* Sample Rate Generator Register 1 */
    0x0000, /* Sample Rate Generator Register 2 */
    0x0000, /* Multi-channel Control Register 1 */
    0x0000, /* Multi-channel Control Register 2 */
    0x0000, /* Pin Control Register */
    0x0000, /* Receive Channel Enable Register Partition A */
    0x0000, /* Receive Channel Enable Register Partition B */
    0x0000, /* Transmit Channel Enable Register Partition A */
    0x0000 /* Transmit Channel Enable Register Partition B */
};

/* Handles */
MCBSP_Handle hMcbsp1;
```

The source file contains the Handle and Configuration Pre-Initialization using the CSL McBSP functions, MCBSP_open() and MCBSP_config() (see Example 11–3).
These two functions are encapsulated in a unique function, CSL_cfgInit(), which is called from your main C file. MCBSP_open() and MCBSP_config() are generated only if Open Handle to McBSP and Enable pre-initialization (with a selected configuration other than MCBSP_NOTHING) are, respectively, checked under the McBSP Resource Manager Properties page.

You must use MCBSP_start() in your code to start the McBSP. The MCBSP_Config structure generated by the CSL GUI does not start the McBSP port transmit or receive operations (unless you force this behavior through the McBSP GUI Advance Tab settings). In the case of a write, this is typically done after you write your first data to the DXR register(s) to guarantee a valid data transfer (see note in MCBSP_start() description on page 11-20).

Example 11–3. McBSP Source File (Body Section)

```c
void CSL_cfgInit()
{

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

Examples for the McBSP module are found in the CCS examples\<target>\csl directory.

Example 11–4 illustrates the McBSP port initialization using MCBSP_config(). The example also explains how to set the McBSP into digital loopback mode and perform 32-bit reads/writes from/to the serial port.

**Example 11–4. McBSP Port Initialization using MCBSP_config**

```c
#include <csl.h>
#include <csl_mcbsp.h>

/* Step 0: This is your McBSP register configuration */
static MCBSP_Config ConfigLoopBack32 = {
    ....
};

void main(void) {
    MCBSP_Handle mhMcbsp;
    Uint32 xmt[n], rcv[n];
    ....

    /* Step 1: Initialize CSL */
    CSL_init();

    /* Step 2: Open and configure the McBSP port */
    mhMcbsp = MCBSP_open(MCBSP_PORT0, MCBSP_OPEN_RESET);
    MCBSP_config(mhMcbsp, &ConfigLoopBack32);

    /* Step 3: Write the first data value and start */
    /* the sample rate generator in the McBSP */
    MCBSP_write32(mhMcbsp, xmt[0]);
    MCBSP_start(mhMcbsp, MCBSP_SRGR_START|MCBSP_SRGR_FRAMESYNC, 0x300u);
    ....

    while (!MCBSP_rrdy(mhMcbsp));
    rcv[0] = MCBSP_read32(mhMcbsp);
```
Example 11–4. McBSP Port Initialization using MCBSP_config (Continued)

```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 */
```
This chapter describes the structure, functions, and macros of the PLL module.

<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 Configuration Structure</td>
<td>12-3</td>
</tr>
<tr>
<td>12.3 Functions</td>
<td>12-4</td>
</tr>
<tr>
<td>12.4 Macros</td>
<td>12-6</td>
</tr>
<tr>
<td>12.5 Configuring the PLL Module Using CSL GUI</td>
<td>12-8</td>
</tr>
</tbody>
</table>
12.1 Overview

The CSL PLL module offers functions and macros to control the Phase Locked Loop fo the C54xx clock.

The PLL module is not handle-based.

Table 12–1 lists the configuration structure to use with the PLL functions. Table 12–2 lists the functions available as part of the PLL module.

Section 12.4 includes a description of available PLL macros.

### Table 12–1. PLL Configuration Structure

<table>
<thead>
<tr>
<th>Structure</th>
<th>Purpose</th>
<th>See page...</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLL_Config</td>
<td>PLL structure that contains the register required to setup the PLL.</td>
<td>12-3</td>
</tr>
</tbody>
</table>

### Table 12–2. PLL Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
<th>See page ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLL_config()</td>
<td>Configure the PLL with the values provided in a configuration structure.</td>
<td>12-4</td>
</tr>
<tr>
<td>PLL_configArgs()</td>
<td>Configures the PLL with values provided as function arguments.</td>
<td>12-4</td>
</tr>
<tr>
<td>PLL_setFreq()</td>
<td>Initializes the PLL to produce the desired CPU output frequency (clkout)</td>
<td>12-5</td>
</tr>
</tbody>
</table>
12.2 Configuration Structure

This section describes the structure in the PLL module.

PLL configuration structure used to set up PLL interface

<table>
<thead>
<tr>
<th>Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLL_Config</td>
<td>PLL configuration structure used to set up PLL interface. You create and initialize this structure and then pass its address to the PLL_config() function.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Members</th>
<th>Description</th>
</tr>
</thead>
</table>
| Uint16 pllmode | available values  
PLL_MODE_DIV = 1 (divide mode)  
PLL_MODE_MUL = 2 (pll multiplier mode)  
This value combines the effect of the PLLNDIV and PLLDIV fields. |
| Uint16 pllcoun | internal lockup counter (number of PLL clock input cycles that the PLL logic should wait before “locking” in the new frequency.) |
| Uint16 pllmul | PLL multiplier register field value. |

For CHIP_5410 and CHIP_5416 add:

<table>
<thead>
<tr>
<th>Members</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uint16 divfct</td>
<td>divide down factor</td>
</tr>
</tbody>
</table>

Example

```c
/* clock_out freq = clock_in freq * final multiplier */
PLL_Config myconfig = {
    PLL_MODE_DIV,
    20,
    1 /* final multiplier = 0.5 */
};
```
PLL_config

12.3 Functions

This section describes the functions in the PLL module.

PLL_config

**Writes a value to set up PLL using configuration structure**

**Function**

void PLL_config (PLL_Config *config);

**Arguments**

Config Pointer to an initialized configuration structure

**Return Value**

one

**Description**

Writes a value to set up the PLL using the configuration structure. The values of the structure are written to the port registers. See also PLL_configArgs() and PLL_Config.

**Example**

PLL_Config MyConfig {
    PLL_MODE_DIV,
    20,
    1
};

PLL_config (&MyConfig);

PLL_configArgs

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

**Function**

void PLL_configArgs (Uint16 pllmode, Uint16 pllmul, Uint16 pllcnt);

**Arguments**

Uint16 pllmode available values:
PLL_MODE_DIV = 1 (divide mode)
PLL_MODE_MUL = 2 (pll multiplier mode)
This value combines the effect of the PLLNDIV and PLLDIV fields.

Uint16 pllcnt Internal lockup counter (number of PLL clock input cycles that the PLL logic should wait before “locking” in the new frequency.)

Uint16 pllmul PLL multiplier register field value.

Uint16 divfct Divide down factor
PLL_configArgs

Return Value: none

Description: Writes to the PLL using the register values passed to the function. The register values are written to the PLL registers. You may use literal values for the arguments.

Clock out frequency is determined as follows:

Example:
PLL_configArgs (PLL_MODE_DIV, 1, 20);

PLL_setFreq

Function: void PLL_setFreq (Uint16 mul, Uint16 div);

Arguments:
- mul: integer multiplier
- div: integer divisor

Where mul should not be further divisible by div. The table below shows valid ranges for mul and div:

<table>
<thead>
<tr>
<th>Range</th>
<th>mul</th>
<th>div</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[1,15]</td>
<td>{1,2,4}</td>
</tr>
</tbody>
</table>

This function does not verify that arguments passed are within valid ranges.

Return Value: None

Description: Initializes the PLL to produce the desired CPU output frequency (clkout)

Example:
PLL_setFreq (1, 2); // set clkout = 1/2 clkin
Macros

12.4 Macros

As covered in section 1.5, CSL offers a collection of macros to get individual access to the peripheral registers (CLKMD) and fields.

The following is a list of macros available for the PLL module. To use them, include “csl_pll.h”.

Table 12–3. PLL CSL Macros

(a) Macros to read/write PLL register values

<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></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>PLL_FMK()</td>
<td>Uint16 PLL_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>PLL_ADDR()</td>
<td>Uint16 PLL_ADDR(REG)</td>
</tr>
</tbody>
</table>

Where:

REG: CLKMD

FIELD: register field name as specified in Appendix A.

- For REG_FSET and REG_FMK, FIELD should be a writable field.
- REG_FGET, the field must be a readable field.
**Macros**

**regVal:** value to write in register (REG)

**fieldVal:** value to write in field (FIELD)

For examples on how to use macros, refer macro sections 6.4 (DMA) and 11.4 (McBSP).
12.5 Configuring the PLL Module Using CSL GUI

12.5.1 Overview

The PLL module facilitates programming of the Phase Locked Loop controlling C54xx clock. The PLL module consists of a configuration manager and a resource manager. The configuration manager allows creation of one or more configuration objects. A configuration object consists of the necessary register settings to control the PLL. The resource manager associates a selected configuration with the PLL.

Figure 12–1 illustrates the PLL sections menu on the CSL graphical user interface (GUI).

Figure 12–1. PLL Sections Menu

The PLL includes the following two sections:

- **PLL Configuration Manager** allows you to create configuration objects by setting the Peripheral registers related to the PLL.
- **PLL Resource Manager** allows you to associate a configuration object to the PLL.

12.5.2 PLL Configuration Manager

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

12.5.2.1 Creating/Inserting a configuration

There is no predefined configuration object.

To configure a PLL setting through the Peripheral Registers, you must insert a new configuration object.

To insert a new configuration object, right-click on the PLL Configuration Manager and select Insert pllCfg. The configuration objects can be renamed.

**Note:**

The number of configuration objects is unlimited. Several configurations can be created. You user can select one for the PLL and can change the configuration later just by selecting another configuration under the PLL Resource Manager. This feature allows you more flexibility and reduces the time required to modify register values.
12.5.2.2 Deleting/Renaming and Object

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

If a configuration object is used by the predefined object of the PLL Resource Manager, the Delete and Rename options are grayed out and non-usable. The Show Dependency option is accessible and shows which device is using the configuration object (see Figure 2–1, The CSL Tree, on page 2-3).

12.5.2.3 Configuring the Object Properties

You can configure object properties through the Properties dialog box (see Figure 12–2). To access the Properties dialog box, right-click on a configuration object and select Properties. By default, the General page of the Properties dialog box is displayed.

The Properties pages allow you to set the Peripheral registers related to the PLL. You can set the configuration options through the following tab page:

- Settings: Allows you to configure the Counter Value, Multiplier, Divide Factor

Figure 12–2 depicts the Properties Page dialog box.

![ PLL Properties Page ]

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

The options represent the fields of the PLL registers; the associated field name is shown in parenthesis. For further details concerning the fields and registers for 5416, refer to the Expansion Bus chapter of the TMS320C54x DSP CPU and Peripherals References Set (SPRU131F).
12.5.3 PLL Resource Manager

The PLL Resource Manager allows you to generate the PLL_config() CSL function.

Because only one PLL is supported, only one resource is available and used as the default.

Figure 12–3 illustrates the PLL Resource Manager menu on the CSL graphical user interface (GUI).

Figure 12–3. PLL Resource Manager Menu

12.5.3.1 Properties Page

You can generate the PLL_config() CSL function through the Properties page.

To access the Properties page, right-click on a predefined PLL channel and select Properties from the drop-down menu (see Figure 12–4).

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

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

To pre-initialize the PLL channel, check the Enable Configuration of PLL box. One of the available configuration objects (see section 6.5.1, PLL Configuration Manager) can then be selected for this channel through the Pre-Initialize drop-down list.

If PLL_NOTHING remains selected, The PLL_config() function will not be generated for the PLL.

In Figure 12–4, the Enable Configuration of PLL option is checked and the PLL_config function will be generated (See section 12.5.4, C Code Generation for PLL Module).
12.5.4 C Code Generation for PLL Module

Two C files are generated from the configuration tool:

- Header file
- Source file.

12.5.4.1 Header File

The header file includes all the csl header files of the modules and contains the PLL configuration objects defined from the configuration tool (see Example 12–1).

Example 12–1. PLL Header File

```c
extern PLL_Config pllCfg0;
```

12.5.4.2 Source File

The source file includes the declaration of the configuration structures (values of the peripheral registers) (see Example 12–2).

Example 12–2. PLL Source File (Declaration Section)

```c
/* Config Structures */
PLL_Config pllCfg0 = {
    0x2, /* PLL Multiplier/Divider Mode */
    0x0, /* PLL Counter Value (PLLCOUNT) */
    0x0, /* PLL Multiplier Value (PLLMUL) */
};
```
Configuring the PLL Module Using CSL GUI

The source file contains the Pre-Initialization PLL API function, PLL_config(). This function is encapsulated into a unique function, CSL_cfgInit(), which is called from your main C file (see Example 12–3).

PLL_config() is generated only if Enable Configuration of PLL is checked under the PLL Resource Manager Properties page (with a selected configuration other than PLL_NOTHING) (see Figure 12–4).

Example 12–3. PLL Source File (Body Section)

```
void CSL_cfgInit()
{

    PLL_config(&pllCfg0);
}
```
The CSL PWR module offers functions to control the power consumption of different sections in the C54x device.

<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 Functions</td>
<td>13-3</td>
</tr>
</tbody>
</table>
13.1 Overview

The CSL PWR module offers functions to control the power consumption of different sections in the C54x device. The PWR module is not handle-based.

Currently, there are no macros available for the power-down module.

Table 13–1 lists the functions for use with the PWR modules that order specific parts of the C54x to power down.

Table 13–1. PWR Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
<th>See page ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>PWR_powerDown</td>
<td>Forces the DSP to enter a power-down(IDLE) state</td>
<td>13-3</td>
</tr>
</tbody>
</table>
13.2 Functions

This section lists the functions in the PWR module.

**PWR_powerDown**  
*Forces the DSP to enter a power-down state*

**Function**

```c
void PWR_powerDown (Uint16 pwrMode, Uint16 wakeMode);
```

**Arguments**

- **mode**  
  - `pwrMode`:
    - `PWR_CPU_DOWN`: CPU goes idle, but peripherals keep running. This corresponds to the IDLE #1 instruction.
    - `PWR_CPU_PER_DOWN`: Both CPU and peripherals power-down. This corresponds to the IDLE #2 instruction.
    - `PWR_CPU_PER_PLL_DOWN`: CPU, peripherals, and PLL power-down. This corresponds to the IDLE #3 instruction.

- **wakeMode** (Valid for all `pwrMode` above)
  - `PWR_WAKEUP_MI`: Wakes up with an unmasked interrupt and jumps to execute the ISRs executed.
  - `PWR_WAKEUP_NMI`: Wakes up with an unmasked interrupt and executes the next instruction (interrupt is not taken).

**Return Value**

None

**Description**

Power-down the device in different power-down and wake-up modes. In the C54x, power-down is achieved by executing an IDLE K instruction.

**Example**

```c
PWR_powerDown (PWR_CPU_DOWN, PWR_WAKEUP_MI);
```
This chapter describes the structure and functions for the TIMER Module.

<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 Structure</td>
<td>14-3</td>
</tr>
<tr>
<td>14.3 Functions</td>
<td>14-4</td>
</tr>
<tr>
<td>14.4 Macros</td>
<td>14-9</td>
</tr>
<tr>
<td>14.5 Configuring the TIMER Module Using CSL GUI</td>
<td>14-12</td>
</tr>
</tbody>
</table>
14.1 Overview

Table 14–1 lists the configuration structure for the TIMER module.

Table 14–2 lists, in the order in which they are typically called, the functions available for use with the TIMER modules.

Section 14.4 includes descriptions for available TIMER macros.

Table 14–1. TIMER Configuration Structure

<table>
<thead>
<tr>
<th>Structure</th>
<th>Purpose</th>
<th>See page...</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMER_Config</td>
<td>TIMER configuration structure used to setup a timer device</td>
<td>14-3</td>
</tr>
</tbody>
</table>

Table 14–2. TIMER Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
<th>See page...</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMER_close()</td>
<td>Closes a previously opened TIMER device</td>
<td>14-4</td>
</tr>
<tr>
<td>TIMER_config()</td>
<td>Sets up the TIMER register using the configuration structure</td>
<td>14-4</td>
</tr>
<tr>
<td>TIMER_configArgs()</td>
<td>Sets up the TIMER using the register values passed in</td>
<td>14-5</td>
</tr>
<tr>
<td>TIMER_getConfig()</td>
<td>Gets the TIMER configuration</td>
<td>14-5</td>
</tr>
<tr>
<td>TIMER_getEventId()</td>
<td>Obtains IRQ event ID for the timer device</td>
<td>14-6</td>
</tr>
<tr>
<td>TIMER_open()</td>
<td>Opens a TIMER device</td>
<td>14-6</td>
</tr>
<tr>
<td>TIMER_reload()</td>
<td>Reloads the TIMER</td>
<td>14-7</td>
</tr>
<tr>
<td>TIMER_reset()</td>
<td>Resets the TIMER device</td>
<td>14-7</td>
</tr>
<tr>
<td>TIMER_start()</td>
<td>Starts the TIMER device</td>
<td>14-7</td>
</tr>
<tr>
<td>TIMER_stop()</td>
<td>Stops the TIMER device</td>
<td>14-8</td>
</tr>
</tbody>
</table>
### 14.2 Configuration Structure

This section lists the structure in the TIMER module.

<table>
<thead>
<tr>
<th><strong>TIMER_Config</strong></th>
<th><em>TIMER configuration structure used to setup Timer device</em></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Structure</strong></td>
<td>TIMER_Config</td>
</tr>
<tr>
<td><strong>Members</strong></td>
<td>Uint16 tcr Control register value</td>
</tr>
<tr>
<td></td>
<td>Uint16 prd Period register value</td>
</tr>
<tr>
<td></td>
<td><strong>For C5440, C5441, and C5471 devices only:</strong></td>
</tr>
<tr>
<td></td>
<td>Uint16 tscr Timer scaler register</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td>TIMER_Config MyConfig = {</td>
</tr>
<tr>
<td></td>
<td>0x0000, /* tcr */</td>
</tr>
<tr>
<td></td>
<td>0x1000 /* prd */</td>
</tr>
<tr>
<td></td>
<td>};</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>TIMER_config(hTimer,&amp;MyConfig);</td>
</tr>
</tbody>
</table>
14.3 Functions

This section lists the functions in the TIMER module.

**TIMER_close**  
*Closes previously opened TIMER device*

**Function**

```c
void TIMER_close(
    TIMER_Handle hTimer
);
```

**Arguments**

- `hTimer`: Device handle (see TIMER_open()).

**Return Value**

None

**Description**

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

The following tasks are performed:

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

**Example**

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

**TIMER_config**  
*Sets up TIMER register using configuration structure*

**Function**

```c
void TIMER_config(
    TIMER_Handle hTimer,
    TIMER_Config *Config
);
```

**Arguments**

- `hTimer`: Device handle, (see TIMER_open()).
- `config`: Pointer to an initialized configuration structure

**Return Value**

None

**Description**

Sets up the TIMER register using the configuration structure. The values of the structure are written to the registers TCR, PRD, TIM, (see also TIMER_configArgs() and TIMER_Config.)

**Example**

```c
TIMER_Config MyConfig = {
    ...;
}:

...;

TIMER_config(hTimer, &MyConfig);
```
### TIMER_configArgs

Sets up TIMER using register values passed in

#### Function

```c
void TIMER_configArgs(
    TIMER_Handle hTimer,
    Uint16 tcr,
    For 5440, 5441, and 5471:
    Uint16 tscr,
    For all devices:
    Uint16 prd);
```

#### Arguments

- **hTimer** Device handle (see TIMER_open()).
- **tcr** Control register value
- **tscr** Secondary control register
- **prd** Period register value

#### Return Value

None

#### Description

Sets up the timer using the register values passed in. The register values are written to the timer registers. The timer control register (tcr) is written last (see also TIMER_config()).

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

#### Example

```c
TIMER_configArgs (hTimer,
    0x0010, /* tcr */
    0x1000 /* prd */
);
```

### TIMER_getConfig

Gets the TIMER configuration structure for the specified device

#### Function

```c
void TIMER_getConfig(
    TIMER_Handle hTimer,
    TIMER_Config *Config
);
```

#### Arguments

- **hTimer** Device handle, see TIMER_open()
- **Config** Pointer to a TIMER configuration structure

#### Return Value

None

#### Description

Gets the Timer configuration structure for the specified device.

#### Example

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

Obtains IRQ event ID for TIMER device

Function

Uint16 TIMER_getEventId(
    TIMER_Handle hTimer
);

Arguments

hTimer Device handle (see TIMER_open()).

Return Value

Event ID IRQ Event ID for the timer device

Description

Obtains the IRQ event ID for the timer device (see IRQ Module, Chapter 9).

Example

Uint16 TimerEventId;
TimerEventId = TIMER_getEventId(hTimer);
IRQ_enable(TimerEventId);

TIMER_open

Opens TIMER device

Function

TIMER_Handle TIMER_open(
    int DevNum,
    Uint16 Flags
);

Arguments

DevNum Device Number:
    TIMER_DEV_ANY
    TIMER_DEV0
    TIMER_DEV1

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

Return Value

Device Handle Device handle

Description

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

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

Example

TIMER_Handle hTimer;
...
hTimer = TIMER_open(TIMER_DEV0, 0);
**TIMER_reload**  
*Reloads TIMER*

**Function**  
void TIMER_reload(  
    TIMER_Handle hTimer  
);  

**Arguments**  
hTimer  
Device handle (see TIMER_open()).  

**Return Value**  
None  

**Description**  
Reloads the timer, TIM loaded with PRD and PSC loaded with TDDR value.  

**Example**  
`TIMER_reload(hTimer);`

---

**TIMER_reset**  
*Resets TIMER device*

**Function**  
void TIMER_reset(  
    TIMER_Handle hTimer  
);  

**Arguments**  
hTimer  
Device handle (see TIMER_open()).  

**Return Value**  
None  

**Description**  
Resets the timer device. Disables and clears the interrupt event and sets the timer registers to default values. If INV (–1) is specified, all timer devices are reset.  

**Example**  
`TIMER_reset(hTimer);`

---

**TIMER_start**  
*Starts TIMER device running*

**Function**  
void TIMER_start(  
    TIMER_Handle hTimer  
);  

**Arguments**  
hTimer  
Device handle (see TIMER_open()).  

**Return Value**  
None  

**Description**  
Starts the timer device running. TSS field =0.  

**Example**  
`TIMER_start(hTimer);`
### TIMER_stop

**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);
```
14.4 Macros

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

Because the TIMER peripheral typically has two independent timers in some, but not all C54x devices, the macros identify the correct timer through either the device number or the handle.

- Table 14–3 lists the TIMER macros available that use the device number as part of the register name.
- Table 14–4 lists the TIMER macros available that use a handle.

Both Table 14–3 and Table 14–4 use the following conventions:

To use the TIMER macros, include csl_timer.h and follow these restrictions:

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

For examples that are similar to the TIMER macros, see section 6.4 in the DMA chapter or section 11.4 in the McBSP chapter.
### Table 14–3. TIMER CSL Macros Using Timer Port Number

#### (a) Macros to read/write TIMER register values

<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 write 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></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>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 register, TCR, PRD, TSCR (C5440, C5441, C5471 only), or TIM.
2. REG# indicates, if applicable, a register name with the channel number (example: TCR0).
3. FIELD indicates the register field name as specified in Appendix A.
   - For REG_FSET and REG_FMK, FIELD must be a writable field.
   - For REG_FGET, the field must be a readable field.
4. regval indicates the value to write in the register (REG).
5. fieldval indicates the value to write in the field (FIELD).
### Table 14–4.  TIMER CSL Macros Using Handle

(a) Macros to read/write TIMER register values

<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>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 register, TCR, PRD, TSCR (C5440, C5441, C5471 only), or TIM.
2) FIELD indicates the register field name as specified in Appendix A.
   - For REG_FSET and REG_FMk, FIELD must be a writable field.
   - For REG_FGET, the field must be a readable field.
3) regVal indicates the value to write in the register (REG)
4) fieldVal indicates the value to write in the field (FIELD)
14.5 Configuring the TIMER Module Using CSL GUI

14.5.1 Overview

The Timer module facilitates configuration/control of the on-chip Timer. The timer module consists of a configuration manager and a resource manager. The configuration manager allows the creation of one or more configuration objects. The configuration object consists of the necessary data to set the Timer control registers. The resource manager associates a selected configuration with a timer.

Figure 14–1 illustrates the Timer sections menu on the CSL graphical user interface (GUI).

Figure 14–1. Timer Sections Menu

The TIMER includes the following two sections:

- **TIMER Configuration Manager** allows you to create configuration objects. There are no predefined configuration objects.
- **TIMER Resource Manager** allows you to select a device that will be used and to associate a configuration object with that device. Two handle objects are predefined for some devices and just one for other devices.

14.5.2 TIMER Configuration Manager

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

14.5.2.1 Creating/Inserting a configuration

There are no predefined configuration objects available.

To configure a TIMER device through the peripheral, you must insert a new configuration object.

To insert a new configuration object, right-click on the TIMER Configuration Manager and select Insert timerCfg from the drop-down menu. The configuration objects can be renamed. Their use depends on the on-chip device resources.
**Note:**

The number of configuration objects is unlimited. Several configurations can be created and you can select the right one for a specific device and change the configuration later just by selecting a new one under the TIMER Resource Manager. This feature provides you with more flexibility and reduces the time required to modify register values.

### 14.5.2.2 Deleting/Renaming an Object

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

If a configuration object is used by one of the predefined handle objects of the TIMER Resource Manager (see section 14.5.3, Timer Resource Manager), the Delete and Rename options are grayed out and non-usable. The Show Dependency option is accessible and shows which device is using the configuration object (see Figure 2–1, The CSL Tree, on page [2-3]).

### 14.5.2.3 Configuring the Object Properties

You can configure object properties through the Properties dialog box (see Figure 14–2). To access the Properties dialog box, right-click on a configuration object and select Properties. By default, the General page of the Properties dialog box is displayed.

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

- **General:** Allows you to configure the Breakpoint Emulation
- **Counter Control:** Allows you to configure the Counter configuration
- **GPIO:** General Purpose I/O pin configuration
- **Advanced Page:** Allows you to configure the Summary of the previous pages
- **This page contains the full hexadecimal register values and reflects the setting of the previous pages**
- **The full register values can be entered directly and the new options will be mirrored on the previous three pages automatically**

Figure 14–2 depicts the Properties Page dialog box.
Each Tab page is composed of several options that are set to a default value (at device reset).

The options represent the fields of the TIMER registers; the associated field name is shown in parenthesis.

14.5.3 TIMER Resource Manager

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

Figure 14–3 illustrates the DMA Resource Manager menu.

14.5.3.1 Predefined Objects

Two handle objects are predefined and each of them is associated with a supported on-chip TIMER device.

- **TIMER0** – Default handle name: hTimer0
- **TIMER1** – Default handle name: hTimer1

**Note:**
The above objects can neither be deleted nor renamed.
A configuration is enabled if at least one configuration object is defined previously in section 14.5.2.

### 14.5.3.2 Properties Page

You can generate the TIMER_config and TIMER_open CSL functions through the Properties page.

To access the Properties page, right-click on a predefined TIMER handle object and select Properties from the drop-down menu (see Figure 14–4).

The first time the properties page appears, only the Open Handle to Timer check-box can be selected. Select this to open the TIMER configuration, allowing pre-initialization.

TIMER NOTHING is used to indicate that there is no configuration object selected for this device.

To pre-initialize the TIMER channel, check the Enable Pre-Initialization box. One of the available configuration objects (see section 14.5.2, TIMER Configuration Manager) can then be selected for this channel through the Pre-initialize drop-down list.

If TIMER NOTHING is selected, no configuration object will be generated for the related TIMER handle (see section 14.5.4, C Code Generation for TIMER.)

In Figure 14–4, the Open Handle to TIMER option is checked and the handle object hTimer1 is now accessible (renaming allowed). The TIMER_open() function will be generated with hTimer1 containing the return handle address.

![Figure 14–4. Timer Properties Page With Handle Object Accessible](image-url)
14.5.4 C Code Generation for TIMER

Two C files are generated from the configuration tool:

- Header file
- Source file.

14.5.4.1 Header File

The header file includes all the csl header files of the modules and contains the TIMER handle and configuration objects defined from the configuration tool (see Example 14–1).

Example 14–1. Timer Header File

```c
extern TIMER_CONFIG timerCfg0;
extern TIMER_HANDLE hTimer1;
```

14.5.4.2 Source File

The source file includes the declaration of the handle object and the configuration structures (see Example 14–2).

Example 14–2. Timer Source File (Declaration Section)

```c
/*  Config Structures */
TIMER_CONFIG timerCfg1 = {
    0x0000,        /*  Timer Control Register   */
    0x0000         /*  Timer Period Register   */
};

/*  Handles  */
TIMER_HANDLE hTimer1;
```

The source file contains the Handle and Configuration Pre-Initialization using CSL TIMER API functions TIMER_open() and TIMER_config() (see Example 14–3). These two functions are encapsulated into a unique function, CSL_cfgInit(), which is called from your main C file.

TIMER_open() and TIMER_config() will be generated only if Open Handle to TIMER and Enable-Pre-Initialization (with timerCfg0) are, respectively, checked on the TIMER Resource Manager Properties page.
Note:
You must use TIMER_start() in your code to start the TIMER. The TIMER_Config structure generated by the CSL GUI does not start the TIMER port.

Example 14–3. Timer Source File (Body Section)

```c
void CSL_cfgInit()
{
    hTimer1 = TIMER_open(TIMER_DEV1, TIMER_OPEN_RESET);
    TIMER_config(hTimer1, &timerCfg1);
}
```
Chapter 15

UART Module

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

<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>15-5</td>
</tr>
<tr>
<td>15.3 Functions</td>
<td>15-8</td>
</tr>
<tr>
<td>15.4 Macros</td>
<td>15-15</td>
</tr>
<tr>
<td>15.5 Configuring the UART Module Using CSL GUI</td>
<td>15-21</td>
</tr>
</tbody>
</table>
15.1 Overview

The Universal Asynchronous Receiver/Transmitter (UART) controller is the key component of the serial communications subsystem of a computer. Asynchronous transmission allows data to be transmitted without a clock signal to the receiver. Instead, the sender and receiver must agree on timing parameters in advance. Special bits are added to each word that is used to synchronize the sending and receiving units.

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

1) Register-based configuration
   A register-based configuration can be performed by calling either UART_config(), UART_configArgs(), or any of the SET register field macros.

2) Parameter-based configuration (Recommended)
   A parameter-based configuration can be performed by calling UART_setup(). Compared to the register-based approach, this method provides a higher level of abstraction.

Table 15–1 lists the configuration structures and functions used with the UART module.

Table 15–1. UART APIs

<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>15-5</td>
</tr>
<tr>
<td>UART_config</td>
<td>F</td>
<td>Sets up the UART using the configuration structure</td>
<td>15-8</td>
</tr>
<tr>
<td>UART_configArgs</td>
<td>F</td>
<td>Sets up the UART using register values</td>
<td>15-8</td>
</tr>
<tr>
<td>UART_eventDisable</td>
<td>F</td>
<td>Disable UART interrupts</td>
<td>15-9</td>
</tr>
<tr>
<td>UART_eventEnable</td>
<td>F</td>
<td>Enable UART interrupts</td>
<td>15-9</td>
</tr>
<tr>
<td>UART_fgetc</td>
<td>F</td>
<td>Read a character from UART by polling</td>
<td>15-11</td>
</tr>
<tr>
<td>UART_fgets</td>
<td>F</td>
<td>This routine reads a string from the uart</td>
<td>15-12</td>
</tr>
<tr>
<td>UART_fputc</td>
<td>F</td>
<td>Write a character from UART by polling</td>
<td>15-12</td>
</tr>
<tr>
<td>UART_fputs</td>
<td>F</td>
<td>This routine writes a string from the uart</td>
<td>15-12</td>
</tr>
<tr>
<td>UART_getConfig</td>
<td>F</td>
<td>Reads the UART configuration</td>
<td>15-12</td>
</tr>
</tbody>
</table>

Note:  F = Function; S = Structure
Table 15–1. 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_read</td>
<td>F</td>
<td>Read a buffer of data from UART by polling</td>
<td>15-13</td>
</tr>
<tr>
<td>UART_setCallback</td>
<td>F</td>
<td>Plugs UART interrupt routines into UART dispatcher table</td>
<td>15-13</td>
</tr>
<tr>
<td>UART_Setup</td>
<td>S</td>
<td>UART configuration structure used to setup the UART</td>
<td>15-5</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>15-14</td>
</tr>
<tr>
<td>UART_write</td>
<td>F</td>
<td>Write a buffer of data to UART by polling</td>
<td>15-14</td>
</tr>
</tbody>
</table>

Note: F = Function; S = Structure
15.2 Configuration Structures

**UART_Config**

*Configuration Structure for UART*

**Members**

- Uint16 dll  Divisor Latch Register (low 8 bits)
- Uint16 dlm  Divisor Latch Register (high 8 bits)
- Uint16 lcr  Line Control Register
- Uint16 fcr  FIFO Control Register
- Uint16 mcr  Modem Control Register

**Description**

UART configuration structure. This structure is created and initialized, and then passed to the UART_Config() function.

**UART_Setup**

*Structure used to initialize the UART*

**Members**

- Uint16 clkInput  UART input clock frequency. Valid symbolic values are:
  - UART_CLK_INPUT_58  // Input clock = 58.9824MHz
  - UART_CLK_INPUT_117  // Input clock = 117.9684MHz
- Uint16 baud  Baud Rate (Range: 150 – 115200). Valid symbolic values are:
  - UART_BAUD_150
  - UART_BAUD_300
  - UART_BAUD_600
  - UART_BAUD_1200
  - UART_BAUD_1800
  - UART_BAUD_2000
  - UART_BAUD_2400
  - UART_BAUD_3600
  - UART_BAUD_4800
  - UART_BAUD_7200
UART_Setup

UART_BAUD_9600
UART_BAUD_19200
UART_BAUD_38400
UART_BAUD_57600
UART_BAUD_115200

Uint16 wordLength  bits per word (Range: 5,6,7,8).
Valid symbolic values are:
UART_WORD5  5 bits per word
UART_WORD6  6 bits per word
UART_WORD7  7 bits per word
UART_WORD8  8 bits per word

Uint16 stopBits  stop bits in a word (1, 1.5, and 2)
Valid symbolic values are:
UART_STOP1  1 stop bit
UART_STOP1_PLUS_HALF  1 and 1/2 stop bits
UART_STOP2  2 stop bits

Uint16 parity  parity setups
Valid symbolic values are:
UART_DISABLE_PARITY
UART_ODD_PARITY  odd parity
UART_EVEN_PARITY  even parity
UART_MARK_PARITY  mark parity
(the parity bit is always ‘1’)
UART_SPACE_PARITY  space parity
(the parity bit is always ‘0’)

Uint16 fifoControl  FIFO Control
Valid symbolic values are:
UART_FIFO_DISABLE
UART_FIFO_DMA0_TRIG01
UART_FIFO_DMA0_TRIG04
UART_FIFO_DMA0_TRIG08
UART_FIFO_DMA0_TRIG14

DMA mode0: always be 0
RCVR FIFO trigger level: There are four trigger levels for the RCVR FIFO interrupt.
TRIG01 – 1 byte
TRIG04 – 4 bytes
TRIG08 – 8 byte
TRIG14 – 14 bytes

Uint16 loopbackEnable

loopback Enable
Valid Symbolic values are:
UART_NO_LOOPBACK
UART_LOOPBACK

Description
Structure used to init the UART. After created and initialized, it is passed to the UART_init() function.
15.3 Functions

15.3.1 CSL Primary Functions

**UART_config** *Initializes the UART using the configuration structure*

**Function**
void UART_config (UART_Config *Config);

**Arguments**
Configure pointer to an initialized configuration structure (containing values for all registers that are visible to the user)

**Description**
Writes a value to initialize the UART using the configuration structure.

**Example**
UART_Config Config = {
    0x00, /* DLL */
    0x06, /* DLM – baud rate 150 */
    0x18, /* LCR – even parity, 1 stop bit, 5 bits word length */
    0x00, /* Disable FIFO */
    0x00 /* No Loop Back */
};
UART_config(&Config);

**UART_configArgs** *Sets up the UART using register values*

**Function**
void UART_configArgs(Uint16 dll, Uint16 dlm, Uint16 lcr,
                     Uint16 fcr, Uint16 mcr);

**Arguments**
dll  value to setup DLL register
dlm  value to setup DLM register
lcr  value to setup LCR register
fcr  value to setup FCR register
mcr  value to setup MCR register

**Description**
Sets up the UART using the register values passed.
**UART_eventEnable**

**Example**

```c
Uint16 tDll = 0x00 /* DLL */
Uint16 tDlm = 0x06 /* DLM */
Uint16 tLcr1 = 0x18 /* LCR */
Uint16 tFcr = 0x00 /* FCR */
Uint16 tMcr = 0x00 /* MCR */

UART_configArgs(tDll, tDlm, tLcr1, tFcr, tMcr);
```

---

**UART_eventDisable**  
Disables UART interrupts

**Function**

```c
void UART_eventDisable(Uint16 ierMask);
```

**Arguments**

ierMask can be one or a combination of the following:

- UART_RINT 0x01 // Enable rx data available interrupt
- UART_TINT 0x02 // Enable tx hold register empty interrupt
- UART_LSINT 0x04 // Enable rx line status interrupt
- UART_MSINT 0x08 // Enable modem status interrupt
- UART_ALLINT 0x0f // Enable all interrupts

**Description**

It disables the interrupt specified by the ierMask.

**Example**

```c
UART_eventDisable(UART_TINT);
```

---

**UART_eventEnable**  
Enables a UART interrupt

**Function**

```c
void UART_eventEnable (Uint16 isrMask);
```

**Arguments**

isrMask can be one or a combination of the following:

- UART_RINT 0x01 // Enable rx data available interrupt
- UART_TINT 0x02 // Enable tx hold register empty interrupt
- UART_LSINT 0x04 // Enable rx line status interrupt
- UART_MSINT 0x08 // Enable modem status interrupt
- UART_ALLINT 0x0f // Enable all interrupts
**UART_eventEnable**

**Description**
It enables the UART interrupt specified by the isrMask.

**Example**
UART_eventEnable(UART_RINT | UART_TINT);
**UART_fgetc**

*Reads UART characters*

**Function**

CSLBool UART_fgetc(int *c, Uint32 timeout);

**Arguments**

- `c` Character read from UART
- `timeout` Time out for data ready. If it is setup as 0, means there will be no time out count. The function will block forever until DR bit is set.

**Description**

Read a character from UART by polling.

**Example**

```c
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**

```c
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.
  The function will block forever if THRE bit is not set.

Description
This routine writes a character out through UART.

Example
Example
const int putchar = 'A';
CSLBool returnFlag;

ReturnFlag = UART_fputc(putchar, 0);

UART_fputs

Function
CSLBool UART_fputs(const char* pBuf, Uint32 timeout);

Arguments
- pBuf: Pointer to a buffer
- timeout: Time out for data ready.
  If it is setup as 0, means there will be no time out count.
  The function will block forever if THRE bit is not set.

Description
This routine writes a string to the uart. The NULL terminator is not written and
a newline is not added to the output.

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

UART_getConfig

Function
void UART_getConfig (UART_Config *Config);

Arguments
- Config: Pointer to an initialized configuration structure (including all registers
  that are visible to the user)

Description
Reads the UART configuration structure.

Example
UART_Config Config;
UART_getConfig(&Config);
UART_read

Reads received data

Function

CSLBool UART_read(char *pBuf, Uint16 length, Uint32 timeout);

Arguments

pbuf Pointer to a buffer
length Length of data to be received
timeout Time out for data ready.
If it is setup as 0, means there will be no time out count.
The function will block forever until DR bit is set.

Description

Receive and put the received data to the buffer pointed by pbuf.

Example

Uint16 length = 10;
char pbuf[length];
CSLBool returnFlag;
ReturnFlag = UART_read(&pbuf[0], length, 0);

UART_setCallback

 Associates a function to the UART dispatch table

Function

void UART_setCallback(UART_IsrAddr *isrAddr);

Arguments

isrAddr is a structure containing pointers to the 5 functions that will be executed when the corresponding events is enabled.

Description

It associates each function specified in the isrAddr structure to the UART dispatch table.

Example

UART_IsrAddr MyIsrAddr = {
    NULL, // Receiver line status
    UartRxIsr, // received data available
    UartTxIsr, // transmiter holding register empty
    NULL     // character time-out indication
};
UART_setCallback(&MyIsrAddr);
UART_setup

**Sets the UART based on the UART_Setup configuration structure**

**Function**
void UART_setup (UART_Setup *Params);

**Arguments**
Params Pointer to an initialized configuration structure that contains values for UART setup.

**Description**
Sets UART based on UART_Setup structure.

**Example**
UART_Setup Params = {
    UART_CLK_INPUT_58, /* input clock freq */
    UART_BAUD_115200, /* baud rate */
    UART_WORD8, /* word length */
    UART_STOP1, /* stop bits */
    UART_DISABLE_PARITY, /* parity */
    UART_FIFO_DISABLE, /* FIFO enable/disable */
    UART_FIFO_DMA_MODE0, /* DMA mode */
    UART_FIFO_TRIG01, /* FIFO trigger level */
    UART_NO_LOOPBACK, /* Loop Back enable/disable */
};

UART_setup(&Params);

UART_write

**Transmits buffers of data by polling**

**Function**
CSLBool UART_write(char *pBuf, Uint16 length, Uint32 timeout);

**Arguments**
pbuf Pointer to a data buffer
Length Length of the data buffer
timeout Time out for data ready.
    If it is setup as 0, means there will be no time out count.
    The function will block forever if THRE bit is not set.

**Description**
Transmit a buffer of data by polling.

**Example**
Uint16 length = 4;
char pbuf[4] = {0x74, 0x65, 0x73, 0x74};
CSLBool returnFlag;

ReturnFlag = UART_write(&pbuf[0],length,0);
15.4 Macros

15.4.1 General Macros

Table 15–2. UART CSL Macros

<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>(Applicable only to registers with more than one field)</td>
<td></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>Note: <em>Start with field values with most significant field positions:</em></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>UART_FMK()</td>
<td>Uint16 UART_FMK(REG, FIELD, fieldval)</td>
</tr>
</tbody>
</table>

Notes:
1) REG indicates the registers: URIER, URIIR, URIERB, URTHR, URFCR, URCSR, URMCR, URLSR, URMSR, URDLL or URDLM.
2) FIELD indicates the register field name.
3) – or REG_FSET and REG_FMK, FIELD must be a writable field.
4) – For REG_FGET, the field must be a readable field.
5) regval indicates the value to write in the register (REG)
6) fieldval indicates the value to write in the field (FIELD)
Table 15–2. UART CSL Macros (Continued)

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<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, URLCR, URMCR, URLSR, URMSR, URDLL or UrdLM.
2) FIELD indicates the register field name.
3) – or REG FSET and REG __FMK, FIELD must be a writable field.
4) – For REG FGET, the field must be a readable field.
5) regval indicates the value to write in the register (REG)
6) fieldval indicates the value to write in the field (FIELD)

15.4.2 UART Control Signal Macros

All the UART control signals are mapped through HPIGPIO pins. They are configurable through GPIOCR and GPIOSR registers. Since C54x DSP are commonly used as DCE (Data Communication Equipment), these signals are configured as following:

HD0 – DTR – Input
HD1 – RTS – Input
HD2 – CTS – Output
HD3 – DSR – Output
HD4 – DCD – Output
HD5 – RI – Output
UART_dtcOff

**Sets a CTS signal to OFF**

**Macro**
UART_dtcOff

**Arguments**
None

**Description**
Set CTS signal off.

**Example**
UART_dtcOff;

UART_ctsOff

**Sets a CTS signal to ON**

**Macro**
UART_ctsOn

**Arguments**
None

**Description**
Set CTS signal on.

**Example**
UART_ctsOn;

UART_flowCtrlInit

*Initializes the HPIGPIO registers for flow control*

**Macro**
UART_flowCtrlInit

**Arguments**
None

**Description**
Initialize HPIGPIO registers for flow control.

**Example**
UART_flowCtrlInit;

UART_isRts

*Verifies that RTS is ON*

**Macro**
UART_isRts

**Arguments**
None

**Description**
Check if RTS is on. Return RTS value.

**Example**
CSLBool rtsSignal;
rtsSignal = UART_isRts;

UART_dtcOff

**Sets a DTC signal to OFF**

**Macro**
UART_dtcOff

**Arguments**
None

**Description**
Set DTC signal off.

**Example**
UART_dtcOff;
UART_dtcOn

Sets a DTC signal to ON

Macro UART_dtcOn
Arguments None
Description Set DTC signal on.
Example UART_dtcOn;

UART_riOff

Sets an RI signal to OFF

Macro UART_riOff
Arguments None
Description Set RI signal off.
Example UART_riOff;

UART_riOn

Sets an RI signal to ON

Macro UART_riOn
Arguments None
Description Set RI signal on.
Example UART_riOn;

UART_dsrOff

Sets a DSR signal to OFF

Macro UART_dsrOff
Arguments None
Description Set DSR signal off.
Example UART_dsrOff;

UART_dsrOn

Sets a DSR signal to ON

Macro UART_dsrOn
Arguments None
Description Set DSR signal on.
Example UART_dsrOn;
<table>
<thead>
<tr>
<th>UART_isDtr</th>
<th>Verifies that DTR is ON</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macro</td>
<td>UART_isDtr</td>
</tr>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Description</td>
<td>Check if DTR is on. Return DTR value.</td>
</tr>
<tr>
<td>Example</td>
<td>CSLBool dtrSignal;</td>
</tr>
<tr>
<td></td>
<td>dtrSignal = UART_isDtr;</td>
</tr>
</tbody>
</table>
15.5 Configuring the UART Module Using CSL GUI

The Universal Asynchronous Receiver/Transmitter (UART) controller is the key component of the serial communications subsystem of a computer. Asynchronous transmission allows data to be transmitted without a clock signal to the receiver.

The configuration manager allows creation of one or more configuration objects. The configuration objects contain all of the data necessary to set the UART Control Registers.

The resource manager allows the user to choose a configuration object to use in configuring the UART.

Figure 15–1 illustrates the UART sections menu on the CSL graphical user interface (GUI)

Figure 15–1. UART Sections Menu

The UART includes the following two sections:

- **UART Configuration Manager** allows you to create configuration objects. No predefined configuration objects.
- **UART Resource Manager** allows you to select a configuration and enable that configuration within the object.

15.5.1 UART Configuration Manager

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

15.5.1.1 Inserting a Configuration Object

There is no predefined configuration object available.

To configure the UART registers, you must insert a new configuration object.

To insert a new configuration object, right-click on the UART Configuration Manager and select insert uartCfg from the drop-down menu. The configuration objects can be renamed. Their use depends upon the on-chip device resources.
Note:
The number of configuration objects is unlimited. Several configurations can be created and the user can select the right one for this application and can change the configuration later just by selecting a new one under the UART Resource Manager. The goal is to provide more flexibility and to reduce the time required to modify register values.

15.5.1.2 Deleting/Renaming a Configuration Object

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

If the Delete and Rename options are grayed out and non-usable, use the UART Resource Manager to configure the UART. The Show Dependency option is accessible and shows which device is using the configuration object (see Figure 2–1, The CSL Tree, on page 2-3).

15.5.1.3 Configuring the Object Properties

The Properties pages allow you to set the UART registers (see Figure 15–2). To access the Properties dialog box, right-click on a configuration object and select Properties. By default, the General page of the Properties dialog box is displayed.

The Properties pages allow you to set the time, date, and alarm registers of the UART. You can set the configuration options through the following pages:

- Setup: Allows you to configure the UART based on a set of parameters.
- Advanced: Allows you to configure the values of the previous pages by directly setting the UART registers.

This page contains the full 8-bit register values; the full register values can be entered directly and the new options will be mirrored on the previous page automatically.

Figure 15–2 depicts the properties page.
Each Tab page is composed of several options that are set to a default value (at device reset).

**15.5.2 UART Resource Manager**

The UART Resource Manager allows you to generate the UART_config() CSL function.

Figure 15–3 illustrates the UART Resource Manager menu on the CSL graphical user interface (GUI).

**15.5.2.1 Properties Page**

You can generate the UART_Config() CSL function through the Properties page.

To access the Properties page, right-click on the predefined UART peripheral and select Properties from the drop-down menu (see Figure 15–4).

The first time the Properties page appears, only the Enable UART Configuration check-box can be selected. Select this to enable the UART configuration.
UART_NOTHING is used to indicate that there is no configuration object selected for this serial port.

To pre-initialize the UART peripheral, check the Enable UART Configuration box. You can then select one of the available configuration objects (see section 15.5.1) for this peripheral through the pre-initialize drop-down list.

If UART_NOTHING remains selected, the UART_config() function will not be generated for the UART. (see section 15.5.3).

In the example shown in Figure 15–4, the uartCfg0 is selected, UART and the UART_config() function will be generated, thereby setting the UART to UARTfg0 configuration settings.

Figure 15–4. UART Resource Manager Properties Page

15.5.3 C Code Generation for UART Module

The two C files generated from the configuration tool are a header file and a source file.

15.5.3.1 Header File

The header file includes all the csl header files of the modules and contains the configuration objects defined from the configuration tool (see Example 15–1).

Example 15–1. UART Header File

```
extern UART_Config uartCfg0;
```
15.5.3.2 Source File

The source file includes the declaration of the handle object and the configuration structures (see Example 15–2).

Example 15–2. UART Source File (Declaration Section)

```c
/* Config Structures */
UART_Config uartCfg0 = {
    0x80, /* Divisor Latch (URDLL - LSB) */
    0x01, /* Divisor Latch (URDLL - MSB) */
    0x00, /* Line Control Register (URLCR) */
    0x00, /* FIFO Control Register (URFCR) */
    0x00 /* Modem Control Register (URMCR) */
};
```

The source file contains the Configuration Pre-Initialization using the CSL UART API function UART_config() (see Example 15–3).

This function is encapsulated into a unique function, CSL_cfgInit(), which is called from your main C file. UART_config() is generated only if Enable UART configuration (with a selected configuration other than UART_NOTHING) is checked under the UART Resource Manager Properties page.

Example 15–3. UART Source File (Body Section)

```c
void CSL_cfgInit()
{
    UART_config(&uartCfg0);
}
```
This chapter lists the configuration structure, functions, and macros available for use with the WDTIM modules.

<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 Configuration Structure</td>
<td>16-3</td>
</tr>
<tr>
<td>16.3 Functions</td>
<td>16-4</td>
</tr>
<tr>
<td>16.4 Macros</td>
<td>16-6</td>
</tr>
<tr>
<td>16.5 Configuring the WATCHDOG TIMER Module Using CSL GUI</td>
<td>16-8</td>
</tr>
</tbody>
</table>
16.1 Overview

Table 16–1 and Table 16–2 list the configuration structures and functions used with the WDTIM module.

Table 16–1. WDTIM Configuration Structure

<table>
<thead>
<tr>
<th>Structure</th>
<th>Purpose</th>
<th>See page...</th>
</tr>
</thead>
<tbody>
<tr>
<td>WDTIM_Config</td>
<td>WDTIM configuration structure used to setup a watchdog timer device</td>
<td>16-3</td>
</tr>
</tbody>
</table>

Table 16–2. WDTIM Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
<th>See page...</th>
</tr>
</thead>
<tbody>
<tr>
<td>WDTIM_config</td>
<td>Sets up the WDTIM register using the configuration structure</td>
<td>16-4</td>
</tr>
<tr>
<td>WDTIM_configArgs</td>
<td>Sets up the WDTIM using the register values passed in</td>
<td>16-4</td>
</tr>
<tr>
<td>WDTIM_getConfig</td>
<td>Reads the current register values of the watchdog timer and stores the result in the configuration structure</td>
<td>16-5</td>
</tr>
<tr>
<td>WDTIM_service</td>
<td>Writes to the watchdog key of the timer</td>
<td>16-5</td>
</tr>
<tr>
<td>WDTIM_start</td>
<td>Starts the WDTIM device running</td>
<td>16-5</td>
</tr>
</tbody>
</table>
16.2 Configuration Structure

This section lists the structure in the WDTIM module.

**WDTIM_Config**

**WDTIM configuration structure used to setup timer device**

<table>
<thead>
<tr>
<th>Structure</th>
<th>WDTIM_Config</th>
</tr>
</thead>
<tbody>
<tr>
<td>Members</td>
<td></td>
</tr>
<tr>
<td>Uint16 wdtcr</td>
<td>control</td>
</tr>
<tr>
<td>Uint16 wdtscr</td>
<td>secondary control</td>
</tr>
<tr>
<td>Uint16 wdpdrd</td>
<td>period</td>
</tr>
</tbody>
</table>

**Description**
The WDTIM configuration structure is used to setup a watchdog timer device. You create and initialize this structure then pass its address to the WDTIM_config() function. You can use literal values or the WDTIM_RMK macros to create the structure member values.

**Example**

```c
WDTIM_Config MyConfig = {
    0x0000, /* control */
    0x1000, /* secondary control */
    0x1000 /* period */
};
...
WDTIM_config(&MyConfig);
```
16.3 Functions

This section lists the functions in the WDTIM module.

**WDTIM_config**

Sets up WDTIM register using configuration structure

- **Function**
  ```c
  void WDTIM_config(
      WDTIM_Config *Config
  );
  ```

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

- **Return Value**
  - None

- **Description**
  Sets up the WDTIM register using the configuration structure. The values of the structure are written to the registers TCR, PRD, and TIM (see also `WDTIM_configArgs()` and `WDTIM_Config`).

- **Example**
  ```c
  WDTIM_Config MyConfig = {
  };
  ...
  WDTIM_config(&MyConfig);
  ```

**WDTIM_configArgs**

Sets up WDTIM using register values passed in

- **Function**
  ```c
  void WDTIM_configArgs(
      Uint16 wdtcr,
      Uint16 wdtscr,
      Uint16 wdprd
  );
  ```

- **Arguments**
  - `wdtcr` Control register value
  - `wdtscr` Secondary Control register value
  - `wdprd` Period register value

- **Return Value**
  - None

- **Description**
  Sets up the timer using the register values passed in. The register values are written to the timer registers. The timer control register (wdtcr) is written last (see also `WDTIM_config()`).

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

- **Example**
  ```c
  WDTIM_configArgs (    
      0x0010, /* wdtcr */    
      0x0000, /* wdtscr */    
      0x1000 /* wdprd */    
  );
  ```
### WDTIM_getConfig

**Function**

```c
void WDTIM_getConfig(
    WDTIM_Config *Config
);
```

**Arguments**

- `Config` Pointer to a WDTIM configuration structure

**Return Value**

None

**Description**

Gets the WDTIM configuration structure for a specified device.

**Example**

```c
WDTIM_Config MyConfig;
WDTIM_getConfig(&MyConfig);
```

### WDTIM_service

**Function**

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

**Arguments**

None

**Return Value**

None

**Description**

Services the watchdog timer by writing a sequence of A5C6h, followed by an A7Eh to the WDKEY field of the WDTSCR register, before the watchdog timer times out. This function must be called periodically to prevent a watchdog timeout.

**Example**

```c
WDTIM_service();
```

### WDTIM_start

**Function**

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

**Arguments**

None

**Return Value**

None

**Description**

Starts the timer device running. TSS field =0.

**Example**

```c
WDTIM_start();
```
16.4 Macros

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

Table 16–3 lists the available WDTIM macros.

Table 16–3 uses the following conventions:

To use the WDTIM macros, include csl_wdtim.h and follow these restrictions:

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

For examples that are similar to the WDTIM macros, see section 6.4 in the DMA chapter or section 11.4 in the McBSP chapter.
Table 16–3.  WDTIM CSL Macros Using Timer Port Number

(a) Macros to read/write WDTIM register values

<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: field_n: MSB field field_0: LSB field * 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 register, WDTCR, WDPRD, WDTSCR, or WDTIM.
2) FIELD indicates the register field name as specified in Appendix A.
   - For REG_FSET and REG_FMK, FIELD must be a writable field.
   - For REG_FGET, the field must be a readable field.
3) regval indicates the value to write in the register (REG)
4) fieldval indicates the value to write in the field (FIELD)
16.5 Configuring the WATCHDOG TIMER Module Using CSL GUI

16.5.1 Overview

The WATCHDOG TIMER module facilitates configuration/control of the on-chip WATCHDOG TIMER. The WATCHDOG TIMER module consists of a configuration manager and a resource manager. The configuration manager allows the creation of one or more configuration objects. The configuration object consists of the necessary data to set the WATCHDOG TIMER control registers. The resource manager associates a selected configuration with a timer.

Figure 16–1 illustrates the WATCHDOG TIMER sections menu on the CSL graphical user interface (GUI).

Figure 16–1. WATCHDOG TIMER Sections Menu

The WATCHDOG TIMER includes the following two sections:

- **WATCHDOG TIMER Configuration Manager**: Allows you to create configuration objects. (There are no predefined configuration objects.)

- **WATCHDOG TIMER Resource Manager**: Allows you to associate a configuration object to the Watchdog Timer. The WATCHDOG TIMER is only available in the TMS320C5440 and TMS320C5441 devices.

16.5.2 WATCHDOG TIMER Configuration Manager

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

16.5.2.1 Creating/Inserting a configuration

There are no predefined configuration objects available.

To configure a WATCHDOG TIMER device through the peripheral, you must insert a new configuration object.

To insert a new configuration object, right-click on the WATCHDOG TIMER Configuration Manager and select Insert wdtimCfg from the drop-down menu. The configuration objects can be renamed. Their use depends on the on-chip device resources.
Note:
The number of configuration objects is unlimited. Several configurations can be created and you can select the right one for a specific device and change the configuration later just by selecting a new one under the WATCHDOG TIMER Resource Manager. This feature provides you with more flexibility and reduces the time required to modify register values.

16.5.2.2 Deleting/Renaming an Object

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

If a configuration object is used by one of the predefined handle objects of the WATCHDOG TIMER Resource Manager, the Delete and Rename options are grayed out and non-usable. The Show Dependency option is accessible and shows which device is using the configuration object (see Figure 2–1, The CSL Tree, on page 2-3).

16.5.2.3 Configuring the Object Properties

You can configure object properties through the Properties dialog box (see Figure 16–2). To access the Properties dialog box, right-click on a configuration object and select Properties. By default, the General page of the Properties dialog box is displayed.

The Properties pages allow you to set the Peripheral registers related to the WATCHDOG TIMER. You can set the configuration options through the following tab pages:

- General: Allows you to configure the Breakpoint Emulation
- Counter Control: Allows you to configure the Breakpoint Emulation Counter configuration
- Advanced Page: Allows you to configure the Summary of the previous two pages
- This page contains the full hexadecimal register values and reflects the setting of the two pages
- The full register values can be entered directly and the new options will be mirrored on the previous two pages automatically

Figure 16–2 depicts the Properties Page dialog box.
Each Tab page is composed of several options that are set to a default value (at device reset).

The options represent the fields of the WATCHDOG TIMER registers; the associated field name is shown in parenthesis.

16.5.3 WATCHDOG TIMER Resource Manager

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

Figure 16–3 illustrates the WATCHDOG TIMER Resource Manager Menu.

16.5.3.1 Properties Page

You can generate the WDTIM_config() csl function through the Properties page.

To access the Properties page, right-click on a predefined TIMER handle object and select Properties from the drop-down menu (see Figure 16–4).

The first time the properties page appears, only the Enable Configuration of WATCHDOG TIMER check-box can be selected. Select this to open the WATCHDOG TIMER configuration, allowing pre-initialization.
WDTIM NOTHING is used to indicate that there is no configuration object selected for this device.

To pre-initialize the Watchdog Timer, check the Enable Configuration of WATCHDOG TIMER box. One of the available configuration objects (see section 16.5.2) can then be selected for this channel through the Pre-Initialize drop-down list.

If WDTIM NOTHING remains selected, no WDTIM_config() function call will be generated for the WATCHDOG TIMER. (See section 16.5.4.)

In Figure 16–4, the Enable Configuration of Watchdog Timer option is checked and wdtimCfg0 is now accessible. The WDTIM_config() function will be generated.

Figure 16–4. WATCHDOG TIMER Properties Page

16.5.4 C Code Generation for WATCHDOG TIMER

Two C files are generated from the configuration tool:

- Header file
- Source file.

16.5.4.1 Header File

The header file includes all the csl header files of the modules and contains the WATCHDOG TIMER configuration objects defined from the configuration tool (see Example 16–1).
Example 16–1. WATCHDOG TIMER Header File

```c
extern WDTIM_Config wdtimCfg0;
```

16.5.4.2 Source File

The source file includes the declaration of the configuration structures (see Example 16–2).

Example 16–2. WATCHDOG TIMER Source File (Declaration Section)

```c
/* Config Structures */
WDTIM_Config wdtimCfg0 = {
  0x0000,        /* Timer Control Register */
  0x0000,        /* Timer Period Register */
  0x0000        /* Timer Scale Register (TSCR) */
};
```

The source file contains the Configuration Pre-Initialization using the CSL WATCHDOG TIMER API WDTIM_config() (see Example 16–3). This function is encapsulated into a unique function, CSL_cfgInit(), which is called from your main C file.

WDTIM_config() will be generated only if Enable Configuration of WATCHDOG TIMER is checked and a configuration other than WDTIM_NOTHING is selected on the Watchdog Timer Resource Manager Properties page.

**Note:**

You must use WDTIM_start() in your code to start the WATCHDOG TIMER. The WDTIM_Config structure generated by the CSL GUI does not start the WATCHDOG TIMER port.

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

```c
void CSL_cfgInit()
{
    WDTIM_config(&wdtimCfg0);
}
```
Peripheral Registers

This appendix provides symbolic constants for the peripheral registers.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1 DMA Registers</td>
<td>A-2</td>
</tr>
<tr>
<td>A.2 EBUS Registers</td>
<td>A-16</td>
</tr>
<tr>
<td>A.3 GPIO Registers (C5440 and C5441)</td>
<td>A-22</td>
</tr>
<tr>
<td>A.4 HPI Registers</td>
<td>A-24</td>
</tr>
<tr>
<td>A.5 Multichannel BSP (McBSP) Registers</td>
<td>A-26</td>
</tr>
<tr>
<td>A.6 PLL Register (CLKMD)</td>
<td>A-47</td>
</tr>
<tr>
<td>A.7 Timer Registers</td>
<td>A-49</td>
</tr>
<tr>
<td>A.8 Watchdog Timer Registers (C5441)</td>
<td>A-52</td>
</tr>
</tbody>
</table>
A.1 DMA Registers

A.1.1 DMA Channel Priority and Enable Control Register (DMPREC)

Figure A–1. DMA Channel Priority and Enable Control Register (DMPREC)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>FREE</td>
<td></td>
<td>OFF</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ON</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>AUTOIX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>USE_DMA0</td>
<td>0</td>
<td>All DMA channels use DMGSA0, DMGDA0, DMGCR0, and DMGFR0 as their reload registers.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>USE_CHAN</td>
<td>1</td>
<td>Each DMA channel uses its local set of reload registers during autoinitialization mode.</td>
</tr>
<tr>
<td>13-8</td>
<td>DPRC</td>
<td>OF(value)</td>
<td>0-63</td>
<td>DMA channel priority control bit. Each bit specifies the priority of a DMA channel. When the bit is cleared to 0, the channel is a low priority; when the bit is set to 1, the channel is a high priority.</td>
</tr>
<tr>
<td>7-6</td>
<td>INTOSEL</td>
<td></td>
<td></td>
<td>Interrupt multiplex control bits. The INTOSEL bits control how the DMA interrupts are assigned in the interrupt vector table and IMR/IMF registers. The effects of this field on the operation are device-specific.</td>
</tr>
<tr>
<td></td>
<td>NONE</td>
<td></td>
<td>00</td>
<td>Interrupts available: Timer 1, McBSP 1 RINT/XINT</td>
</tr>
<tr>
<td></td>
<td>CH2_CH3</td>
<td></td>
<td>01</td>
<td>Interrupts available: Timer 1, DMA channel 2, DMA channel 3</td>
</tr>
<tr>
<td></td>
<td>CH0_TO_CH3</td>
<td></td>
<td>10</td>
<td>Interrupts available: DMA channel 0, DMA channel 1, DMA channel 2, DMA channel 3</td>
</tr>
</tbody>
</table>

† Only available on specific devices.

Legend: R/W-x = Read/Write-Reset value
Table A–1. DMA Channel Priority and Enable Control Register (DMPREC)
Field Values (DMA_DMPREC_field_symval) (Continued)

<table>
<thead>
<tr>
<th>Bit field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTOSEL</td>
<td>11</td>
<td>0x0</td>
<td>Reserved</td>
</tr>
<tr>
<td>CH4_CH5</td>
<td>00</td>
<td>0x0</td>
<td>Interrupts available: McBSP 0 RINT/XINT, McBSP 1 RINT/XINT, McBSP 2 RINT/XINT, DMA channel 4, DMA channel 5</td>
</tr>
<tr>
<td>CH2_TO_CH5</td>
<td>01</td>
<td>0x0</td>
<td>Interrupts available: McBSP 0 RINT/XINT, McBSP 2 RINT/XINT, DMA channel 2, DMA channel 3, DMA channel 4, DMA channel 5</td>
</tr>
<tr>
<td>CH0_TO_CH5</td>
<td>10</td>
<td>0x0</td>
<td>Interrupts available: McBSP 0 RINT/XINT, DMA channel 0, DMA channel 1, DMA channel 2, DMA channel 3, DMA channel 4, DMA channel 5</td>
</tr>
<tr>
<td>DE</td>
<td>0-63</td>
<td>0-63</td>
<td>DMA channel enable bit. Each bit enables a DMA channel. When the bit is cleared to 0, the channel is disabled; when the bit is set to 1, the channel is enabled.</td>
</tr>
</tbody>
</table>

A.1.2 DMA Channel n Sync Select and Frame Count Register (DMSFCn)

Figure A–2. DMA Channel n Sync Select and Frame Count Register (DMSFCn)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-12</td>
<td>DSYN</td>
<td></td>
<td></td>
<td>DMA sync event. Specifies which sync event is used to initiate DMA transfers for the corresponding DMA channel. The effects of this field on the operation are device-specific.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NONE</td>
<td>0000</td>
<td>No sync event (nonsynchronization operation)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>REVT0</td>
<td>0001</td>
<td>McBSP 0 receive event (REVT0)</td>
</tr>
</tbody>
</table>

Legend: R/W-x = Read/Write-Reset value
### Table A–2. DMA Channel n Sync Select and Frame Count Register (DMSFCn)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>XEV0</td>
<td>0010</td>
<td>McBSP 0 transmit event (XEV0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0011</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0100</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DSYN</td>
<td>REV1</td>
<td>0101</td>
<td>McBSP 1 receive event (REV1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>XEV1</td>
<td>0110</td>
<td>McBSP 1 transmit event (XEV1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0111-1100</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>INT0</td>
<td>1101</td>
<td>Timer 0 interrupt event</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INT3</td>
<td>1110</td>
<td>External interrupt 3 event</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INT1</td>
<td>1111</td>
<td>Timer 1 interrupt event</td>
<td></td>
</tr>
</tbody>
</table>

**For C5409, C5409A, and C5471:**

<table>
<thead>
<tr>
<th>NONE</th>
<th>0000</th>
<th>No sync event (nonsynchronization operation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>REV0</td>
<td>0001</td>
<td>McBSP 0 receive event (REV0)</td>
</tr>
<tr>
<td>XEV0</td>
<td>0010</td>
<td>McBSP 0 transmit event (XEV0)</td>
</tr>
<tr>
<td>REV2</td>
<td>0011</td>
<td>McBSP 2 receive event (REV2)</td>
</tr>
<tr>
<td>XEV2</td>
<td>0100</td>
<td>McBSP 2 transmit event (XEV2)</td>
</tr>
<tr>
<td>REV1</td>
<td>0101</td>
<td>McBSP 1 receive event (REV1)</td>
</tr>
<tr>
<td>XEV1</td>
<td>0110</td>
<td>McBSP 1 transmit event (XEV1)</td>
</tr>
<tr>
<td></td>
<td>0111-1100</td>
<td>Reserved</td>
</tr>
<tr>
<td>INT0</td>
<td>1101</td>
<td>Timer interrupt event</td>
</tr>
<tr>
<td>INT3</td>
<td>1110</td>
<td>External interrupt 3 event</td>
</tr>
</tbody>
</table>

**For C5410, C5410A, and C5416:**

<table>
<thead>
<tr>
<th>NONE</th>
<th>0000</th>
<th>No sync event (nonsynchronization operation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>REV0</td>
<td>0001</td>
<td>McBSP 0 receive event (REV0)</td>
</tr>
<tr>
<td>XEV0</td>
<td>0010</td>
<td>McBSP 0 transmit event (XEV0)</td>
</tr>
<tr>
<td>REV2</td>
<td>0011</td>
<td>McBSP 2 receive event (REV2)</td>
</tr>
<tr>
<td>XEV2</td>
<td>0100</td>
<td>McBSP 2 transmit event (XEV2)</td>
</tr>
<tr>
<td>REV1</td>
<td>0101</td>
<td>McBSP 1 receive event (REV1)</td>
</tr>
<tr>
<td>XEV1</td>
<td>0110</td>
<td>McBSP 1 transmit event (XEV1)</td>
</tr>
<tr>
<td>REVTA0</td>
<td>0111</td>
<td>McBSP 0 receive event — ABIS mode (REVTA0)</td>
</tr>
<tr>
<td>XEVTA0</td>
<td>1000</td>
<td>McBSP 0 transmit event — ABIS mode (XEVTA0)</td>
</tr>
</tbody>
</table>
### Table A–2. DMA Channel n Sync Select and Frame Count Register (DMSFCn) Field Values (DMA_DMSFC_field_symval) (Continued)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REVTA2</td>
<td></td>
<td>1001</td>
<td>McBSP 2 receive event — ABIS mode (REVTA2)</td>
<td></td>
</tr>
<tr>
<td>XEVTA2</td>
<td></td>
<td>1010</td>
<td>McBSP 2 transmit event — ABIS mode (XEVTA2)</td>
<td></td>
</tr>
<tr>
<td>DSYN</td>
<td>REVTA1</td>
<td>1011</td>
<td>McBSP 1 receive event — ABIS mode (REVTA1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>XEVTA1</td>
<td>1100</td>
<td>McBSP 1 transmit event — ABIS mode (XEVTA1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TINT0</td>
<td>1101</td>
<td>Timer interrupt event</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INT3</td>
<td>1110</td>
<td>External interrupt 3 event</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1111</td>
<td>Reserved</td>
<td></td>
</tr>
</tbody>
</table>

**For C5420 and C5421:**

- NONE 0000 No sync event (nonsynchronization operation)
- REV0 0001 McBSP 0 receive event (REV0)
- XEV0 0010 McBSP 0 transmit event (XEV0)
- REV2 0011 McBSP 2 receive event (REV2)
- XEV2 0100 McBSP 2 transmit event (XEV2)
- REV1 0101 McBSP 1 receive event (REV1)
- XEV1 0110 McBSP 1 transmit event (XEV1)
- FIFO_REVT 0111 FIFO receive buffer not empty event
- FIFO_XEVT 1000 FIFO transmit buffer not full event
- 1001-1111 Reserved

**For C5440 and C5441:**

- NONE 0000 No sync event (nonsynchronization operation)
- REV0 0001 McBSP 0 receive event (REV0)
- XEV0 0010 McBSP 0 transmit event (XEV0)
- REV2 0011 McBSP 2 receive event (REV2)
- XEV2 0100 McBSP 2 transmit event (XEV2)
- REV1 0101 McBSP 1 receive event (REV1)
- XEV1 0110 McBSP 1 transmit event (XEV1)
- 0111-1100 Reserved

**For C54CST, 5404, and 5407:**

- NONE 0000 No sync event (nonsynchronization operation)
- REV0 0001 McBSP 0 receive event (REV0)
### DMA Registers

#### Table A–2. DMA Channel n Sync Select and Frame Count Register (DMSFCn) Field Values (DMA_DMSFC_field_symval) (Continued)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XEVT0</td>
<td>0010</td>
<td>McBSP 0 transmit event (XEVT0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REV2</td>
<td>0011</td>
<td>McBSP 2 receive event (REV2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XEVT2</td>
<td>0100</td>
<td>McBSP 2 transmit event (XEVT2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REV1</td>
<td>0101</td>
<td>McBSP 1 receive event (REV1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XEVT1</td>
<td>0110</td>
<td>McBSP 1 transmit event (XEVT1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0611-1100</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000-1101</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UART</td>
<td>0111</td>
<td>UART interrupt event</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TINT0</td>
<td>1101</td>
<td>Timer interrupt event</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TINT1</td>
<td>111</td>
<td>Timer1 interrupt event</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TINT3</td>
<td>1110</td>
<td>External interrupt 3 event</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### A.1.3 DMA Channel n Transfer Mode Control Register (DMMCRn)

#### Figure A–3. DMA Channel n Transfer Mode Control Register (DMMCRn)

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTOINIT</td>
<td>DINM</td>
<td>IMOD</td>
<td>CTMOD</td>
<td>SLAXS†</td>
<td>SIND</td>
<td></td>
</tr>
<tr>
<td>R/W-0</td>
<td>R/W-0</td>
<td>R/W-0</td>
<td>R/W-0</td>
<td>R/W-0</td>
<td>R/W-0</td>
<td></td>
</tr>
</tbody>
</table>

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

**Legend:**  
R/W-x = Read/Write-Reset value
### Table A–3. DMA Channel n Transfer Mode Control Register (DMMCRn) Field Values (DMA_DMMCR_field_symval)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>AUTOINIT</td>
<td></td>
<td></td>
<td>DMA autoinitialization mode enable bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>OFF</td>
<td>0 Autoinitialization is disabled.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ON</td>
<td>1 Autoinitialization is enabled.</td>
</tr>
<tr>
<td>14</td>
<td>DINM</td>
<td></td>
<td></td>
<td>DMA interrupt generation mask bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>OFF</td>
<td>0 No interrupt is generated</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ON</td>
<td>1 Interrupt is generated based on IMOD bit</td>
</tr>
<tr>
<td>13</td>
<td>IMOD</td>
<td></td>
<td></td>
<td>DMA interrupt generation mode bit operates in conjunction with CTMOD bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>FULL_ONLY</td>
<td>0 Interrupt at buffer full only.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>HALF_AND_FULL</td>
<td>1 Interrupt at half full buffer and buffer full.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>BLOCK_ONLY</td>
<td>0 Interrupt at completion of block transfer.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>FRAME_AND_BLOCK</td>
<td>1 Interrupt at end of frame and end of block.</td>
</tr>
<tr>
<td>12</td>
<td>CTMOD</td>
<td></td>
<td>MULTIFRAME</td>
<td>0 Multiframe mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ABU</td>
<td>1 ABU mode</td>
</tr>
<tr>
<td>11</td>
<td>SLAXS</td>
<td></td>
<td>OFF</td>
<td>0 No external access</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ON</td>
<td>1 External access</td>
</tr>
<tr>
<td>10-8</td>
<td>SIND</td>
<td></td>
<td>NOMOD</td>
<td>000 No modification</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>POSTINC</td>
<td>001 Postincrement</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>POSTDEC</td>
<td>010 Postdecrement</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DMIDX0</td>
<td>011 Postincrement with index offset (DMIDX0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DMIDX1</td>
<td>100 Postincrement with index offset (DMIDX1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DMFRI0</td>
<td>101 Postincrement with index offset (DMIDX0 and DMFRI0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DMFRI1</td>
<td>110 Postincrement with index offset (DMIDX1 and DMFRI1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>111</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
### Table A–3. DMA Channel n Transfer Mode Control Register (DMMCRn) Field Values (DMA_DMMCR_field_symval) (Continued)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symvalfield</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-6</td>
<td>DMS</td>
<td></td>
<td></td>
<td>DMA source address space select bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PROGRAM</td>
<td>00</td>
<td>Program space</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DATA</td>
<td>01</td>
<td>Data space</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IO</td>
<td>10</td>
<td>I/O space</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td>Reserved</td>
</tr>
<tr>
<td>5</td>
<td>DLAXS</td>
<td></td>
<td></td>
<td>For devices with DMA extended data memory: DMA destination space select bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OFF</td>
<td>0</td>
<td>No external access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ON</td>
<td>1</td>
<td>External access</td>
</tr>
<tr>
<td>4-2</td>
<td>DIND</td>
<td></td>
<td></td>
<td>DMA destination address transfer index mode bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NOMOD</td>
<td>000</td>
<td>No modification</td>
</tr>
<tr>
<td></td>
<td></td>
<td>POSTINC</td>
<td>001</td>
<td>Postincrement</td>
</tr>
<tr>
<td></td>
<td></td>
<td>POSTDEC</td>
<td>010</td>
<td>Postdecrement</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DMIDX0</td>
<td>011</td>
<td>Postincrement with index offset (DMIDX0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DMIDX1</td>
<td>100</td>
<td>Postincrement with index offset (DMIDX1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DMFRI0</td>
<td>101</td>
<td>Postincrement with index offset (DMIDX0 and DMFRI0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DMFRI1</td>
<td>110</td>
<td>Postincrement with index offset (DMIDX1 and DMFRI1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>111</td>
<td>Reserved</td>
</tr>
<tr>
<td>1-0</td>
<td>DMD</td>
<td></td>
<td></td>
<td>DMA destination address space select bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PROGRAM</td>
<td>00</td>
<td>Program space</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DATA</td>
<td>01</td>
<td>Data space</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IO</td>
<td>10</td>
<td>I/O space</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
A.1.4 DMA Channel n Source Address Register (DMSRCn)

Figure A–4. DMA Channel n Source Address Register (DMSRCn)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-0</td>
<td>SRC</td>
<td>OF(value)</td>
<td>0-FFFFh</td>
<td>Specifies the 16 least-significant bits of the extended address for the source location. The source address register is initialized prior to starting the DMA transfer in software, and updated automatically during transfers by the DMA controller.</td>
</tr>
</tbody>
</table>

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

Table A–4. DMA Channel n Source Address Register (DMSRCn) Field Values (DMA_DMSRC_field_symval)

A.1.5 DMA Global Source Address Reload Register (DMGSA)

Figure A–5. DMA Global Source Address Reload Register (DMGSA)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-0</td>
<td>GSA</td>
<td>OF(value)</td>
<td>0-FFFFh</td>
<td>A 16-bit source address used to reload DMSRCn.</td>
</tr>
</tbody>
</table>

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

Table A–5. DMA Global Source Address Reload Register (DMGSA) Field Values (DMA_DMGSA_field_symval)

A.1.6 DMA Source Program Page Address Register (DMSRCP)

Figure A–6. DMA Source Program Page Address Register (DMSRCP)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-0</td>
<td>reserved</td>
<td>7-6</td>
<td>0</td>
<td>Source Program Page Address (PAGE)</td>
</tr>
</tbody>
</table>

Legend: R/W-x = Read/Write-Reset value
Table A–6. DMA Source Program Page Address Register (DMSRCP) Field Values
(DMA_DMSRCP_field_symval)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-7</td>
<td>reserved</td>
<td></td>
<td></td>
<td>Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.</td>
</tr>
<tr>
<td>6-0</td>
<td>PAGE</td>
<td>OF(value)</td>
<td>0-127</td>
<td>Specifies the 7 most-significant bits of the extended program page address for the source location.</td>
</tr>
</tbody>
</table>

A.1.7 DMA Channel n Destination Address Register (DMDSTn)

Figure A–7. DMA Channel n Destination Address Register (DMDSTn)

<table>
<thead>
<tr>
<th>15</th>
<th></th>
<th></th>
<th></th>
<th>Destination Address (DST)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>R/W-0</td>
</tr>
</tbody>
</table>

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

Table A–7. DMA Channel n Destination Address Register (DMDSTn) Field Values
(DMA_DMDST_field_symval)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-0</td>
<td>DST</td>
<td>OF(value)</td>
<td>0-FFFFh</td>
<td>Specifies the 16 least-significant bits of the extended address for the destination location. The destination address register is initialized prior to starting the DMA transfer in software, and updated automatically during transfers by the DMA controller.</td>
</tr>
</tbody>
</table>

A.1.8 DMA Global Destination Address Reload Register (DMGDA)

Figure A–8. DMA Global Destination Address Reload Register (DMGDA)

<table>
<thead>
<tr>
<th>15</th>
<th></th>
<th></th>
<th></th>
<th>Global Destination Address (GDA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>R/W-0</td>
</tr>
</tbody>
</table>

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

Table A–8. DMA Global Destination Address Reload Register (DMGDA) Field Values
(DMA_DMGDA_field_symval)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-0</td>
<td>GDA</td>
<td>OF(value)</td>
<td>0-FFFFh</td>
<td>A 16-bit destination address used to reload DMDSTn.</td>
</tr>
</tbody>
</table>
A.1.9 DMA Destination Program Page Address Register (DMDSTP)

Figure A–9. DMA Destination Program Page Address Register (DMDSTP)

![DMA Destination Program Page Address Register (DMDSTP)](image)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15–7</td>
<td>reserved</td>
<td></td>
<td></td>
<td>Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.</td>
</tr>
<tr>
<td>6–0</td>
<td>PAGE</td>
<td>OF(value)</td>
<td>0-127</td>
<td>Specifies the 7 most-significant bits of the extended program page address for the destination location.</td>
</tr>
</tbody>
</table>

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

A.1.10 DMA Channel n Element Count Register (DMCTRn)

Figure A–10. DMA Channel n Element Count Register (DMCTRn)

![DMA Channel n Element Count Register (DMCTRn)](image)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15–0</td>
<td>ELECNT</td>
<td>OF(value)</td>
<td>0-FFFFh</td>
<td>A 16-bit element counter that keeps track of the number of DMA transfers to be performed. The element count register should be initialized to 1 less than the desired number of element transfers.</td>
</tr>
</tbody>
</table>

Table A–10. DMA Channel n Element Count Register (DMCTRn) Field Values (DMA_DMCTR_field_symval)
A.1.11 DMA Global Element Count Reload Register (DMGCR)

Figure A–11. DMA Global Element Count Reload Register (DMGCR)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-0</td>
<td>ELECNT</td>
<td>OF(value)</td>
<td>0-FFFFh</td>
<td>A 16-bit unsigned element count value used to reload DMCTR.</td>
</tr>
</tbody>
</table>

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

Table A–11. DMA Global Element Count Reload Register (DMGCR) Field Values (DMA_DMGCR_field_symval)

A.1.12 DMA Global Frame Count Reload Register (DMGFR)

Figure A–12. DMA Global Frame Count Reload Register (DMGFR)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-8</td>
<td>reserved</td>
<td></td>
<td></td>
<td>Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.</td>
</tr>
<tr>
<td>7-0</td>
<td>FRAMECNT</td>
<td>OF(value)</td>
<td>0-FFFFh</td>
<td>An 8-bit unsigned frame count value used to reload the Frame Count field of DMSFCn.</td>
</tr>
</tbody>
</table>

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

Table A–12. DMA Global Frame Count Reload Register (DMGFR) Field Values (DMA_DMGFR_field_symval)
A.1.13 DMA Element Address Index Register 0 (DMIDX0)

Figure A–13. DMA Element Address Index Register 0 (DMIDX0)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-0</td>
<td>ELEIDX</td>
<td>OF(value)</td>
<td>0-FFFFh</td>
<td>A 16-bit unsigned index value used to modify the source or</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>destination address following the transfer of each element.</td>
</tr>
</tbody>
</table>

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

A.1.14 DMA Element Address Index Register 1 (DMIDX1)

Figure A–14. DMA Element Address Index Register 1 (DMIDX1)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-0</td>
<td>ELEIDX</td>
<td>OF(value)</td>
<td>0-FFFFh</td>
<td>A 16-bit unsigned index value used to modify the source or</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>destination address following the transfer of each element.</td>
</tr>
</tbody>
</table>
A.1.15 DMA Frame Address Index Register 0 (DMFRI0)

Figure A–15. DMA Frame Address Index Register 0 (DMFRI0)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-0</td>
<td>FRAMEIDX</td>
<td>OF(value)</td>
<td>0-FFFFh</td>
<td>A 16-bit unsigned index value used to modify the source or destination address following the completion of blocks (or frames) of element transfers. When both element and frame indexes are used, the address is modified by the element index after each transfer and then modified by the frame index at the end of each frame.</td>
</tr>
</tbody>
</table>

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

A.1.16 DMA Frame Address Index Register 1 (DMFRI1)

Figure A–16. DMA Frame Address Index Register 1 (DMFRI1)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-0</td>
<td>FRAMEIDX</td>
<td>OF(value)</td>
<td>0-FFFFh</td>
<td>A 16-bit unsigned index value used to modify the source or destination address following the completion of blocks (or frames) of element transfers. When both element and frame indexes are used, the address is modified by the element index after each transfer and then modified by the frame index at the end of each frame.</td>
</tr>
</tbody>
</table>

Legend:  R/W-x = Read/Write-Reset value
### A.1.17 DMA Global Extended Source Data Page Register (DMSRCDP)

Figure A–17. DMA Global Extended Source Data Page Register (DMSRCDP)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-7</td>
<td>reserved</td>
<td></td>
<td></td>
<td>Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.</td>
</tr>
<tr>
<td>6-0</td>
<td>DMSRCDP</td>
<td>OF(value)</td>
<td>0-127</td>
<td>Specifies 1 of the 128 extended source data pages.</td>
</tr>
</tbody>
</table>

### A.1.18 DMA Global Extended Destination Data Page Register (DMDSTDP)

Figure A–18. DMA Global Extended Destination Data Page Register (DMDSTDP)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-7</td>
<td>reserved</td>
<td></td>
<td></td>
<td>Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.</td>
</tr>
<tr>
<td>6-0</td>
<td>DMDSTDP</td>
<td>OF(value)</td>
<td>0-127</td>
<td>Specifies 1 of the 128 extended destination data pages.</td>
</tr>
</tbody>
</table>
A.2  EBUS Registers

A.2.1  Software Wait-State Register (SWWSR)

Figure A–19. Software Wait-State Register (SWWSR)-All devices except C5440 and C5441

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>XPA†</td>
<td></td>
<td></td>
<td>For devices with extended program memory: Extended program address control bit. Selects the address ranges selected by the program fields.</td>
</tr>
<tr>
<td></td>
<td>ADDRLO</td>
<td>0</td>
<td>Address range: xx0000 - xxFFFFh</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ADDREXT</td>
<td>1</td>
<td>Address range: 000000h-7FFFFF</td>
<td></td>
</tr>
<tr>
<td>14-12</td>
<td>IO</td>
<td>OF(value)</td>
<td>0-7</td>
<td>The value corresponds to the number of wait states for I/O space 0000-FFFFh.</td>
</tr>
<tr>
<td>11-9</td>
<td>DATAHI</td>
<td>OF(value)</td>
<td>0-7</td>
<td>The value corresponds to the number of wait states for data space 8000-FFFFh.</td>
</tr>
<tr>
<td>8-6</td>
<td>DATALO</td>
<td>OF(value)</td>
<td>0-7</td>
<td>The value corresponds to the number of wait states for data space 0000-7FFFh.</td>
</tr>
<tr>
<td>5-3</td>
<td>PROGHI</td>
<td>OF(value)</td>
<td>0-7</td>
<td>The value corresponds to the number of wait states for program space 8000-FFFFh.</td>
</tr>
<tr>
<td>2-0</td>
<td>PROGLO</td>
<td>OF(value)</td>
<td>0-7</td>
<td>The value corresponds to the number of wait states for program space 0000-7FFFh.</td>
</tr>
</tbody>
</table>
A.2.2 Software Wait-State Control Register (SWCR)

Figure A–20. Software Wait-State Control Register (SWCR)-All devices except C5440 and C5441

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-1</td>
<td>reserved</td>
<td></td>
<td></td>
<td>Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.</td>
</tr>
<tr>
<td>0</td>
<td>SWSM</td>
<td></td>
<td></td>
<td>Software wait-state multiplier bit.</td>
</tr>
<tr>
<td></td>
<td>NOMULT</td>
<td>0</td>
<td>NOMULT</td>
<td>The wait states specified in SWWSR are unchanged (not multiplied).</td>
</tr>
<tr>
<td></td>
<td>MULTBY2</td>
<td>1</td>
<td>MULTBY2</td>
<td>The wait states specified in SWWSR are multiplied by 2, extending the maximum number of wait states from 7 to 14.</td>
</tr>
</tbody>
</table>

A.2.3 Bank-Switching Control Register (BSCR)

Figure A–21. Bank-Switching Control Register (BSCR) — C5401, C5402, C5409, C5420, C5421, and 5471

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>BNKCMP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IPIRQ†</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>R/W-1111b</td>
<td>R/W-1</td>
<td>R/W-0</td>
<td>R/W-0</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>HBH†</td>
<td>BH</td>
<td>EXIO</td>
<td>R/W-0</td>
</tr>
</tbody>
</table>

†HBH, IPIRQ, and EXIO bits only on selected devices.

Legend:  \( R/W-x = \text{Read/Write-Reset value} \)
### Table A–21. Bank-Switching Control Register (BSCR) Field Values — C5401, C5402, C5409, and C5420, and C5471 (EBUS_BSCR_field_symval)

<table>
<thead>
<tr>
<th>Bit field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-12 BNKCMP</td>
<td></td>
<td>64K</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0001-0111</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td>32K</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1001-1011</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16K</td>
<td>1100</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1101</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8K</td>
<td>1110</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4K</td>
<td>1111</td>
</tr>
<tr>
<td>11 PSDS</td>
<td></td>
<td>NOEXCY</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INSCY</td>
<td>1</td>
</tr>
<tr>
<td>10-9 reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 IPIRQ</td>
<td></td>
<td>CLR</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTR</td>
<td>1</td>
</tr>
<tr>
<td>7-3 reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 HBH</td>
<td></td>
<td>DISABLE</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ENABLE</td>
<td>1</td>
</tr>
</tbody>
</table>
## Table A–21. Bank-Switching Control Register (BSCR) Field Values — C5401, C5402, C5409, and C5420, and C5471 (EBUS_BSCR_field_symval) (Continued)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BH</td>
<td></td>
<td></td>
<td></td>
<td>For C5420: Data bus holder enable bit.</td>
</tr>
<tr>
<td></td>
<td>DISABLE</td>
<td>0</td>
<td>The data bus holder is disabled.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ENABLE</td>
<td>1</td>
<td>The data bus holder is enabled. When not driven, the data bus, PPD(15-0), is held in the previous logic level.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>BH</td>
<td></td>
<td></td>
<td>For C5401, C5402, C5409, and 5416: Bus holder enable bit.</td>
</tr>
<tr>
<td></td>
<td>DISABLE</td>
<td>0</td>
<td>The bus holder is disabled. When HPI16 pin is set to a logic high, address bus holder is enabled.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ENABLE</td>
<td>1</td>
<td>The data bus holder is enabled. When not driven, the data bus, D(15-0), is held in the previous logic level. When HPI16 pin is set to a logic high, address bus holder is enabled.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>EXIO</td>
<td></td>
<td></td>
<td>Hot any call on C5420 External bus interface off enable bit controls the external-bus-off function.</td>
</tr>
<tr>
<td></td>
<td>NORMAL</td>
<td>0</td>
<td>The external-bus-off function is disabled.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INACTIF</td>
<td>1</td>
<td>The external-bus-off function is enabled. The address bus, data bus, and control signals become inactive after completing the current bus cycle. The DROM, MP/MC, and OVLY bits in PMST and the HM bit in ST1 cannot be modified.</td>
<td></td>
</tr>
</tbody>
</table>

## Figure A–22. Bank-Switching Control Register (BSCR) — C5410, C5410A, and C5416

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSEC</td>
<td>DIVFCT</td>
<td>IACK</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>R/W-1</td>
<td>R/W-11b</td>
<td>R/W-0</td>
<td>R/W-0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>reserved</td>
<td>HBH†</td>
<td>BH†</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>R/W-0</td>
<td>R/W-0</td>
<td>R/W-0</td>
<td>R/W-0</td>
<td></td>
</tr>
</tbody>
</table>

†BH and HBH bits only on selected devices.

Legend: R/W-x = Read/Write-Reset value
### Table A–22. Bank-Switching Control Register (BSCR) Field Values — C5410, and C5416 (EBUS_BSCR_field_symval)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>CONSEC</td>
<td></td>
<td>0</td>
<td>Consecutive bank switching bit specifies the bank-switching mode. This bit is cleared if fast access is desired for continuous memory reads (that is, no starting and trailing cycles between read cycles).</td>
</tr>
<tr>
<td></td>
<td>32KFASTREAD</td>
<td>0</td>
<td>Bank-switching on 32K bank boundaries only.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EXTMEM</td>
<td>1</td>
<td>Consecutive bank switches on external memory reads. Each read cycle consists of 3 cycles: starting, read, and trailing.</td>
<td></td>
</tr>
<tr>
<td>14-13</td>
<td>DIVFCT</td>
<td></td>
<td></td>
<td>CLKOUT output divide factor. The CLKOUT output is driven by an on-chip source having a frequency equal to 1/(DIVFCT + 1) of the DSP clock.</td>
</tr>
<tr>
<td></td>
<td>ZERO</td>
<td>00</td>
<td>CLKOUT is not divided.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CLKBYTWO</td>
<td>01</td>
<td>CLKOUT is divided by 2 from the DSP clock.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CLKBYTHREE</td>
<td>10</td>
<td>CLKOUT is divided by 3 from the DSP clock.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CLKBYFOUR</td>
<td>11</td>
<td>CLKOUT is divided by 4 from the DSP clock.</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>IACK</td>
<td></td>
<td>0</td>
<td>IACK signal output off enable bit controls the IACK signal output function.</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>0</td>
<td>IACK signal output off function is disabled.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OFF</td>
<td>1</td>
<td>IACK signal output off function is enabled.</td>
<td></td>
</tr>
<tr>
<td>11-3</td>
<td>reserved</td>
<td></td>
<td></td>
<td>Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.</td>
</tr>
<tr>
<td>2</td>
<td>HBH</td>
<td></td>
<td></td>
<td>For C5416: HPI data bus holder enable bit.</td>
</tr>
<tr>
<td></td>
<td>DISABLE</td>
<td>0</td>
<td>The HPI data bus holder is disabled. When HPI16 pin is set to a logic high, HPI data bus holder is enabled.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ENABLE</td>
<td>1</td>
<td>The HPI data bus holder is enabled. When not driven, the HPI data bus, HD(7-0), is held in the previous logic level.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>BH</td>
<td></td>
<td></td>
<td>For C5416: Bus holder enable bit.</td>
</tr>
<tr>
<td></td>
<td>DISABLE</td>
<td>0</td>
<td>The bus holder is disabled. When HPI16 pin is set to a logic high, address bus holder is enabled.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ENABLE</td>
<td>1</td>
<td>The data bus holder is enabled. When not driven, the data bus, D(15-0), is held in the previous logic level. When HPI16 pin is set to a logic high, address bus holder is enabled.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>reserved</td>
<td></td>
<td></td>
<td>Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.</td>
</tr>
</tbody>
</table>
Figure A–23. Bank-Switching Control Register (BSCR) — C5440 and C5441

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-3</td>
<td>reserved</td>
<td></td>
<td></td>
<td>Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.</td>
</tr>
<tr>
<td>2</td>
<td>BHD</td>
<td></td>
<td>DISABLE</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ENABLE</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>BHA</td>
<td></td>
<td>DISABLE</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ENABLE</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>reserved</td>
<td></td>
<td></td>
<td>Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.</td>
</tr>
</tbody>
</table>
A.3 GPIO Registers (C5440 and C5441)

A.3.1 General Purpose I/O Register (GPIO)

Figure A–24. General Purpose I/O Register (GPIO)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>TOUT</td>
<td></td>
<td></td>
<td>Timer output enable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ENABLE</td>
<td>1</td>
<td>Enables the timer output</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DISABLE</td>
<td>0</td>
<td>Disables the timer output</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.</td>
</tr>
<tr>
<td>11</td>
<td>DIR3</td>
<td></td>
<td></td>
<td>GPIO pin direction 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INPUT</td>
<td>0</td>
<td>GPIO pin 3 is used as input</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OUTPUT</td>
<td>1</td>
<td>GPIO pin 3 is used as output</td>
</tr>
<tr>
<td>10</td>
<td>DIR2</td>
<td></td>
<td></td>
<td>GPIO pin direction 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INPUT</td>
<td>0</td>
<td>GPIO pin 2 is used as input</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OUTPUT</td>
<td>1</td>
<td>GPIO pin 2 is used as output</td>
</tr>
<tr>
<td>9</td>
<td>DIR1</td>
<td></td>
<td></td>
<td>GPIO pin direction 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INPUT</td>
<td>0</td>
<td>GPIO pin 1 is used as input</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OUTPUT</td>
<td>1</td>
<td>GPIO pin 1 is used as output</td>
</tr>
<tr>
<td>8</td>
<td>DIR0</td>
<td></td>
<td></td>
<td>GPIO pin direction 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INPUT</td>
<td>0</td>
<td>GPIO pin 0 is used as input</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OUTPUT</td>
<td>1</td>
<td>GPIO pin 0 is used as output</td>
</tr>
</tbody>
</table>

† Only available on devices with a second on-chip timer.

Legend:  R/W-\(x\) = Read/Write-Reset value

Table A–24. General Purpose I/O Register (GPIO) Field Values (GPIO_GPIO_field_symval)
### General Purpose I/O Register (GPIO) Field Values (GPIO_GPIO_field_symval) (Continued)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-4</td>
<td>reserved</td>
<td></td>
<td>0</td>
<td>Reserved. The reserved but location is always read as zero. A value written to this field has no effect.</td>
</tr>
<tr>
<td>3</td>
<td>DAT3</td>
<td></td>
<td>0</td>
<td>GPIO data bit 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LOW</td>
<td>0</td>
<td>GPIO0 is driven with a 0 (DIR3 = 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>GPIO0 is read as 0 (DIR3 = 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HI</td>
<td>1</td>
<td>GPIO0 is driven with a 1 (DIR3 = 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>GPIO0 is read as 1 (DIR3 = 0)</td>
</tr>
<tr>
<td>2</td>
<td>DAT2</td>
<td></td>
<td>0</td>
<td>GPIO data bit 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LOW</td>
<td>0</td>
<td>GPIO0 is driven with a 0 (DIR2 = 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>GPIO0 is read as 0 (DIR2 = 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HI</td>
<td>1</td>
<td>GPIO0 is driven with a 1 (DIR2 = 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>GPIO0 is read as 1 (DIR2 = 0)</td>
</tr>
<tr>
<td>1</td>
<td>DAT1</td>
<td></td>
<td>0</td>
<td>GPIO data bit 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LOW</td>
<td>0</td>
<td>GPIO0 is driven with a 0 (DIR1 = 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>GPIO0 is read as 0 (DIR1 = 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HI</td>
<td>1</td>
<td>GPIO0 is driven with a 1 (DIR1 = 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>GPIO0 is read as 1 (DIR1 = 0)</td>
</tr>
<tr>
<td>0</td>
<td>DAT0</td>
<td></td>
<td>0</td>
<td>GPIO data bit 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LOW</td>
<td>0</td>
<td>GPIO0 is driven with a 0 (DIR0 = 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>GPIO0 is read as 0 (DIR0 = 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HI</td>
<td>1</td>
<td>GPIO0 is driven with a 1 (DIR0 = 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>GPIO0 is read as 1 (DIR0 = 0)</td>
</tr>
</tbody>
</table>
A.4 HPI Registers

A.4.1 General Purpose I/O Control Register (GPIOCR)

Figure A–25. General Purpose I/O Control Register (GPIOCR)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>TOUT1</td>
<td></td>
<td>0</td>
<td>The timer1 output is not available externally.</td>
</tr>
<tr>
<td></td>
<td>MASK</td>
<td>1</td>
<td></td>
<td>The timer1 output is driven on the HINT pin.</td>
</tr>
<tr>
<td>14-8</td>
<td>reserved</td>
<td></td>
<td></td>
<td>Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.</td>
</tr>
<tr>
<td>7-0</td>
<td>DIR</td>
<td></td>
<td>0</td>
<td>The HD corresponding pin is configured as an input.</td>
</tr>
<tr>
<td></td>
<td>MASK</td>
<td>1</td>
<td></td>
<td>The HD corresponding pin is configured as an output. When the HPI-8 is enabled, this bit is forced to 0 and is not affected by writes.</td>
</tr>
</tbody>
</table>

† Only available on devices with a second on-chip timer.

Legend:  R/W-x = Read/Write-Reset value
A.4.2 General Purpose I/O Status Register (GPIOSR)

Figure A–26. General Purpose Status Register (GPIOSR)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field†</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-0</td>
<td>IO</td>
<td></td>
<td></td>
<td>I/O pin status bit reflects the logic level on the HD pin. When the HD pin is configured as an input (DIR = 0 in GPIOCR), the IO bit latches the logic value (1 or 0) of the HD pin. Writes to the IO bit have no effect when the HD pin is configured as an input. When the HD pin is configured as an output (DIR = 1 in GPIOCR), the HD pin is driven to the logic level (1 or 0) written in the IO bit.</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td>The HD input is externally driven low, or the HD output is internally driven low.</td>
</tr>
<tr>
<td>MASK</td>
<td>1</td>
<td></td>
<td>1</td>
<td>The HD input is externally driven high, or the HD output is internally driven high.</td>
</tr>
</tbody>
</table>

† The GPIOSR register can be treated as a single field register (IO).

A.4.3 HPI Control Register (HPIC) (for 5401, 5402, 5409, and 5410 only)

Figure A–27. HPI Control Register (HPIC) (for 5410)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field†</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td></td>
<td></td>
<td>1</td>
<td>HINT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>SMOD</td>
</tr>
</tbody>
</table>

Note:  

X = Any value can be written

Refer to the specific device data sheet for an explanation of the fields of this register.
A.5 Multichannel BSP (McBSP) Registers

A.5.1 McBSP Serial Port Control Register (SPCR1)

Figure A-28. McBSP Serial Port Control Register 1 (SPCR1)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>DLB</td>
<td></td>
<td>OFF</td>
<td>Digital loop back mode is disabled.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>Digital loop back mode is disabled.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ON</td>
<td>Digital loop back mode is enabled.</td>
</tr>
<tr>
<td>14-13</td>
<td>RJUST</td>
<td></td>
<td>RZF</td>
<td>Right-justify and zero-fill MSBs in DRR[1, 2].</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>00</td>
<td>Right-justify and zero-fill MSBs in DRR[1, 2].</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RSE</td>
<td>Right-justify and sign-extend MSBs in DRR[1, 2].</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>01</td>
<td>Right-justify and sign-extend MSBs in DRR[1, 2].</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LZF</td>
<td>Left-justify and zero-fill LSBs in DRR[1, 2].</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td>Left-justify and zero-fill LSBs in DRR[1, 2].</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td>Reserved</td>
</tr>
<tr>
<td>12-11</td>
<td>CLKSTP</td>
<td></td>
<td>DISABLE</td>
<td>Clock stop mode is disabled. Normal clocking for non-SPI mode.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x</td>
<td>Clock stop mode is disabled. Normal clocking for non-SPI mode.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>NODELAY</td>
<td>Clock starts with rising edge without delay.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td>Clock starts with rising edge without delay.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DELAY</td>
<td>Clock starts with rising edge with delay.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td>Clock starts with rising edge with delay.</td>
</tr>
<tr>
<td>10-8</td>
<td>reserved</td>
<td></td>
<td>NODELAY</td>
<td>Clock starts with falling edge without delay.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td>Clock starts with falling edge without delay.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DELAY</td>
<td>Clock starts with falling edge with delay.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td>Clock starts with falling edge with delay.</td>
</tr>
</tbody>
</table>

† Only available on specific devices.

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

Table A-27. McBSP Serial Port Control Register 1 (SPCR1) Field Values (MCBSP_SPCR1_field_symval)

**Bit field symval Value Description**

15 DLB Digital loop back mode enable bit.

OFF 0 Digital loop back mode is disabled.

ON 1 Digital loop back mode is enabled.

14-13 RJUST Receive sign-extension and justification mode bit.

RZF 00 Right-justify and zero-fill MSBs in DRR[1, 2].

RSE 01 Right-justify and sign-extend MSBs in DRR[1, 2].

LZF 10 Left-justify and zero-fill LSBs in DRR[1, 2].

11 Reserved

12-11 CLKSTP Clock stop mode bit. In SPI mode, operates in conjunction with CLKXP bit of Pin Control Register (PCR).

DISABLE 0x Clock stop mode is disabled. Normal clocking for non-SPI mode.

In SPI mode with data sampled on rising edge (CLKXP = 0):

NODELAY 10 Clock starts with rising edge without delay.

DELAY 11 Clock starts with rising edge with delay.

In SPI mode with data sampled on falling edge (CLKXP = 1):

NODELAY 10 Clock starts with falling edge without delay.

DELAY 11 Clock starts with falling edge with delay.

10-8 reserved Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.
<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>DXENA</td>
<td></td>
<td></td>
<td>DX enabler bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OFF</td>
<td>0</td>
<td>DX enabler is off.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ON</td>
<td>1</td>
<td>DX enabler is on.</td>
</tr>
<tr>
<td>6</td>
<td>ABIS</td>
<td></td>
<td></td>
<td>For C5410, C5410A, and C5416: A-bis enable mode bit.</td>
</tr>
<tr>
<td></td>
<td>DISABLE</td>
<td>0</td>
<td>A-bis mode is disabled.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ENABLE</td>
<td>1</td>
<td>A-bis mode is enabled.</td>
<td></td>
</tr>
<tr>
<td>5-4</td>
<td>RINTM</td>
<td>RRDY</td>
<td>00</td>
<td>RINT is driven by RRDY (end-of-word) and end-of-frame in A-bis mode.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>EOS</td>
<td>01</td>
<td>RINT is generated by end-of-block or end-of-frame in multichannel operation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FRM</td>
<td>10</td>
<td>RINT is generated by a new frame synchronization.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RSYNCERR</td>
<td>11</td>
<td>RINT is generated by RSYNCERR.</td>
</tr>
<tr>
<td>3</td>
<td>RSYNCERR</td>
<td>NO</td>
<td>0</td>
<td>No synchronization error is detected.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>YES</td>
<td>1</td>
<td>Synchronization error is detected.</td>
</tr>
<tr>
<td>2</td>
<td>RFULL</td>
<td></td>
<td></td>
<td>Receive shift register full bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NO</td>
<td>0</td>
<td>RBR[1, 2] is not in overrun condition.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>YES</td>
<td>1</td>
<td>DRR[1, 2] is not read, RBR[1, 2] is full, and RSR[1, 2] is also full with new word.</td>
</tr>
<tr>
<td>1</td>
<td>RRDY</td>
<td></td>
<td></td>
<td>Receiver ready bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NO</td>
<td>0</td>
<td>Receiver is not ready.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>YES</td>
<td>1</td>
<td>Receiver is ready with data to be read from DRR[1, 2].</td>
</tr>
<tr>
<td>0</td>
<td>RRST</td>
<td>DISABLE</td>
<td>0</td>
<td>The serial port receiver is disabled and in reset state.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ENABLE</td>
<td>1</td>
<td>The serial port receiver is enabled.</td>
</tr>
</tbody>
</table>
A.5.2 McBSP Serial Port Control Register 2 (SPCR2)

Figure A–29. McBSP Serial Port Control Register 2 (SPCR2)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-10</td>
<td>reserved</td>
<td></td>
<td></td>
<td>Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.</td>
</tr>
<tr>
<td>9</td>
<td>FREE</td>
<td></td>
<td></td>
<td>Free-running enable mode bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NO</td>
<td>0</td>
<td>Free-running mode is disabled.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>YES</td>
<td>1</td>
<td>Free-running mode is enabled.</td>
</tr>
<tr>
<td>8</td>
<td>SOFT</td>
<td></td>
<td></td>
<td>Soft bit enable mode bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NO</td>
<td>0</td>
<td>Soft mode is disabled.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>YES</td>
<td>1</td>
<td>Soft mode is enabled.</td>
</tr>
<tr>
<td>7</td>
<td>FRST</td>
<td></td>
<td></td>
<td>Frame-sync generator reset.</td>
</tr>
<tr>
<td></td>
<td>RESET</td>
<td></td>
<td>0</td>
<td>Frame-synchronization logic is reset. Frame-sync signal (FSG) is not generated by the sample-rate generator.</td>
</tr>
<tr>
<td></td>
<td>FSG</td>
<td></td>
<td>1</td>
<td>Frame-sync signal (FSG) is generated after ((FPER + 1)) number of CLKG clocks; that is, all frame counters are loaded with their programmed values.</td>
</tr>
<tr>
<td>6</td>
<td>GRST</td>
<td></td>
<td></td>
<td>Sample-rate generator reset.</td>
</tr>
<tr>
<td></td>
<td>RESET</td>
<td></td>
<td>0</td>
<td>Sample-rate generator is reset.</td>
</tr>
<tr>
<td></td>
<td>CLKG</td>
<td></td>
<td>1</td>
<td>Sample-rate generator is taken out of reset. CLKG is driven as per programmed value in sample-rate generator registers ((SRGR[1, 2])).</td>
</tr>
</tbody>
</table>

† Caution: Writing a 1 to this bit sets the error condition; thus, it is mainly used for testing purposes or if this operation is desired.

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

Table A–28. McBSP Serial Port Control Register 2 (SPCR2) Field Values

(MCBSP_SPCR2_field_symval)
### McBSP Serial Port Control Register 2 (SPCR2) Field Values (MCBSP_SPCR2_field_symval) (Continued)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-4</td>
<td>XINTM</td>
<td></td>
<td></td>
<td>Transmit interrupt (XINT) mode bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>XRDY</td>
<td>00  XINT is driven by XRDY (end-of-word) and end-of-frame in A-bis mode.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>EOS</td>
<td>01  XINT is generated by end-of-block or end-of-frame in multi-channel operation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>FRM</td>
<td>10  XINT is generated by a new frame synchronization.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>XSYNCERR</td>
<td>11  XINT is generated by XSYNCERR.</td>
</tr>
<tr>
<td>3</td>
<td>XSYNCERR</td>
<td></td>
<td></td>
<td>Transmit synchronization error bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>NO</td>
<td>0   No synchronization error is detected.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>YES</td>
<td>1   Synchronization error is detected.</td>
</tr>
<tr>
<td>2</td>
<td>XEMPTY</td>
<td></td>
<td></td>
<td>Transmit shift register empty bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>YES</td>
<td>0   XSR[1, 2] is empty.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>NO</td>
<td>1   XSR[1, 2] is not empty.</td>
</tr>
<tr>
<td>1</td>
<td>XRDY</td>
<td></td>
<td></td>
<td>Transmitter ready bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>NO</td>
<td>0   Transmitter is not ready.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>YES</td>
<td>1   Transmitter is ready for new data in DXR[1, 2].</td>
</tr>
<tr>
<td>0</td>
<td>XRST</td>
<td></td>
<td></td>
<td>Transmitter reset bit resets or enables the transmitter.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DISABLE</td>
<td></td>
<td>0   Serial port transmitter is disabled and in reset state.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ENABLE</td>
<td></td>
<td>1   Serial port transmitter is enabled.</td>
</tr>
</tbody>
</table>
### A.5.3 McBSP Pin Control Register (PCR)

**Figure A–30. McBSP Pin Control Register (PCR)**

![McBSP Pin Control Register (PCR)](image)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-14</td>
<td>reserved</td>
<td></td>
<td>R/W-0</td>
<td>Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.</td>
</tr>
<tr>
<td>13</td>
<td>XIOEN</td>
<td></td>
<td>0</td>
<td>DX, FSX, and CLKX pins are configured as serial port pins and do not function as general-purpose I/O pins.</td>
</tr>
<tr>
<td>12</td>
<td>RIOEN</td>
<td></td>
<td>0</td>
<td>DR, FSR, CLKR, and CLKS pins are configured as serial port pins and do not function as general-purpose I/O pins.</td>
</tr>
<tr>
<td>11</td>
<td>FSXM</td>
<td></td>
<td>0</td>
<td>Frame-synchronization signal is derived from an external source.</td>
</tr>
<tr>
<td>11</td>
<td>FSRM</td>
<td></td>
<td>0</td>
<td>Frame-synchronization signal is determined by FSGM bit in SRGR2.</td>
</tr>
<tr>
<td>10</td>
<td>CLKXM</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>CLKRM</td>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

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

**Legend:**  R/W-x = Read/Write-Reset value

**Table A–29. McBSP Pin Control Register (PCR) Field Values**

(MCBSP_PCR_field_symval)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>FSXM</td>
<td></td>
<td>0</td>
<td>Transmit frame-synchronization mode bit.</td>
</tr>
<tr>
<td>11</td>
<td>FSRM</td>
<td></td>
<td>0</td>
<td>Frame-synchronization signal is derived from an external source.</td>
</tr>
<tr>
<td>11</td>
<td>CLKXM</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>CLKRM</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>CLKXM</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>CLKRM</td>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**Legend:**
- **R/W-x:** Read/Write-Reset value
- **XIOEN:** Transmit general-purpose I/O mode only when transmitter is disabled (XRST = 0 in SPCR2).
- **SP 0 DX, FSX, and CLKX pins are configured as serial port pins and do not function as general-purpose I/O pins.**
- **GPIO 1 DX pin is configured as general-purpose output pin; FSX and CLKX pins are configured as general-purpose I/O pins. These serial port pins do not perform serial port operations.**
- **RIOEN:** Receive general-purpose I/O mode only when receiver is disabled (RRST = 0 in SPCR1).
- **SP 0 DR, FSR, CLKR, and CLKS pins are configured as serial port pins and do not function as general-purpose I/O pins.**
- **GPIO 1 DR and CLKS pins are configured as general-purpose input pins; FSR and CLKR pins are configured as general-purpose I/O pins. These serial port pins do not perform serial port operations.**

A-30
### Table A–29. McBSP Pin Control Register (PCR) Field Values (MCBSP_PCR_field_symval) (Continued)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>FSRM</td>
<td></td>
<td></td>
<td>Receive frame-synchronization mode bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>EXTERNAL</td>
<td>0</td>
<td>Frame-synchronization signal is derived from an external source. FSR is an input pin.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTERNAL</td>
<td>1</td>
<td>Frame-synchronization signal is generated internally by the sample-rate generator. FSR is an output pin, except when GSYNC = 1 in SRGR2.</td>
</tr>
<tr>
<td>9</td>
<td>CLKXM</td>
<td></td>
<td></td>
<td>Transmitter clock mode bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INPUT</td>
<td>0</td>
<td>CLXX is an input pin and is driven by an external clock.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OUTPUT</td>
<td>1</td>
<td>CLXX is an output pin and is driven by the internal sample-rate generator.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>In SPI mode when CLKSTP in SPCR1 is a non-zero value:</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>INPUT</td>
<td>0</td>
<td>McBSP is a slave and clock (CLXX) is driven by the SPI master in the system. CLKR is internally driven by CLXX.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OUTPUT</td>
<td>1</td>
<td>McBSP is a master and generates the clock (CLXX) to drive its receive clock (CLKR) and the shift clock of the SPI-compliant slaves in the system.</td>
</tr>
<tr>
<td>8</td>
<td>CLKRM</td>
<td></td>
<td></td>
<td>Receiver clock mode bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INPUT</td>
<td>0</td>
<td>CLKR is an input pin and is driven by an external clock.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OUTPUT</td>
<td>1</td>
<td>CLKR is an output pin and is driven by the internal sample-rate generator.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>Digital loop back mode is disabled (DLB = 0 in SPCR1):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>INPUT</td>
<td>0</td>
<td>Receive clock (not the CLKR pin) is driven by transmit clock (CLXX) that is based on CLKXM bit. CLKR pin is in high-impedance state.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OUTPUT</td>
<td>1</td>
<td>CLKR is an output pin and is driven by the transmit clock. The transmit clock is based on CLKXM bit.</td>
</tr>
<tr>
<td>7</td>
<td>SCLKME</td>
<td></td>
<td></td>
<td><strong>For devices with 128-channel selection capability:</strong> Sample-rate clock mode extended enable bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NO</td>
<td>0</td>
<td>BCLKR and BCLKX are not used by the sample-rate generator for external synchronization.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BCLK</td>
<td>1</td>
<td>BCLKR and BCLKX are used by the sample-rate generator for external synchronization.</td>
</tr>
<tr>
<td>Bit</td>
<td>field</td>
<td>symval</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>---------</td>
<td>--------</td>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>6</td>
<td>CLKSSTAT</td>
<td>0x0</td>
<td>0</td>
<td>CLKS pin reflects a logic low.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x1</td>
<td>1</td>
<td>CLKS pin reflects a logic high.</td>
</tr>
<tr>
<td>5</td>
<td>DXSTAT</td>
<td>0x0</td>
<td>0</td>
<td>DX pin reflects a logic low.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x1</td>
<td>1</td>
<td>DX pin reflects a logic high.</td>
</tr>
<tr>
<td>4</td>
<td>DRSTAT</td>
<td>0x0</td>
<td>0</td>
<td>DR pin reflects a logic low.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x1</td>
<td>1</td>
<td>DR pin reflects a logic high.</td>
</tr>
<tr>
<td>3</td>
<td>FSXP</td>
<td>ACTIVEHIGH</td>
<td>0</td>
<td>Transmit frame-synchronization pulse is active high.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ACTIVELOW</td>
<td>1</td>
<td>Transmit frame-synchronization pulse is active low.</td>
</tr>
<tr>
<td>2</td>
<td>FSRP</td>
<td>ACTIVEHIGH</td>
<td>0</td>
<td>Receive frame-synchronization pulse is active high.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ACTIVELOW</td>
<td>1</td>
<td>Receive frame-synchronization pulse is active low.</td>
</tr>
<tr>
<td>1</td>
<td>CLKXP</td>
<td>RISING</td>
<td>0</td>
<td>Transmit data sampled on rising edge of CLKX.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FALLING</td>
<td>1</td>
<td>Transmit data sampled on falling edge of CLKX.</td>
</tr>
<tr>
<td>0</td>
<td>CLKRP</td>
<td>FALLING</td>
<td>0</td>
<td>Receive data sampled on falling edge of CLKR.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RISING</td>
<td>1</td>
<td>Receive data sampled on rising edge of CLKR.</td>
</tr>
</tbody>
</table>
A.5.4 Receive Control Register 1 (RCR1)

Figure A–31. Receive Control Register 1 (RCR1)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>reserved</td>
<td></td>
<td></td>
<td>Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.</td>
</tr>
<tr>
<td>14-8</td>
<td>RFRLEN1</td>
<td>OF(value)</td>
<td>0-127</td>
<td>Specifies the number of words (length) in the receive frame.</td>
</tr>
<tr>
<td>7-5</td>
<td>RWDLEN1</td>
<td></td>
<td></td>
<td>Specifies the number of bits (length) in the receive word.</td>
</tr>
<tr>
<td></td>
<td>8BIT</td>
<td></td>
<td>000</td>
<td>Receive word length is 8 bits.</td>
</tr>
<tr>
<td></td>
<td>12BIT</td>
<td></td>
<td>001</td>
<td>Receive word length is 12 bits.</td>
</tr>
<tr>
<td></td>
<td>16BIT</td>
<td></td>
<td>010</td>
<td>Receive word length is 16 bits.</td>
</tr>
<tr>
<td></td>
<td>20BIT</td>
<td></td>
<td>011</td>
<td>Receive word length is 20 bits.</td>
</tr>
<tr>
<td></td>
<td>24BIT</td>
<td></td>
<td>100</td>
<td>Receive word length is 24 bits.</td>
</tr>
<tr>
<td></td>
<td>32BIT</td>
<td></td>
<td>101</td>
<td>Receive word length is 32 bits.</td>
</tr>
<tr>
<td></td>
<td>110</td>
<td></td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>111</td>
<td></td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>4-0</td>
<td>reserved</td>
<td></td>
<td></td>
<td>Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.</td>
</tr>
</tbody>
</table>
A.5.5 Receive Control Register 2 (RCR2)

Figure A-32. Receive Control Register 2 (RCR2)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>RPHASE</td>
<td></td>
<td></td>
<td>Receive phases bit.</td>
</tr>
<tr>
<td></td>
<td>SINGLE</td>
<td>0</td>
<td>Single-phase frame</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DUAL</td>
<td>1</td>
<td>Dual-phase frame</td>
<td></td>
</tr>
<tr>
<td>14-8</td>
<td>RFRLEN2</td>
<td>OF(value)</td>
<td>0-127</td>
<td>Specifies the number of words (length) in the receive frame.</td>
</tr>
<tr>
<td>7-5</td>
<td>RWDLEN2</td>
<td></td>
<td></td>
<td>Specifies the number of bits (length) in the receive word.</td>
</tr>
<tr>
<td></td>
<td>8BIT</td>
<td>000</td>
<td>8BIT</td>
<td>Receive word length is 8 bits.</td>
</tr>
<tr>
<td></td>
<td>12BIT</td>
<td>001</td>
<td>12BIT</td>
<td>Receive word length is 12 bits.</td>
</tr>
<tr>
<td></td>
<td>16BIT</td>
<td>010</td>
<td>16BIT</td>
<td>Receive word length is 16 bits.</td>
</tr>
<tr>
<td></td>
<td>20BIT</td>
<td>011</td>
<td>20BIT</td>
<td>Receive word length is 20 bits.</td>
</tr>
<tr>
<td></td>
<td>24BIT</td>
<td>100</td>
<td>24BIT</td>
<td>Receive word length is 24 bits.</td>
</tr>
<tr>
<td></td>
<td>32BIT</td>
<td>101</td>
<td>32BIT</td>
<td>Receive word length is 32 bits.</td>
</tr>
<tr>
<td></td>
<td>110</td>
<td></td>
<td>110</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>111</td>
<td></td>
<td>111</td>
<td>Reserved</td>
</tr>
<tr>
<td>4-3</td>
<td>RCOMPAND</td>
<td></td>
<td></td>
<td>Receive companding mode. Modes other than 00 are only enabled when RWDLEN[1, 2] bit is 000 (indicating 8-bit data).</td>
</tr>
<tr>
<td></td>
<td>MSB</td>
<td>00</td>
<td>No companding, data transfer starts with MSB first.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8BITLSB</td>
<td>01</td>
<td>No companding, 8-bit data transfer starts with LSB first.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ULAW</td>
<td>10</td>
<td>Compand using µ-law for receive data.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ALAW</td>
<td>11</td>
<td>Compand using A-law for receive data.</td>
<td></td>
</tr>
</tbody>
</table>
Table A–31. Receive Control Register 2 (RCR2) Field Values  
(MCBSP_RCR2_field_symval) (Continued)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>RFIG</td>
<td></td>
<td></td>
<td>Receive frame ignore bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>YES</td>
<td>0</td>
<td>Receive frame-synchronization pulses after the first pulse restarts the transfer.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NO</td>
<td>1</td>
<td>Receive frame-synchronization pulses after the first pulse are ignored.</td>
</tr>
<tr>
<td>1-0</td>
<td>RDATDLY</td>
<td></td>
<td></td>
<td>Receive data delay bit.</td>
</tr>
<tr>
<td></td>
<td>0BIT</td>
<td>00</td>
<td></td>
<td>0-bit data delay</td>
</tr>
<tr>
<td></td>
<td>1BIT</td>
<td>01</td>
<td></td>
<td>1-bit data delay</td>
</tr>
<tr>
<td></td>
<td>2BIT</td>
<td>10</td>
<td></td>
<td>2-bit data delay</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11</td>
<td></td>
<td>Reserved</td>
</tr>
</tbody>
</table>

A.5.6 Transmit Control Register 1 (XCR1)

Figure A–33. Transmit Control Register 1 (XCR1)

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>reserved</td>
<td>XFRLEN1</td>
<td></td>
</tr>
<tr>
<td>R/W-0</td>
<td>R/W-0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>XWDLEN1</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>R/W-0</td>
<td>R/W-0</td>
<td></td>
</tr>
</tbody>
</table>

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

Table A–32. Transmit Control Register 1 (XCR1) Field Values  
(MCBSP_XCR1_field_symval)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>reserved</td>
<td></td>
<td></td>
<td>Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.</td>
</tr>
<tr>
<td>14-8</td>
<td>XFRLEN1</td>
<td>OF(value)</td>
<td>0-127</td>
<td>Specifies the number of words (length) in the transmit frame.</td>
</tr>
<tr>
<td>7-5</td>
<td>XWDLEN1</td>
<td></td>
<td></td>
<td>Specifies the number of bits (length) in the transmit word.</td>
</tr>
<tr>
<td></td>
<td>8BIT</td>
<td>000</td>
<td></td>
<td>Transmit word length is 8 bits.</td>
</tr>
<tr>
<td></td>
<td>12BIT</td>
<td>001</td>
<td></td>
<td>Transmit word length is 12 bits.</td>
</tr>
<tr>
<td></td>
<td>16BIT</td>
<td>010</td>
<td></td>
<td>Transmit word length is 16 bits.</td>
</tr>
</tbody>
</table>
### Table A–32. Transmit Control Register 1 (XCR1) Field Values (MCBSP_XCR1_field_symval)

<table>
<thead>
<tr>
<th>Bit field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>20BIT</td>
<td>011</td>
<td></td>
<td>Transmit word length is 20 bits.</td>
</tr>
<tr>
<td>24BIT</td>
<td>100</td>
<td></td>
<td>Transmit word length is 24 bits.</td>
</tr>
<tr>
<td>32BIT</td>
<td>101</td>
<td></td>
<td>Transmit word length is 32 bits.</td>
</tr>
<tr>
<td>110</td>
<td></td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>111</td>
<td></td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>4-0</td>
<td>reserved</td>
<td></td>
<td>Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.</td>
</tr>
</tbody>
</table>

#### A.5.7 Transmit Control Register 2 (XCR2)

### Figure A–34. Transmit Control Register 2 (XCR2)

<table>
<thead>
<tr>
<th>Bit field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>XPHASE</td>
<td></td>
<td>Transmit phases bit.</td>
</tr>
<tr>
<td>SINGLE</td>
<td>0</td>
<td></td>
<td>Single-phase frame</td>
</tr>
<tr>
<td>DUAL</td>
<td>1</td>
<td></td>
<td>Dual-phase frame</td>
</tr>
<tr>
<td>14-8</td>
<td>XFRLEN2</td>
<td>OF(value)</td>
<td>Specifies the number of words (length) in the transmit frame.</td>
</tr>
<tr>
<td>7-5</td>
<td>XWDLEN2</td>
<td></td>
<td>Specifies the number of bits (length) in the transmit word.</td>
</tr>
<tr>
<td>8BIT</td>
<td>000</td>
<td></td>
<td>Transmit word length is 8 bits.</td>
</tr>
<tr>
<td>12BIT</td>
<td>001</td>
<td></td>
<td>Transmit word length is 12 bits.</td>
</tr>
<tr>
<td>16BIT</td>
<td>010</td>
<td></td>
<td>Transmit word length is 16 bits.</td>
</tr>
<tr>
<td>20BIT</td>
<td>011</td>
<td></td>
<td>Transmit word length is 20 bits.</td>
</tr>
<tr>
<td>24BIT</td>
<td>100</td>
<td></td>
<td>Transmit word length is 24 bits.</td>
</tr>
</tbody>
</table>
### Table A–33. Transmit Control Register 2 (XCR2) Field Values (MCBSP_XCR2_field_symval) (Continued)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32BIT</td>
<td>101</td>
<td>Transmit word length is 32 bits.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>110</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>111</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4-3</td>
<td>XCOMPAND</td>
<td></td>
<td></td>
<td>Transmit companding mode. Modes other than 00 are only enabled when XWDLEN[1, 2] bit is 000 (indicating 8-bit data).</td>
</tr>
<tr>
<td>MSB</td>
<td>00</td>
<td>No companding, data transfer starts with MSB first.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8BITLSB</td>
<td>01</td>
<td>No companding, 8-bit data transfer starts with LSB first.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ULAW</td>
<td>10</td>
<td>Compand using μ-law for transmit data.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALAW</td>
<td>11</td>
<td>Compand using A-law for transmit data.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>XFIG</td>
<td>YES</td>
<td>0</td>
<td>Transmit frame-synchronization pulses after the first pulse restarts the transfer.</td>
</tr>
<tr>
<td></td>
<td>NO</td>
<td>1</td>
<td>Transmit frame-synchronization pulses after the first pulse are ignored.</td>
<td></td>
</tr>
<tr>
<td>1-0</td>
<td>XDATDLY</td>
<td>0BIT</td>
<td>00</td>
<td>0-bit data delay</td>
</tr>
<tr>
<td></td>
<td>1BIT</td>
<td>01</td>
<td>1-bit data delay</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2BIT</td>
<td>10</td>
<td>2-bit data delay</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A.5.8 Sample Rate Generator Register 1 (SRGR1)

Figure A–35. Sample Rate Generator Register 1 (SRGR1)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-8</td>
<td>FWID</td>
<td>OF(value)</td>
<td>0-255</td>
<td>The value plus 1 specifies the width of the frame-sync pulse (FSG) during its active period.</td>
</tr>
<tr>
<td>7-0</td>
<td>CLKGDV</td>
<td>OF(value)</td>
<td>0-255</td>
<td>The value is used as the divide-down number to generate the required sample-rate generator clock frequency.</td>
</tr>
</tbody>
</table>

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

Table A–34. Sample Rate Generator Register 1 (SRGR1) Field Values (MCBSP_SRGR1_field_symval)

A.5.9 Sample Rate Generator Register 2 (SRGR2)

Figure A–36. Sample Rate Generator Register 2 (SRGR2)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>GSYNC</td>
<td></td>
<td></td>
<td>Sample-rate generator clock synchronization bit only used when the external clock (CLKS) drives the sample-rate generator clock (CLKSM = 0).</td>
</tr>
<tr>
<td>14</td>
<td>CLKSP</td>
<td></td>
<td></td>
<td>The sample-rate generator clock (CLKG) is free running.</td>
</tr>
<tr>
<td>13</td>
<td>CLKSM</td>
<td></td>
<td></td>
<td>The sample-rate generator clock (CLKG) is running; however, CLKG is resynchronized and frame-sync signal (FSG) is generated only after detecting the receive frame-synchronization signal (FSR). Also, frame period (FPER) is a don’t care because the period is dictated by the external frame-sync pulse.</td>
</tr>
<tr>
<td>12</td>
<td>FSGM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>FPER</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Table A–35. Sample Rate Generator Register 2 (SRGR2) Field Values (MCBSP_SRGR2_field_symval)
Table A–35. Sample Rate Generator Register 2 (SRGR2) Field Values (MCBSP_SRGR2_field_symval) (Continued)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>CLKSP</td>
<td></td>
<td>0x0</td>
<td>CLKS polarity clock edge select bit only used when the external clock (CLKS) drives the sample-rate generator clock (CLKSM = 0).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RISING</td>
<td>0</td>
<td>Rising edge of CLKS generates CLKG and FSG.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FALLING</td>
<td>1</td>
<td>Falling edge of CLKS generates CLKG and FSG.</td>
</tr>
<tr>
<td>13</td>
<td>CLKSM</td>
<td></td>
<td></td>
<td>McBSP sample-rate generator clock mode bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CLKS</td>
<td>0</td>
<td>Sample-rate generator clock derived from the CLKS pin.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTERNAL</td>
<td>1</td>
<td>Sample-rate generator clock derived from CPU clock.</td>
</tr>
<tr>
<td>12</td>
<td>FSGM</td>
<td></td>
<td></td>
<td>Sample-rate generator transmit frame-synchronization mode bit used when FSXM = 1 in PCR.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DXR2XSR</td>
<td>0</td>
<td>Transmit frame-sync signal (FSX) due to DXR[1, 2]-to-XSR[1, 2] copy. When FSGM = 0, FWID bit in SRGR1 and FPER bit are ignored.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FSG</td>
<td>1</td>
<td>Transmit frame-sync signal (FSX) driven by the sample-rate generator frame-sync signal (FSG).</td>
</tr>
<tr>
<td>11-0</td>
<td>FPER</td>
<td>OF(value)</td>
<td>0-4095</td>
<td>The value plus 1 specifies when the next frame-sync signal becomes active. Range: 1 to 4096 sample-rate generator clock (CLKG) periods.</td>
</tr>
</tbody>
</table>
A.5.10 Multichannel Control Register 1 (MCR1)

Figure A–37. Multichannel Control Register 1 (MCR1)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Symbol</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-10</td>
<td>reserved</td>
<td></td>
<td></td>
<td>Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.</td>
</tr>
<tr>
<td>9</td>
<td>RMCME</td>
<td></td>
<td></td>
<td>For devices with 128-channel selection capability: Receive 128-channel selection enable bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NO</td>
<td>0</td>
<td>Normal 32-channel selection is enabled.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ATOH</td>
<td>1</td>
<td>Six additional registers (RCERC-RCERH) are used to enable 128-channel selection.</td>
</tr>
<tr>
<td>8-7</td>
<td>RPBBLK</td>
<td></td>
<td></td>
<td>Receive partition B block bit. Enables 16 contiguous channels in each block.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SF1</td>
<td>00</td>
<td>Block 1. Channel 16 to channel 31</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SF3</td>
<td>01</td>
<td>Block 3. Channel 48 to channel 63</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SF5</td>
<td>10</td>
<td>Block 5. Channel 80 to channel 95</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SF7</td>
<td>11</td>
<td>Block 7. Channel 112 to channel 127</td>
</tr>
<tr>
<td>6-5</td>
<td>RPABLK</td>
<td></td>
<td></td>
<td>Receive partition A block bit. Enables 16 contiguous channels in each block.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SF0</td>
<td>00</td>
<td>Block 0. Channel 0 to channel 15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SF2</td>
<td>01</td>
<td>Block 2. Channel 32 to channel 47</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SF4</td>
<td>10</td>
<td>Block 4. Channel 64 to channel 79</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SF6</td>
<td>11</td>
<td>Block 6. Channel 96 to channel 111</td>
</tr>
<tr>
<td>4-2</td>
<td>RCBLK</td>
<td></td>
<td></td>
<td>Receive current block bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SF0</td>
<td>000</td>
<td>Block 0. Channel 0 to channel 15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SF1</td>
<td>001</td>
<td>Block 1. Channel 16 to channel 31</td>
</tr>
</tbody>
</table>

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

Legend: R/W-x = Read/Write-Reset value
Table A–36. Multichannel Control Register 1 (MCR1) Field Values (MCBSP_MCR1_field_symval) (Continued)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SF2</td>
<td>010</td>
<td>Block 2. Channel 32 to channel 47</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SF3</td>
<td>011</td>
<td>Block 3. Channel 48 to channel 63</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SF4</td>
<td>100</td>
<td>Block 4. Channel 64 to channel 79</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RCBLK</td>
<td>SF5</td>
<td>101</td>
<td>Block 5. Channel 80 to channel 95</td>
<td></td>
</tr>
<tr>
<td>SF6</td>
<td>110</td>
<td>Block 6. Channel 96 to channel 111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SF7</td>
<td>111</td>
<td>Block 7. Channel 112 to channel 127</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>reserved</td>
<td>Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>RMCM</td>
<td>Receive multichannel selection enable bit.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHENABLE</td>
<td>0</td>
<td>All 128 channels enabled.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ELDISABLE</td>
<td>1</td>
<td>All channels disabled by default. Required channels are selected by enabling RP[A, B]BLK and RCER[A, B] appropriately.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A.5.11 Multichannel Control Register 2 (MCR2)

Figure A–38. Multichannel Control Register 2 (MCR2)

<table>
<thead>
<tr>
<th>15</th>
<th>10</th>
<th>9</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>reserved</td>
<td>XMCME†</td>
<td>XPBBLK</td>
<td></td>
</tr>
<tr>
<td>R/W-0</td>
<td>R/W-0</td>
<td>R/W-0</td>
<td></td>
</tr>
<tr>
<td>7 6</td>
<td>5 4</td>
<td>2 1</td>
<td>0</td>
</tr>
<tr>
<td>XPBBLK</td>
<td>XPABLK</td>
<td>XCBLK</td>
<td>XMCM</td>
</tr>
<tr>
<td>R/W-0</td>
<td>R/W-0</td>
<td>R-0</td>
<td>R/W-0</td>
</tr>
</tbody>
</table>

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

Legend: R/W-x = Read/Write-Reset value
### Multichannel Control Register 2 (MCR2) Field Values

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-10</td>
<td>reserved</td>
<td></td>
<td></td>
<td>Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.</td>
</tr>
<tr>
<td>9</td>
<td>XMCME</td>
<td></td>
<td></td>
<td>For devices with 128-channel selection capability: Transmit 128-channel selection enable bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NO</td>
<td>0</td>
<td>Normal 32-channel selection is enabled.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ATOH</td>
<td>1</td>
<td>Six additional registers (XCERC-XCERH) are used to enable 128-channel selection.</td>
</tr>
<tr>
<td>8-7</td>
<td>XPBBLK</td>
<td></td>
<td></td>
<td>Transmit partition B block bit. Enables 16 contiguous channels in each block.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SF1</td>
<td>00</td>
<td>Block 1. Channel 16 to channel 31</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SF3</td>
<td>01</td>
<td>Block 3. Channel 48 to channel 63</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SF5</td>
<td>10</td>
<td>Block 5. Channel 80 to channel 95</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SF7</td>
<td>11</td>
<td>Block 7. Channel 112 to channel 127</td>
</tr>
<tr>
<td>6-5</td>
<td>XPABLK</td>
<td></td>
<td></td>
<td>Transmit partition A block bit. Enables 16 contiguous channels in each block.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SF0</td>
<td>00</td>
<td>Block 0. Channel 0 to channel 15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SF2</td>
<td>01</td>
<td>Block 2. Channel 32 to channel 47</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SF4</td>
<td>10</td>
<td>Block 4. Channel 64 to channel 79</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SF6</td>
<td>11</td>
<td>Block 6. Channel 96 to channel 111</td>
</tr>
<tr>
<td>4-2</td>
<td>XCBLK</td>
<td></td>
<td></td>
<td>Transmit current block bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SF0</td>
<td>000</td>
<td>Block 0. Channel 0 to channel 15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SF1</td>
<td>001</td>
<td>Block 1. Channel 16 to channel 31</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SF2</td>
<td>010</td>
<td>Block 2. Channel 32 to channel 47</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SF3</td>
<td>011</td>
<td>Block 3. Channel 48 to channel 63</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SF4</td>
<td>100</td>
<td>Block 4. Channel 64 to channel 79</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SF5</td>
<td>101</td>
<td>Block 5. Channel 80 to channel 95</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SF6</td>
<td>110</td>
<td>Block 6. Channel 96 to channel 111</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SF7</td>
<td>111</td>
<td>Block 7. Channel 112 to channel 127</td>
</tr>
</tbody>
</table>
Table A–37. Multichannel Control Register 2 (MCR2) Field Values (MCBSP_MCR2_field_symval) (Continued)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-0</td>
<td>XMCM</td>
<td></td>
<td>0x0</td>
<td>Transmit multichannel selection enable bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ENNOMASK</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DISXP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ENMASK</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DISRP</td>
</tr>
</tbody>
</table>

† DX is masked or driven to a high-impedance state during (a) interpacket intervals, (b) when a channel is masked regardless of whether it is enabled, or (c) when a channel is disabled.

A.5.12 Receive Channel Enable Register (RCERn)

Figure A–39. Receive Channel Enable Register (RCERn)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-0</td>
<td>RCEA</td>
<td>OF(value)</td>
<td>0-FFFFh</td>
<td>A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) reception of the nth channel within the 16-channel-wide block in partition A. The 16-channel-wide block is selected by the RPABLK bit in MCR1.</td>
</tr>
</tbody>
</table>

† The register is also treated as having a single field (RCEn).
Table A–38. Receive Channel Enable Register (RCERn) Field Values
(MCBSP_RCERn_field_symval) (Continued)

<table>
<thead>
<tr>
<th>Bit</th>
<th>feld†</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-0</td>
<td>RCEB</td>
<td>OF(value)</td>
<td>0-FFFFh</td>
<td>A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) reception of the n-th channel within the 16-channel-wide block in partition B. The 16-channel-wide block is selected by the RPBBLK bit in MCR1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>For devices with 128-channel selection capability:</strong></td>
</tr>
<tr>
<td>15-0</td>
<td>RCEA</td>
<td>OF(value)</td>
<td>0-FFFFh</td>
<td>A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) reception of channels 0-15 within the 16-channel-wide block.</td>
</tr>
<tr>
<td>15-0</td>
<td>RCEB</td>
<td>OF(value)</td>
<td>0-FFFFh</td>
<td>A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) reception of channels 16-31 within the 16-channel-wide block.</td>
</tr>
<tr>
<td>15-0</td>
<td>RCEC</td>
<td>OF(value)</td>
<td>0-FFFFh</td>
<td>A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) reception of channels 32-47 within the 16-channel-wide block.</td>
</tr>
<tr>
<td>15-0</td>
<td>RCED</td>
<td>OF(value)</td>
<td>0-FFFFh</td>
<td>A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) reception of channels 48-63 within the 16-channel-wide block.</td>
</tr>
<tr>
<td>15-0</td>
<td>RCEE</td>
<td>OF(value)</td>
<td>0-FFFFh</td>
<td>A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) reception of channels 64-79 within the 16-channel-wide block.</td>
</tr>
<tr>
<td>15-0</td>
<td>RCEF</td>
<td>OF(value)</td>
<td>0-FFFFh</td>
<td>A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) reception of channels 80-95 within the 16-channel-wide block.</td>
</tr>
<tr>
<td>15-0</td>
<td>RCEG</td>
<td>OF(value)</td>
<td>0-FFFFh</td>
<td>A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) reception of channels 96-111 within the 16-channel-wide block.</td>
</tr>
<tr>
<td>15-0</td>
<td>RCEH</td>
<td>OF(value)</td>
<td>0-FFFFh</td>
<td>A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) reception of channels 112-127 within the 16-channel-wide block.</td>
</tr>
</tbody>
</table>

† The register is also treated as having a single field (RCEn).
A.5.13 Transmit Channel Enable Register (XCERn)

Figure A–40. Transmit Channel Enable Register (XCERn)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-0</td>
<td>XCEA</td>
<td>OF(value)</td>
<td>0-FFFFh</td>
<td>A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) transmission of the nth channel within the 16-channel-wide block in partition A. The 16-channel-wide block is selected by the XPABLK bit in MCR2.</td>
</tr>
<tr>
<td>15-0</td>
<td>XCEB</td>
<td>OF(value)</td>
<td>0-FFFFh</td>
<td>A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) transmission of the nth channel within the 16-channel-wide block in partition B. The 16-channel-wide block is selected by the XPBBLK bit in MCR2.</td>
</tr>
</tbody>
</table>

Table A–39. Transmit Channel Enable Register (XCERn) Field Values (MCBSP_XCERn_field_symval)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field†</th>
<th>Symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-0</td>
<td>XCEA</td>
<td>OF(value)</td>
<td>0-FFFFh</td>
<td>For devices with only 32-channel selection capability:</td>
</tr>
<tr>
<td>15-0</td>
<td>XCEB</td>
<td>OF(value)</td>
<td>0-FFFFh</td>
<td>A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) transmission of channels 0-15 within the 16-channel-wide block.</td>
</tr>
<tr>
<td>15-0</td>
<td>XCEC</td>
<td>OF(value)</td>
<td>0-FFFFh</td>
<td>A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) transmission of channels 16-31 within the 16-channel-wide block.</td>
</tr>
<tr>
<td>15-0</td>
<td>XCED</td>
<td>OF(value)</td>
<td>0-FFFFh</td>
<td>A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) transmission of channels 32-47 within the 16-channel-wide block.</td>
</tr>
<tr>
<td>15-0</td>
<td>XCEE</td>
<td>OF(value)</td>
<td>0-FFFFh</td>
<td>A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) transmission of channels 48-63 within the 16-channel-wide block.</td>
</tr>
<tr>
<td>15-0</td>
<td>XCEF</td>
<td>OF(value)</td>
<td>0-FFFFh</td>
<td>A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) transmission of channels 64-79 within the 16-channel-wide block.</td>
</tr>
</tbody>
</table>

† The registers are also treated as having a single field (XCEn).
### Transmit Channel Enable Register (XCERn) Field Values (MCBSP_XCERn_field_symval) (Continued)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field†</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-0</td>
<td>XCEF</td>
<td>OF(value)</td>
<td>0-FFFFh</td>
<td>A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) transmission of channels 80-95 within the 16-channel-wide block.</td>
</tr>
<tr>
<td>15-0</td>
<td>XCEG</td>
<td>OF(value)</td>
<td>0-FFFFh</td>
<td>A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) transmission of channels 96-111 within the 16-channel-wide block.</td>
</tr>
<tr>
<td>15-0</td>
<td>XCEH</td>
<td>OF(value)</td>
<td>0-FFFFh</td>
<td>A 16-bit unsigned value used to disable (bit value = 0) or enable (bit value = 1) transmission of channels 112-127 within the 16-channel-wide block.</td>
</tr>
</tbody>
</table>

† The registers are also treated as having a single field (XCEn).
A.6 PLL Registers (CLKMD)

Figure A–41. Clock Mode Register (CLKMD)

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Field</th>
<th>Symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-12</td>
<td>PLLMUL</td>
<td>OF(value)</td>
<td>0-15</td>
<td>This PLL multiplier value defines the frequency multiplier in conjunction with the PLLDIV and PLLNDIV bits.</td>
</tr>
<tr>
<td>11</td>
<td>PLLDIV</td>
<td></td>
<td></td>
<td>PLL divider. Defines the frequency multiplier in conjunction with the PLLMUL and PLLNDIV bits.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OFF</td>
<td>0</td>
<td>PLL is off unless PLLNDIV = 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ON</td>
<td>1</td>
<td>PLL is on regardless of the PLLNDIV bit status.</td>
</tr>
<tr>
<td>10-3</td>
<td>PLLCOUNT</td>
<td>OF(value)</td>
<td>0-255</td>
<td>This PLL counter value specifies the number of input clock cycles (in increments of 16 cycles) for the PLL lock timer to count before the PLL begins clocking the processor after the PLL is started. The PLL counter is a down-counter, which is driven by the input clock divided by 16; therefore, for every 16 input clocks, the PLL counter decrements by 1. The PLL counter can be used to ensure that the processor is not clocked until the PLL is locked, so that only valid clock signals are sent to the device.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OFF</td>
<td>0</td>
<td>PLL is off unless PLLNDIV = 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ON</td>
<td>1</td>
<td>PLL is on regardless of the PLLNDIV bit status.</td>
</tr>
</tbody>
</table>

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

Legend: R/W-x = Read/Write-Reset value
Table A–40. Clock Mode Register (CLKMD) Field Values
(PLL_CLKMD_field_symval) (Continued)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PLLNDIV</td>
<td></td>
<td>PLL clock generator mode select bit. Determines whether the clock generator works in PLL mode or in divider (DIV) mode, thus defining the frequency multiplier in conjunction with the PLLMUL and PLLDIV bits.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>OFF</td>
<td>0</td>
<td>DIV mode is used.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ON</td>
<td>1</td>
<td>PLL mode is used.</td>
</tr>
<tr>
<td>0</td>
<td>PLLSTATUS</td>
<td></td>
<td>This read-only bit indicates the mode that the clock generator is operating.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>Divider (DIV) mode</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>PLL mode</td>
<td></td>
</tr>
</tbody>
</table>
A.7 Timer Registers

A.7.1 Timer Control Register (TCR)

Figure A–42. Timer Control Register (TCR)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-12</td>
<td>reserved</td>
<td></td>
<td></td>
<td>Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.</td>
</tr>
<tr>
<td>11</td>
<td>SOFT</td>
<td></td>
<td></td>
<td>Used in conjunction with FREE bit to determine the state of the timer when a breakpoint is encountered in the HLL debugger. When FREE bit is cleared, SOFT bit selects the timer mode.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BRKPTNOW</td>
<td>0</td>
<td>The timer stops immediately.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>WAITZERO</td>
<td>1</td>
<td>The timer stops when the counter decrements to 0.</td>
</tr>
<tr>
<td>10</td>
<td>FREE</td>
<td></td>
<td></td>
<td>Used in conjunction with SOFT bit to determine the state of the timer when a breakpoint is encountered in the HLL debugger. When FREE bit is cleared, SOFT bit selects the timer mode.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>WITHSOFT</td>
<td>0</td>
<td>SOFT bit selects the timer mode.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NOSOFT</td>
<td>1</td>
<td>The timer runs free regardless of SOFT bit status.</td>
</tr>
<tr>
<td>9-6</td>
<td>PSC</td>
<td></td>
<td></td>
<td>Timer prescalar counter. This read-only bit specifies the count for the on-chip timer when in direct mode (PREMD bit is cleared in the TSCR). When PSC bit is decremented past 0 or the timer is reset, PSC bit is loaded with the contents of TDDR bit and the TIM is decremented.</td>
</tr>
<tr>
<td>5</td>
<td>TRB</td>
<td></td>
<td></td>
<td>Timer reload bit. TRB bit is always read as a 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NORESET</td>
<td>0</td>
<td>The on-chip timer is not reset.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RESET</td>
<td>1</td>
<td>The on-chip timer is reset. When TRB bit is set, the TIM is loaded with the value in the PRD and PSC bit is loaded with the value in TDDR bit when in direct mode (PREMD bit is cleared in the TSCR).</td>
</tr>
</tbody>
</table>

Legend: R/W-x = Read/Write-Reset value
### Table A–41. Timer Control Register (TCR) Field Values

**TIMER_TCR_field_symval** (Continued)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>TSS</td>
<td></td>
<td></td>
<td>Timer stop status bit. Stops or starts the on-chip timer. At reset, TSS bit is cleared and the timer immediately starts timing.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>START</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>STOP</td>
<td>1</td>
</tr>
<tr>
<td>3-0</td>
<td>TDDR</td>
<td></td>
<td></td>
<td>The timer prescalar for the on-chip timer.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>OF(value)</td>
<td>0-15</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>OF(value)</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0001</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0011</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0100</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0101</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0110</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0111</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1001</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1011</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1100</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1101</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1110</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1111</td>
</tr>
</tbody>
</table>
### A.7.2 Timer Secondary Control Register (TSCR)

**Figure A–43. Timer Secondary Control Register (TSCR) — C5440, C5441, and C5471**

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-13</td>
<td>reserved</td>
<td></td>
<td></td>
<td>Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.</td>
</tr>
<tr>
<td>12</td>
<td>PREMD</td>
<td></td>
<td></td>
<td>Prescalar mode select bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DIRECT</td>
<td>0</td>
<td>Direct mode. When PSC bit in TCR is decremented past 0, PSC bit is loaded with TDDR content in TCR.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INDIRECT</td>
<td>1</td>
<td>Indirect mode. When PSC bit in TCR is decremented past 0, PSC bit is loaded with the prescalar value associated with TDDR bit in TCR.</td>
</tr>
<tr>
<td>11-0</td>
<td>reserved</td>
<td></td>
<td></td>
<td>Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.</td>
</tr>
</tbody>
</table>

### A.7.3 Timer Period Register (PRD)

**Figure A–44. Timer Period Register (PRD)**

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-0</td>
<td>PRD</td>
<td>0F(value)</td>
<td>0-FFFFh</td>
<td>Used to store a 16-bit value that is used to load the timer register (TIM). Set to FFFFh at reset.</td>
</tr>
</tbody>
</table>
A.8 Watchdog Timer Registers (C5441)

A.8.1 Watchdog Timer Control Register (WDTCR)

Figure A–45. Watchdog Timer Control Register (WDTCR)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-12</td>
<td>reserved</td>
<td></td>
<td></td>
<td>Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.</td>
</tr>
<tr>
<td>11</td>
<td>SOFT</td>
<td></td>
<td></td>
<td>Used in conjunction with FREE bit to determine the state of the watchdog timer when a breakpoint is encountered in the HLL debugger. When FREE bit is cleared, SOFT bit selects the watchdog timer mode.</td>
</tr>
<tr>
<td></td>
<td>BRKPTNOW</td>
<td>0</td>
<td></td>
<td>The watchdog timer stops immediately.</td>
</tr>
<tr>
<td></td>
<td>WAITZERO</td>
<td>1</td>
<td></td>
<td>The watchdog timer stops when the counter decrements to 0.</td>
</tr>
<tr>
<td>10</td>
<td>FREE</td>
<td></td>
<td></td>
<td>Used in conjunction with SOFT bit to determine the state of the watchdog timer when a breakpoint is encountered in the HLL debugger. When FREE bit is cleared, SOFT bit selects the watchdog timer mode.</td>
</tr>
<tr>
<td></td>
<td>WITHSOFT</td>
<td>0</td>
<td></td>
<td>SOFT bit selects the watchdog timer mode.</td>
</tr>
<tr>
<td></td>
<td>NOSOFT</td>
<td>1</td>
<td></td>
<td>The watchdog timer runs free regardless of SOFT bit status.</td>
</tr>
<tr>
<td>9-6</td>
<td>PSC</td>
<td></td>
<td></td>
<td>Timer prescalar counter. This read-only bit specifies the count for the on-chip watchdog timer when in direct mode (PREMD bit is cleared in the WDTSCR). When PSC bit is decremented past 0 or the watchdog timer is reset, PSC bit is loaded with the contents of TDDR bit and the WDTIM is decremented.</td>
</tr>
<tr>
<td>5-4</td>
<td>reserved</td>
<td></td>
<td></td>
<td>Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.</td>
</tr>
</tbody>
</table>
### Table A–44. Watchdog Timer Control Register (WDTCR) Field Values
(WDTIM\_WDTCR\_field\_symval) (Continued)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-0</td>
<td>TDDR</td>
<td></td>
<td></td>
<td>The timer prescalar for the on-chip watchdog timer.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>In prescalar direct mode (PREMD = 0 in WDTSCR):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>OF(value) 0-15</td>
<td>This value specifies the prescalar count for the on-chip watchdog timer. When PSC bit is decremented past 0, PSC bit is loaded with this TDDR content.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>In prescalar indirect mode (PREMD = 1 in WDTSCR):</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>OF(value)</td>
<td>This value relates to an indirect prescalar count, up to 65535, for the on-chip watchdog timer. When PSC bit is decremented past 0, PSC bit is loaded with this prescalar value.</td>
</tr>
<tr>
<td>0000</td>
<td></td>
<td></td>
<td>Prescalar value: 0001h</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td></td>
<td></td>
<td>Prescalar value: 0003h</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td></td>
<td></td>
<td>Prescalar value: 0007h</td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td></td>
<td></td>
<td>Prescalar value: 000Fh</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td></td>
<td></td>
<td>Prescalar value: 001Fh</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td></td>
<td></td>
<td>Prescalar value: 003Fh</td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td></td>
<td></td>
<td>Prescalar value: 007Fh</td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td></td>
<td></td>
<td>Prescalar value: 00FFh</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td></td>
<td>Prescalar value: 01FFh</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td></td>
<td></td>
<td>Prescalar value: 03FFh</td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td></td>
<td></td>
<td>Prescalar value: 07FFh</td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td></td>
<td></td>
<td>Prescalar value: 0FFFh</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td></td>
<td></td>
<td>Prescalar value: 1FFFh</td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td></td>
<td></td>
<td>Prescalar value: 3FFFh</td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td></td>
<td></td>
<td>Prescalar value: 7FFFh</td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td></td>
<td></td>
<td>Prescalar value: FFFFh</td>
<td></td>
</tr>
</tbody>
</table>
A.8.2 Watchdog Timer Secondary Control Register (WDTSCR)

Figure A-46. Watchdog Timer Secondary Control Register (WDTSCR)

<table>
<thead>
<tr>
<th>Bit</th>
<th>field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>WDFLAG</td>
<td></td>
<td></td>
<td>Watchdog timer flag bit. This bit can be cleared by enabling the watchdog timer, by a device reset, or by being written with a 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TIMEOUT 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NOTIMEOUT</td>
</tr>
<tr>
<td>14</td>
<td>WDEN</td>
<td></td>
<td></td>
<td>Watchdog timer enable bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DISABLE 0</td>
<td>Watchdog timer is disabled. Watchdog timer output pin is disconnected from the watchdog timer time-out event and the counter starts to run.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ENABLE 1</td>
<td>Watchdog timer is enabled. Watchdog timer output pin is connected to the watchdog timer time-out event. Watchdog timer can be disabled by a watchdog timer time-out event or by a device reset.</td>
</tr>
<tr>
<td>13</td>
<td>reserved</td>
<td></td>
<td></td>
<td>Reserved. The reserved bit location is always read as zero. A value written to this field has no effect.</td>
</tr>
<tr>
<td>12</td>
<td>PREMD</td>
<td></td>
<td></td>
<td>Prescalar mode select bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DIRECT 0</td>
<td>Direct mode. When PSC bit in WDTCR is decremented past 0, PSC bit is loaded with TDDR content in WDTCR.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>INDIRECT 1</td>
<td>Indirect mode. When PSC bit in WDTCR is decremented past 0, PSC bit is loaded with the prescalar value associated with TDDR bit in WDTCR.</td>
</tr>
<tr>
<td>11-0</td>
<td>WDKEY</td>
<td></td>
<td></td>
<td>Watchdog timer reset key. A 12-bit value that before a watchdog timer times out, only a write sequence of a 5C6h followed by an A7Eh services the watchdog timer. Any other writes triggers a watchdog timer time-out event immediately.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PREACTIVE 5C6h</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ACTIVE A7Eh</td>
<td></td>
</tr>
</tbody>
</table>
A.8.3 Watchdog Timer Period Register (WDPRD)

Figure A–47. Watchdog Timer Period Register (WDPRD)

Table A–46. Watchdog Timer Period Register (WDPRD)

<table>
<thead>
<tr>
<th>Bit field</th>
<th>symval</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-0 PRD</td>
<td>0F(value)</td>
<td>0-FFFFh</td>
<td>Stores a value that is used to reload the Watchdog timer counter register. (WDTIM)</td>
</tr>
</tbody>
</table>
A
accessing data
as defined under the CSL GUI 2-11
header file mytimercfg.h 2-12
source file mytimercfg.c 2-15
adding a .cdb file to your project 2-7
adding a linker command file associated with the .cdb file 2-7
advanced A and B pages 6-33
API modules, illustration of 1-3
architecture, of the CSL 1-2
autoinit 6-33

B
bank-switching control register (BSCR) A-17
breakpoint emulation behavior 6-33
BSCR (bank-switching control register) A-17
build options
defining a target device 2-21
defining far mode 2-22
defining large memory model 2-23
defining library paths 2-24

C
C code generation for DMA module 6-37
header file 6-37
source file 6-38
C code generation for GPIO module 8-8
header file 8-8
source file 8-8
C code generation for McBSP module 11-47
header file 11-47
source file 11-48
C code generation for PLL module 12-11
header file 12-11
source file 12-11
C code generation for TIMER 14-16
C code generation for TIMER module
header file 14-16
source file 14-16
C code generation for UART module 15-24
header file 15-24
source file 15-25
C code generation for WATCHDOG TIMER module, 16-11
header file 16-11
source file 16-12
CCStudio project view, illustration of 2-9
CCStudio project window, illustration of 2-8
CHIP, functions 3-2
CHIP functions
CHIP_getCpuld 3-3
CHIP_getMapMode 3-3
CHIP_getRevld 3-4
CHIP_getSubsysld 3-4
CHIP module
device-specific support 1-17
functions 3-3
include file 1-4
module support symbol 1-4
overview 3-2
chip support library 1-2
CLKMD (clock mode register) A-47
clock mode register (CLKMD) A-47
Code Composer Studio
defining a target device 2-21
defining far mode 2-22
defining large memory model 2-23
defining library paths 2-24
code section 2-13
compiling, with CSL 2-19
Index

compiling and linking with CSL: 2-19
configuration structure, WDTIM: 16-3
configuring peripherals using CSL GUI
  DMA module: 6-32
  McBSP module: 11-42
  PLL module: 12-8
  TIMER module: 14-12
  WATCHDOG TIMER module: 16-8
configuring peripherals using CSL GUI, GPIO module: 8-7
configuring TIMER1: 2-9
  illustration of: 2-10
constant values for fields: 1-14
constant values for registers: 1-14
CSL
  architecture: 1-2
  benefits of: 1-2
  data types: 1-7
  device support: 1-5
  functions: 1-8
  introduction to: 1-2
  macros: 1-11
  generic: 1-12
    macros, generic, handle-based: 1-13
    modules and include files: 1-4
    naming conventions: 1-6
    symbolic constants: 1-14
    using interrupts with: 10-7
CSL, generic functions: 1-9
CSL
  compiling and linking with: 2-19
  configuration methods: 2-2
  configuring without the use of the CSL GUI: 2-17
  destination address: 2-17
  directory structure: 2-20
  how to use
    introduction to the DSP/BIOS configuration tool: 2-3
    overview: 2-2
  linking with: 2-19
  rebuilding CSL: 2-26
  source address: 2-17
  transfer size: 2-17
CSL GUI
  accessing data defined in files under: 2-11
  configuring a peripheral, basic workflow: 2-7
  configuring peripherals without the use of: 2-17
  how to use: 2-3
  introduction to: 2-3
CSL bool. See data types
CSL tree (for 5510), illustration of: 2-3
CSL tree (for C5510), expanded, illustration of: 2-4
CSL_cfgInit: 2-13
  6-39, 8-8, 11-49, 12-12, 15-25
CSL_init: 2-25
  .csldata, allocation of: 2-25

D

DAA Module
  Configuration Structures: 4-4
    DAA_CircBufCtrl: 4-4
    DAA_DevSetup: 4-6
    DAA_Params: 4-4
    DAA_PrivateObject/DAA_Handle: 4-5
    DAA_Setup: 4-6
  Functions: 4-7
    DAA_availability: 4-9
    DAA_delay: 4-10
    DAA_Handle: 4-8
    DAA_isr: 4-10
    DAA_readWrite: 4-9
    DAA_reset: 4-8
    DAA_resetInternalBuffer: 4-9
    DAA_setup: 4-7
  Macros: 4-11
    DAA_ADDR: 4-12
    DAA_FMK: 4-12
    DAA_RGETH: 4-11
    DAA_RSETH: 4-11
  Overview: 4-2
DAT: 5-2
  functions: 5-3
  module support symbol: 1-4
DAT functions
  DAT_close(): 5-4
  DAT_copy(): 5-4
  DAT_copy2D(): 5-6
  DAT_fill(): 5-8
  DAT_open(): 5-9
  DAT_wait(): 5-10
DAT module
  functions: 5-4
  include file: 1-4
  overview: 5-2
DAT_close(): 5-2
DAT_open(): 5-2
DAT_wait(): 5-2
data types: 1-7
declaration list, variables handle and configuration names, 2-11
declaration section, description of, 2-13
delete/rename options, illustration of, 2-5
device-specific features, support for, 1-17
device Support symbols, 1-5
device support, 1-5
devices. See CSL device support
direct register initialization, 1-8
directory structure, 2-20
documentation, 2-20
examples, 2-20
include files, 2-20
libraries, 2-20
source library, 2-20
DMA, configuration structure, 6-2
DMA channel n destination address register (DMDSTn), A-10
DMA channel n element count register (DMCTRn), A-11
DMA channel n source address register (DMMCEN), A-9
DMA channel n sync event and frame count register (DMSFCn), A-3
DMA channel n transfer mode control register (DMMCRn), A-6
DMA channel priority and enable control register (DMPREC), A-2
DMA Configuration Manager, address formats, 6-34
DMA configuration manager, 6-32
configuring the object properties, 6-33
creating/inserting a configuration, 6-32
deleting/renameing an object, 6-33
description of, 6-32
DMA configuration structure
DMA_Config, 6-4
DMA_GblConfig, 6-5
DMA destination program page address register (DMDSTP), A-11
DMA element address index register 0 (DMIDX0), A-13
DMA element address index register 1 (DMIDX1), A-13
DMA frame address index register 0 (DMFR10), A-14
DMA frame address index register 1 (DMFR11), A-14
DMA functions
DMA_autostart, 6-18
DMA_close, 6-7
DMA_config, 6-7
DMA_configArgs, 6-8
DMA_getConfig, 6-18
DMA_getEventId, 6-19
DMA_getStatus, 6-19
DMA_globalAlloc, 6-12
DMA_globalConfig, 6-14
DMA_globalConfigArgs, 6-15
DMA_globalFree, 6-16
DMA_globalGetConfig, 6-17
DMA_open, 6-10
DMA_pause, 6-11
DMA_reset, 6-11
DMA_resetGbl, 6-17
DMA_start, 6-11
DMA_stop, 6-12
primary summary, 6-2
DMA global destination address reload register (DMGDA), A-10
DMA global element count reload register (DMGCR), A-12
DMA global extended destination data page register (DMDSTP), A-15
DMA global extended source data page register (DMSRCDP), A-15
DMA global frame count reload register (DMGFR), A-12
DMA global source address reload register (DMGSA), A-9
DMA header file, example of, 6-37
DMA Macros
DMA_ADDR, 6-28
DMA_RGETH, 6-28
DMA macros
DMA_ADDRH, 6-31
DMA_FGET, 6-26
DMA_FGETH, 6-29
DMA_FMK, 6-25
DMA_FSET, 6-30
DMA_FSETH, 6-29
to read a register address, 6-20
to read a register address (using handles), 6-21
Index

DMA module
- C code generation for, 6-37
- channel reload support, 1-17
- configuration manager, 6-32
- configuration structure, 6-4
- configuring using CSL GUI, 6-32
- extended data reach, 1-17
- functions, 6-7
- include file, 1-4
- initialization examples, 6-40
  - using DMA_config, 6-41
- macros, 6-20
  - using channel number, 6-20
- module support symbol, 1-4
- overview, 6-2, 6-32
- resource manager, 6-35

DMA properties page
- illustration of, 6-34
- with handle object accessible, illustration of, 6-36

DMA registers
- channel n destination address register (DMDSnT), A-10
- channel n element count register (DMCTrn), A-11
- channel n source address register (DMSRCn), A-9
- channel n sync event and frame count register (DMSFSCn), A-3
- channel n transfer mode control register (DMMCRN), A-6
- channel priority and enable control register (DMPREC), A-2
- destination program page address register (DMDSP), A-11
- element address index register 0 (DMIDX0), A-13
- element address index register 1 (DMIDX1), A-13
- frame address index register 0 (DMFRI0), A-14
- frame address index register 1 (DMFRI1), A-14
- global destination address reload register (DMGDA), A-10
- global element count reload register (DMGCR), A-12
- global extended destination data page register (DMDSFDP), A-15
- global extended source data page register (DMSSFCP), A-15
- global frame count reload register (DMGFR), A-12
- global source address reload register (DMGSA), A-9
- source program page address register (DMSSCP), A-9

DMA resource manager, 6-35
- description of, 6-32
- predefined objects, 6-35
- properties page, 6-35

DMA resource manager menu, illustration of, 6-35

DMA sections menu, illustration of, 6-32

DMA source file, example of, 6-38, 6-39

DMA source program page address register (DMSSCP), A-9

DMA_AdrPtr, 1-7

DMA_config, 6-38
- using, 2-18

DMA_open(), 6-38

DMA0, 6-35

DMA1, 6-35

DMA2, 6-35

DMA3, 6-35

DMA4, 6-35

DMA5, 6-35

DMDSnT (channel n destination address register), A-10

DMDSFDP (DMA global extended destination data page register), A-15

DMDSnT (DMA channel n destination address register), A-10

DMDSnP (DMA destination program page address register), A-11

DMFRI0 (DMA frame address index register 0), A-14

DMFRI1 (DMA frame address index register 1), A-14

DMGCR (DMA global element count reload register), A-12

DMGDA (DMA global destination address reload register), A-10

DMGFR (DMA global frame count reload register), A-12
DMGSA (DMA global source address reload register), A-9
DMIDX0 (DMA element address index register 0), A-13
DMIDX1 (DMA element address index register 1), A-13
DMMCRn (DMA channel n transfer mode control register), A-6
DMPREC (DMA channel priority and enable control register), A-2
DMSFCan (DMA channel n sync event and frame count register), A-3
DMSRCn (DMA channel n source address register), A-9
DMSRCP (DMA source program page address register), A-9
documentation. See directory structure

E
EBUS
configuration structure, 7-2
functions, 7-2
macros, 7-6
EBUS configuration structure, EBUS_Config, 7-3
EBUS functions
EBUS_config, 7-4
EBUS_configArgs, 7-5
for C544X devices, 7-5
for C54X devices, 7-5
EBUS module
configuration structure, 7-3
functions, 7-4
include file, 1-4
macros, 7-6
module support symbol, 1-4
overview, 7-2
examples. See directory structure

F
far–mode library. See CSL device support
far/near mode selection, instructions, 2-21
FIELD, explanation of, 1-11
fieldval, explanation of, 1-11
funcArg. See naming conventions
function, naming conventions, 1-6
function argument, naming conventions, 1-6
function inlining, using, 2-25
generics, 1-8
WDTIM, 16-4

G
general purpose I/O control register (GPIOCR), A-22, A-24
general purpose I/O registers, A-22, A-24
general purpose I/O control register (GPIOCR), A-22, A-24
general purpose I/O status register (GPIOCR), A-25
general purpose I/O status register (GPIOSR), A-25
generation of the C files
header files, 2-11
source files, 2-11
generic CSL functions, 1-9
GPIO, macros, 8-5
GPIO configuration manager, 8-7
description of, 8-7
GPIO functions, listing of, 8-2
GPIO module
classing using CSL GUI, 8-7
functions, 8-3
include file, 1-4
macros, 8-5
module support symbol, 1-4
overview, 8-2
GPIO properties page, illustration of, 8-7
GPIO source file (body section), example of, 8-8
GPIO_pinDirection, 8-3
GPIO_pinRead, 8-3
GPIO_pinWrite, 8-4
GPIO_RST(), 8-8
GPIOCR (general purpose I/O control register), A-22, A-24
GPIOCSR (general purpose I/O status register), A-25
graphical user interface. See CSL GUI
GUI. See CSL GUI
guidelines, EBUS, 7-2

Index-5
**H**

handles
resource management: 1-15
use of: 1-15
header file, mytimercfg.h: 2-11
header file mytimercfg.h. See accessing data
HPI, macros: 9-2
HPI module
include file: 1-4
macros: 9-2
module support symbol: 1-4

**I**

include Files. See directory structure
include files, for CSL modules: 1-4
include section, description of: 2-12
inserting a configuration object, illustration of: 2-5
Int16: 1-7
Int32: 1-7
interrupts
See also IRQ module
eexample of a manual interrupt setting outside
DSP/BIOS HWIs: 10-7
using with CSL: 10-7
IRQ
configuration structure: 10-3
functions: 10-3
IRQ configuration structure, IRQ_Config: 10-3, 10-8
IRQ functions
IRQ_clear: 10-9
IRQ_config: 10-9
IRQ_configArgs: 10-10
IRQ_disable: 10-10
IRQ_enable: 10-11
IRQ_getArg: 10-11
IRQ_getConfig: 10-12
IRQ_globalDisable: 10-12
IRQ_globalEnable: 10-13
IRQ_globalRestore: 10-13
IRQ_map: 10-14
IRQ_plug: 10-14
IRQ_restore: 10-15
IRQ_setArg: 10-15
IRQ_setVecs: 10-16
IRQ_test: 10-16

**L**

large/small memory model selection, instructions, 2-23
libraries
See also directory structure
linking to a project: 2-22, 2-24
linker command file
creating. See compiling and linking with CSL using: 2-25
linking, with CSL: 2-19

**M**

macro, naming conventions: 1-6
macros, generic: 1-12
handle-based: 1-13
macros, generic description of
CSL: 1-11
FIELD: 1-11
fieldval: 1-11
PER: 1-11
REG: 1-11
REG#: 1-11
regval: 1-11
main.c file, example of. See using data generated by
the DSP/BIOS configuration tool
McBSP
configuration manager: 11-42
configuration structure: 11-2
functions: 11-2
macros using handle: 11-25
macros using port number: 11-24
McBSP configuration manager: 11-42
configuring the object properties: 11-43
creating/inserting a configuration object: 11-42
deleting/renaming an object: 11-43
McBSP configuration structure, McBSP_Config, 11-4
Index

N
naming conventions, 1-6
near-mode library. See CSL device support

O
object types. See naming conventions

P
PCR (pin control register), A-30
PER, 1-14
   explanation of, 1-11
PER_ADDR, 1-12
PER_ADDRH, 1-13
PER_close, 1-9
PER_config, 1-9
   initialization of registers, 1-10
PER_config(), 1-9
PER_configArgs, 1-9
   initialization of registers, 1-10
PER_configArgs(), 1-9
PER_FGET, 1-12
PER_FGETH, 1-13
PER_FMK, 1-12
PER_FSET, 1-12
PER_FSETH, 1-13
PER_funcName(). See naming conventions
PER_Handle, 1-7
PER_MACRO_NAME. See naming conventions
PER_open, 1-9
PER_open(), 1-9
PER_REG_DEFAULT, 1-14
PER_REG_FIELD_DEFAULT, 1-14
PER_REG_FIELD_SYMVAL, 1-14
PER_REG_RMK, 1-12
PER_reset, 1-9
PER_RGET, 1-12
PER_RGETH, 1-13
PER_RSET, 1-12
PER_RSETH, 1-13
PER_start, 1-9
PER_TypeDef. See naming conventions
PER_varName(). See naming conventions
peripheral configuration
   basic workflow, 2-7
   using CSL GUI, 2-7
peripheral configuration manager, description of, 2-4
peripheral initialization via registers, 1-10
   using PER_config or PER_configArgs, 1-10
peripheral modules
   descriptions of, 1-4
   include files, 1-4
pin control register (PCR), A-30
PLL, macros, 12-6
PLL
   configuration structure, 12-2
   functions, 12-2
PLL module, overview, 12-2
PLL configuration manager, 12-8
   configuring the object properties, 12-9
   creating/inserting a configuration, 12-8
   deleting/renameing an object, 12-9
   description of, 12-6
PLL configuration structure, PLL_Config, 12-3
PLL functions
   PLL_config, 12-4
   PLL_configArgs, 12-4
   PLL_setFreq, 12-5
PLL header file, example of, 12-11
PLL macros
   PLL_ADDR, 12-6
   PLL_FGET, 12-6
   PLL_FMK, 12-6
   PLL_FSET, 12-6
   PLL_REG_RMK, 12-6
   PLL_RGET, 12-6
   PLL_RSET, 12-6
PLL module
   configuration structure, 12-3
   configuring using the CSL GUI, 12-8
   functions, 12-4
   include file, 1-4
   macros, 12-6
   module support symbol, 1-4
PLL properties page, illustration of, 12-9
PLL resource manager, 12-10
   description of, 12-8
   properties page, 12-10
PLL resource manager menu, illustration of, 12-10

Index-8
PLL sections menu, illustration of, 8-7, 12-8
PLL source file (body section), example of, 12-12
PLL source file (declaration section), example of, 12-11
PLL_config, 12-12
predefined objects, 11-46
DMA0, 6-35
DMA1, 6-35
DMA2, 6-35
DMA3, 6-35
DMA4, 6-35
DMA5, 6-35
McBSP0, 11-46
McBSP1, 11-46
McBSP2, 11-46
TIMER, 14-14
TIMER0, 14-14
TIMER1, 14-14
project.cdb, 2-11
project_cfg.h, 2-11
projectcfg_c.c, 2-11
properties page
TIMER, 14-15
WATCHDOG TIMER, 16-10
properties pages
advanced A and B pages, 6-33
autoinit, 6-33
breakpoint emulation behavior, 6-33
source/destination, 6-33
properties pages of the non-multiplexed GPIO configuration, GPIO, 8-7
PWR, functions, 13-2
PWR functions, PWR_powerDown, 13-2, 13-3
PWR module
functions, 13-3
include file, 1-4
module support symbol, 1-4
overview, 13-2
REG, 1-14
explanation of, 1-11
REG#, explanation of, 1-11
registers, peripheral initialization, 1-10
regval, explanation of, 1-11
resource management, using CSL handles, 1-15
resource manager
McBSP, 11-46
TIMER, 14-14
WATCHDOG TIMER, 16-10
resource manager properties page, illustration of, 2-14
S
sample rate generator register 1 (SRGR1), A-38
sample rate generator register 2 (SRGR2), A-38
scratch pad memory, 2-25
serial port control register 1 (SPCR1), A-26
serial port control register 2 (SPCR2), A-28
show dependency option, illustration of, 2-6
software wait-state control register (SWCR), A-17
software wait-state register (SWWSR), A-16
software wait-state registers, A-16
software wait-state control register (SWCR), A-17
software wait-state register (SWWSR), A-16
source file
source file, 11-48
mytimercfg_c.c, 2-12
source file mytimercfg.c. See accessing data
source file projectcfg_c.c
code section, 2-13
declaration section, 2-13
include section, 2-12
source library. See directory structure
source/destination, 6-33
SPCR1 (serial port control register 1), A-26
SPCR2 (serial port control register 2), A-28
SRGR1 (sample rate generator register 1), A-38
SRGR2 (sample rate generator register 2), A-38
static inline. See function inlining
structure member, naming conventions, 1-6
SWCR (software wait-state control register), A-17
SWWSR (software wait-state register), A-16
symbolic constant values, 1-14
symbolic constants, generic, 1-14
SYMVAL, 1-14
target device, specifying. See compiling and linking with CSL

TCR (timer control register), A-49

TIMER
  C code generation for, 14-16
  configuration structure, 14-2
  functions, 14-2
  Timer configuration manager, 14-12
    configuring the object properties, 14-13
    creating/inserting a configuration, 14-12
    deleting/renaming an object, 14-13
    description of, 14-12
  Timer configuration structure, TIMER_Config, 14-3
    for C5440, C541, and C5472 devices only, 14-3
  timer control register (TCR), A-49
  TIMER functions
    TIMER_close, 14-4
    TIMER_config, 14-4
    TIMER_configArgs, 14-5
    TIMER_getEventID, 14-6
    TIMER_open, 14-6
    TIMER_reload, 14-7
    TIMER_reset, 14-7
    TIMER_start, 14-7
    TIMER_stop, 14-8
  timer header file, example of, 14-16
  TIMER macros
    TIMER_ADDR, 14-10
    TIMER_ADDRH, 14-11
    TIMER_FGET, 14-10
    TIMER_FGETH, 14-11
    TIMER_FMK, 14-10
    TIMER_FSET, 14-10
    TIMER_FSETH, 14-11
    TIMER_REG_RMK, 14-10
    TIMER_RGET, 14-10
    TIMER_RGETH, 14-11
    TIMER_RSET, 14-10
    TIMER_RSETH, 14-11
    using handle, 14-11
    using timer port number, 14-10
  TIMER module, 14-12
    configuration structure, 14-3
    configuring using the CSL GUI, 14-12
    device-specific support, 1-17
    functions, 14-4

  include file, 1-4
  macros, 14-9
  module support symbol, 1-4
  overview, 14-2
  TIMER properties page, illustration of, 14-14
  timer properties page with handle object accessible, illustration of, 14-15
  timer registers, A-49
    timer control register (TCR), A-49
    timer secondary control register (TSCR), A-51
    A-55
  TIMER resource manager, 14-14
    description of, 14-12
  timer resource manager menu, illustration of, 14-14
  timer secondary control register (TSCR), A-51, A-55
  timer sections menu, illustration of, 14-12
  timer source file (body section), example of, 14-17
  timer source file (declaration section), example of, 14-16
  TIMER0, 14-14
  TIMER1, 14-14
    configuration of, 2-9
  transfer modes, 6-33
  transmit channel enable register (XCRn), A-45
  transmit control register 1 (XCR1), A-35
  transmit control register 2 (XCR2), A-36
  TSCR (timer secondary control register), A-51, A-55
  typedef, naming conventions, 1-6

UART
  C code generation for, 15-24
  Control Signal Macros, 15-16
  UART configuration manager, 15-21
    configuring the object properties, 15-22
    creating/inserting a configuration object, 15-21
    deleting/renaming an object, 15-22
  UART Functions
    UART_config, 15-8
    UART_configArgs, 15-8
    UART_eventDisable, 15-8
    UART_eventEnable, 15-9
    UART_fgetc, 15-11
    UART_fgets, 15-11
    UART_fputc, 15-12
    UART_fputs, 15-12
    UART_getConfig, 15-12
Index

WDTIM configuration structure, WDTIM_Config, 16-3
WDTIM functions
  WDTIM_config, 16-4
  WDTIM_configArgs, 16-4
  WDTIM_getConfig, 16-5
  WDTIM_service, 16-5
  WDTIM_start, 16-5
WDTIM macros
  using timer port number, 16-7
  WDTIM_ADDR, 16-7
  WDTIM_FGET, 16-7
  WDTIM_FMk, 16-7
  WDTIM_FSET, 16-7

WDTIM_REG_RMK, 16-7
WDTIM_RGET, 16-7
WDTIM_RSET, 16-7
WDTIM module, overview, 16-2
WDTIMER_config, 16-12
WDTSCR (watchdog timer secondary control register), A-54

X

XCELER (transmit channel enable register), A-45
XCR1 (transmit control register 1), A-35
XCR2 (transmit control register 2), A-36