Copyright Information

© 2013 Analog Devices, Inc., ALL RIGHTS RESERVED. This document may not be reproduced in any form without prior, express written consent from Analog Devices, Inc.

Printed in the USA.

Disclaimer

Analog Devices, Inc. reserves the right to change this product without prior notice. Information furnished by Analog Devices is believed to be accurate and reliable. However, no responsibility is assumed by Analog Devices for its use; nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under the patent rights of Analog Devices, Inc.

Trademark and Service Mark Notice

The Analog Devices logo, Blackfin, CrossCore, EngineerZone, EZ-KIT Lite, and VisualDSP++ are registered trademarks of Analog Devices, Inc.

All other brand and product names are trademarks or service marks of their respective owners.
CONTENTS

PREFACE

Purpose of This Manual ............................................................. xliii
Intended Audience ...................................................................... xliii
Manual Contents ......................................................................... xliv
What’s New in This Manual ..................................................... xlviii
Technical Support ........................................................................ xlix
Supported Processors ........................................................................ 1
Product Information ........................................................................ 1
   Analog Devices Web Site ........................................................ li
   EngineerZone ............................................................................ li
Notation Conventions .................................................................... lii
Register Diagram Conventions .................................................. liii

INTRODUCTION

Purpose of this Manual ............................................................. 1-1
Peripherals .................................................................................... 1-4
Core Architecture .......................................................................... 1-5
<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Architecture</td>
<td>1-9</td>
</tr>
<tr>
<td>Internal Memory</td>
<td>1-10</td>
</tr>
<tr>
<td>External Memory</td>
<td>1-10</td>
</tr>
<tr>
<td>I/O Memory Space</td>
<td>1-11</td>
</tr>
<tr>
<td>Event Handling</td>
<td>1-11</td>
</tr>
<tr>
<td>Core Event Controller (CEC)</td>
<td>1-12</td>
</tr>
<tr>
<td>System Interrupt Controllers (SICx)</td>
<td>1-13</td>
</tr>
<tr>
<td>DMA Support</td>
<td>1-13</td>
</tr>
<tr>
<td>External Bus Interface Unit</td>
<td>1-14</td>
</tr>
<tr>
<td>PC133 SDRAM Controller</td>
<td>1-14</td>
</tr>
<tr>
<td>Asynchronous Controller</td>
<td>1-15</td>
</tr>
<tr>
<td>Parallel Peripheral Interface</td>
<td>1-15</td>
</tr>
<tr>
<td>General-Purpose Mode Descriptions</td>
<td>1-16</td>
</tr>
<tr>
<td>Input Mode</td>
<td>1-16</td>
</tr>
<tr>
<td>Frame Capture Mode</td>
<td>1-16</td>
</tr>
<tr>
<td>Output Mode</td>
<td>1-17</td>
</tr>
<tr>
<td>ITU-R 656 Mode Descriptions</td>
<td>1-17</td>
</tr>
<tr>
<td>Active Video Only Mode</td>
<td>1-17</td>
</tr>
<tr>
<td>Vertical Blanking Interval Mode</td>
<td>1-17</td>
</tr>
<tr>
<td>Entire Field Mode</td>
<td>1-18</td>
</tr>
<tr>
<td>Serial Ports (SPORTs)</td>
<td>1-18</td>
</tr>
<tr>
<td>Serial Peripheral Interface (SPI) Ports</td>
<td>1-20</td>
</tr>
<tr>
<td>Timers</td>
<td>1-21</td>
</tr>
<tr>
<td>UART Ports</td>
<td>1-21</td>
</tr>
</tbody>
</table>
CONTENTS

Controller Area Network Port ..................................................... 1-23
Two-Wire Interface Port .............................................................. 1-23
Real-Time Clock ......................................................................... 1-23
Watchdog Timer .......................................................................... 1-24
General-Purpose I/O ................................................................... 1-25
Clock Signals .............................................................................. 1-25
Dynamic Power Management ...................................................... 1-26
   Full On Operating Mode (Maximum Performance) ............... 1-26
   Active Operating Mode (Moderate Power Savings) .......... 1-26
   Sleep Operating Mode (High Power Savings) ................. 1-27
   Deep Sleep Operating Mode (Maximum Power Savings) ....... 1-27
   Hibernate State .................................................................... 1-28
Voltage Regulation ...................................................................... 1-28
Boot Modes ................................................................................ 1-28
Instruction Set Description ......................................................... 1-30
Development Tools ..................................................................... 1-31

COMPUTATIONAL UNITS

Using Data Formats ....................................................................... 2-3
   Binary String .......................................................................... 2-3
   Unsigned ............................................................................ 2-4
   Signed Numbers: Two’s-Complement ................................ 2-4
   Fractional Representation: 1.15 ............................................. 2-4
Quad 16-Bit Operations .................................................... 2-27
Single 32-Bit Operations .................................................. 2-28
Dual 32-Bit Operations ..................................................... 2-28
ALU Instruction Summary .................................................... 2-29
ALU Data Flow Details ....................................................... 2-33
Dual 16-Bit Cross Options ............................................... 2-35
ALU Status Signals ............................................................ 2-36
ALU Division Support Features ............................................. 2-36
Special SIMD Video ALU Operations .................................. 2-37
Multiply Accumulators (Multipliers) ..................................... 2-37
Multiplier Operation .......................................................... 2-38
  Placing Multiplier Results in Multiplier Accumulator
  Registers ........................................................................ 2-39
  Rounding or Saturating Multiplier Results ......................... 2-39
Saturating Multiplier Results on Overflow  ......................... 2-40
Multiplier Instruction Summary ........................................... 2-40
  Multiplier Instruction Options ........................................ 2-42
Multiplier Data Flow Details ............................................. 2-44
Multiply Without Accumulate ........................................... 2-46
Special 32-Bit Integer MAC Instruction ............................... 2-48
Dual MAC Operations ....................................................... 2-49
Barrel Shifter (Shifter) ...................................................... 2-50
Shifter Operations ............................................................ 2-50
Contents

Two-Operand Shifts .......................................................... 2-51
Immediate Shifts .......................................................... 2-51
Register Shifts ............................................................... 2-52
Three-Operand Shifts ....................................................... 2-52
Immediate Shifts .......................................................... 2-52
Register Shifts ............................................................... 2-53
Bit Test, Set, Clear, Toggle ................................................. 2-53
Field Extract and Field Deposit ......................................... 2-54
Shifter Instruction Summary ................................................. 2-54

OPERATING MODES AND STATES

User Mode ................................................................................... 3-3
Protected Resources and Instructions ................................. 3-4
Protected Memory ............................................................... 3-5
Entering User Mode ............................................................ 3-5
Example Code to Enter User Mode Upon Reset ................. 3-5
Return Instructions That Invoke User Mode ....................... 3-6
Supervisor Mode .......................................................................... 3-7
Non-OS Environments ........................................................... 3-7
Example Code for Supervisor Mode Coming Out of Reset ... 3-8
Emulation Mode ............................................................................. 3-9
Idle State .................................................................................... 3-10
Example Code for Transition to Idle State ......................... 3-10
Reset State .................................................................................. 3-11
Contents

System Reset and Power-up .......................................................... 3-12
  Hardware Reset ...................................................................... 3-14
  SYSCR Register ................................................................... 3-14
  Software Resets and Watchdog Timer ..................................... 3-15
  SWRST Register .................................................................. 3-16
  Core-Only Software Reset .................................................... 3-17
  Core and System Reset ........................................................ 3-17
Booting Methods ........................................................................ 3-19

PROGRAM SEQUENCER

  Sequencer Related Registers .................................................. 4-4
    Sequencer Status (SEQSTAT) Register ................................. 4-4
    Zero-Overhead Loop (LCx, LTx, LBx) Registers ................. 4-5
    System Configuration (SYSCFG) Register ........................ 4-6
  Instruction Pipeline ............................................................... 4-7
  Branches and Sequencing ....................................................... 4-9
    Direct Short and Long Jumps ............................................. 4-10
    Direct Call ....................................................................... 4-11
    Indirect Branch and Call .................................................. 4-11
    PC-Relative Indirect Branch and Call ................................. 4-12
    Condition Code Flag ....................................................... 4-12
      Conditional Branches ................................................... 4-13
      Conditional Register Move ........................................... 4-14
      Branch Prediction ....................................................... 4-14
  Loops and Sequencing .......................................................... 4-15
# Contents

- Events and Sequencing ............................................................... 4-18
- System Interrupt Processing .................................................. 4-21
- System Peripheral Interrupts .................................................. 4-23
- System Interrupt Wake-Up Enable (SIC_IWRx) Registers ...... 4-27
- System Interrupt Status (SIC_ISRx) Registers ...................... 4-29
- System Interrupt Mask (SIC_IMASKx) Registers ................... 4-32
- System Interrupt Assignment (SIC_IARx) Registers ............... 4-34
- Core Event Controller Registers .................................................. 4-39
- Core Interrupt Mask (IMASK) Register ................................. 4-39
- Core Interrupt Latch (ILAT) Register .................................... 4-40
- Core Interrupts Pending (IPEND) Register ......................... 4-41
- Global Enabling/Disabling of Interrupts ............................... 4-42
- Event Vector Table ................................................................. 4-43
- Emulation ............................................................................. 4-44
- Reset .................................................................................... 4-44
- NMI (Non-Maskable Interrupt) ............................................ 4-45
- Exceptions ............................................................................ 4-46
- Exceptions While Executing an Exception Handler ............... 4-51
- Hardware Error Interrupt ...................................................... 4-52
- Core Timer ........................................................................... 4-53
- General-Purpose Interrupts (IVG7-IVG15) ............................ 4-53
- Servicing Interrupts ................................................................. 4-54
Contents

Nesting of Interrupts ................................................................. 4-55
Non-Nested Interrupts ............................................................. 4-55
Nested Interrupts ................................................................. 4-57
  Example Prolog Code for Nested Interrupt Service
    Routine ................................................................. 4-59
  Example Epilog Code for Nested Interrupt Service
    Routine ................................................................. 4-59
Logging of Nested Interrupt Requests ............................... 4-60
Exception Handling ................................................................. 4-61
  Deferring Exception Processing ...................................... 4-61
  Example Code for an Exception Handler ......................... 4-62
  Example Code for an Exception Routine ....................... 4-64
  Example Code for Using Hardware Loops in an ISR .......... 4-64
Other Usability Issues ............................................................. 4-65
  Executing RTX, RTN, or RTE in a Lower Priority Event .... 4-65
  Recommendation for Allocating the System Stack .......... 4-66
  Latency in Servicing Events ........................................... 4-66

DATA ADDRESS GENERATORS

  Addressing With DAGs ......................................................... 5-4
  Frame and Stack Pointers .................................................. 5-5
  Addressing Circular Buffers ............................................. 5-6
  Addressing With Bit-Reversed Addresses ....................... 5-9
  Indexed Addressing With Index and Pointer Registers .... 5-10
  Auto-Increment and Auto-Decrement Addressing .......... 5-11
Contents

Pre-Modify Stack Pointer Addressing ........................................ 5-11
Indexed Addressing With Immediate Offset ............................... 5-12
Post-Modify Addressing .......................................................... 5-12
Modifying DAG and Pointer Registers ........................................ 5-13
Memory Address Alignment ...................................................... 5-14
DAG Instruction Summary ....................................................... 5-17

MEMORY

Memory Architecture .................................................................. 6-1
   Overview of Internal Memory .............................................. 6-2
   Overview of Scratchpad Data SRAM .................................... 6-5
L1 Instruction Memory ........................................................... 6-5
   Instruction Memory Control (IMEM_CONTROL) Register ... 6-6
L1 Instruction SRAM .............................................................. 6-7
L1 Instruction Cache .............................................................. 6-9
   Cache Lines ........................................................................ 6-11
   Cache Hits and Misses ...................................................... 6-14
   Cache Line Fills ................................................................ 6-14
   Line Fill Buffer ................................................................ 6-15
   Cache Line Replacement .................................................... 6-15
Instruction Cache Management ............................................... 6-16
   Instruction Cache Locking by Line .................................... 6-17
   Instruction Cache Locking by Way .................................... 6-17
   Instruction Cache Invalidation .......................................... 6-18
Contents

Instruction Test Registers ............................................................ 6-20
Instruction Test Command (ITEST_COMMAND) Register ........... 6-21
Instruction Test Data (ITEST_DATA1) Register ....................... 6-22
Instruction Test Data 0 (ITEST_DATA0) Register ................. 6-23
L1 Data Memory ........................................................................ 6-24
Data Memory Control (DMEM_CONTROL) Register .......... 6-24
L1 Data SRAM ...................................................................... 6-27
L1 Data Cache ...................................................................... 6-30
  Example of Mapping Cacheable Address Space .......... 6-31
Data Cache Access ............................................................ 6-34
Cache Write Method ............................................................. 6-36
Interrupt Priority Register and Write Buffer Depth .......... 6-36
Data Cache Control Instructions ....................................... 6-37
Data Cache Invalidation .................................................... 6-38
Data Test Registers ............................................................... 6-39
Data Test Command (DTEST_COMMAND) Register .......... 6-40
Data Test Data (DTEST_DATA1) Register ......................... 6-41
Data Test Data (DTEST_DATA0) Register ......................... 6-42
External Memory ................................................................... 6-43
Memory Protection and Properties ......................................... 6-43
Memory Management Unit .................................................. 6-43
Memory Pages .................................................................... 6-45
  Memory Page Attributes .................................................. 6-45
Page Descriptor Table ........................................................... 6-47
CPLB Management ............................................................... 6-47
MMU Application .............................................................. 6-49
Examples of Protected Memory Regions ......................... 6-51
Instruction CPLB Data (ICPLB_DATAx) Registers .............. 6-52
Data CPLB Data (DCPLB_DATAx) Registers .................. 6-53
Data CPLB Address (DCPLB_ADDRx) Registers ........... 6-56
Instruction CPLB Address (ICPLB_ADDRx) Registers .... 6-57
Instruction and Data CPLB Status (ICPLB_STATUS,
DCPLB_STATUS) Registers .............................................. 6-59
Instruction and Data CPLB Fault Address
(ICPLB_FAULT_ADDR, DCPLB_FAULT_ADDR)
Registers ............................................................................ 6-60
Memory Transaction Model .................................................. 6-62
Load/Store Operation ......................................................... 6-63
Interlocked Pipeline ......................................................... 6-64
Ordering of Loads and Stores ........................................... 6-64
Synchronizing Instructions ............................................... 6-66
Speculative Load Execution ............................................. 6-67
Conditional Load Behavior ............................................. 6-68
Working With Memory ...................................................... 6-68
Alignment ............................................................................. 6-68
Cache Coherency .............................................................. 6-69
Atomic Operations ............................................................ 6-69
Contents

Memory-Mapped Registers .................................................... 6-70
Core MMR Programming Code Example ............................... 6-70
Terminology ............................................................................... 6-71

CHIP BUS HIERARCHY

Internal Interfaces ............................................................................ 7-1
Internal Clocks .................................................................................. 7-1
Core Overview .................................................................................. 7-3
System Overview .............................................................................. 7-4
System Interfaces ............................................................................. 7-4
  Peripheral Access Bus (PAB) .............................................................. 7-5
    PAB Arbitration ........................................................................ 7-5
    PAB Performance ..................................................................... 7-5
    PAB Agents (Masters, Slaves) ................................................... 7-6
DMA Access (DAB0/DAB1), Core (DCB0/DCB1),
and External Buses (DEB0/DEB1) ................................................ 7-7
  DABx, DCBx, and DEBx Arbitration ........................................ 7-7
  DAB, DCB, and DEB Performance ............................................. 7-9
  DAB Bus Agents (Masters) ......................................................... 7-10
External Access Bus (EAB) .............................................................. 7-10
  EAB Arbitration ........................................................................ 7-11
DYNAMIC POWER MANAGEMENT

Clocking ........................................................................................................... 8-1

Phase-Locked Loop and Clock Control ......................................................... 8-2

PLL Overview ............................................................................................... 8-2

PLL Clock Multiplier Ratios ......................................................................... 8-3

Core Clock/System Clock Ratio Control ....................................................... 8-4

PLL Registers ................................................................................................. 8-6

PLL Divide (PLL_DIV) Register .................................................................... 8-6

PLL Control (PLL_CTL) Register .................................................................. 8-7

PLL Status (PLL_STAT) Register .................................................................... 8-9

PLL Lock Count (PLL_LOCKCNT) Register ................................................. 8-10

Dynamic Power Management Controller ...................................................... 8-11

Operating Modes ............................................................................................ 8-11

Dynamic Power Management Controller States ............................................. 8-12

Full On Mode ................................................................................................. 8-12

Active Mode ................................................................................................... 8-13

Sleep Mode ..................................................................................................... 8-13

Deep Sleep Mode ........................................................................................... 8-14

Hibernate State ............................................................................................. 8-14

Operating Mode Transitions ........................................................................... 8-15

Programming Operating Mode Transitions ................................................... 8-18

PLL Programming Sequence .......................................................................... 8-19

PLL Programming Sequence Continues ....................................................... 8-21

Examples ......................................................................................................... 8-21
Dynamic Supply Voltage Control ............................................... 8-23
Power Supply Management ..................................................... 8-24
  Voltage Regulator Control (VR_CTL) Register .................... 8-25
  Changing Voltage .......................................................... 8-28
  Powering Down the Core (Hibernate State) ....................... 8-29

DIRECT MEMORY ACCESS

DMA and Memory DMA MMRs ................................................. 9-3
Naming Conventions for DMA MMRs ..................................... 9-5
Naming Conventions for Memory DMA Registers ....................... 9-7
  Next Descriptor Pointer (DMAx_NEXT_DESC_PTR,
     MDMAx_yy_NEXT_DESC_PTR) Registers ..................... 9-8
  Start Address (DMAx_START_ADDR,
     MDMAx_yy_START_ADDR) Registers ......................... 9-10
  DMA Configuration (DMAx_CONFIG,
     MDMAx_yy_CONFIG) Registers .............................. 9-11
  Inner Loop Count (DMAx_X_COUNT,
     MDMAx_yy_X_COUNT) Registers .......................... 9-14
  Inner Loop Address Increment (DMAx_X_MODIFY,
     MDMAx_yy_X_MODIFY) Registers ......................... 9-15
  Outer Loop Count (DMAx_Y_COUNT,
     MDMAx_yy_Y_COUNT) Registers .......................... 9-16
  Outer Loop Address Increment (DMAx_Y_MODIFY,
     MDMAx_yy_Y_MODIFY) Registers ......................... 9-17
  Current Descriptor Pointer (DMAx_CURR_DESC_PTR,
     MDMAx_yy_CURR_DESC_PTR) Registers ................... 9-17
Current Address (DMAx_CURR_ADDR, MDMAx_yy_CURR_ADDR) Registers ........................................ 9-18
Current Inner Loop Count (DMAx_CURR_X_COUNT, MDMAx_yy_CURR_X_COUNT) Registers .................... 9-19
Current Outer Loop Count (DMAx_CURR_Y_COUNT, MDMAx_yy_CURR_Y_COUNT) Registers .................. 9-20
Peripheral Map (DMAx_PERIPHERAL_MAP, MDMAx_yy_PERIPHERAL_MAP) Registers ..................... 9-21
Interrupt Status (DMAx_IRQ_STATUS, MDMAx_yy_IRQ_STATUS) Registers ........................................... 9-26
Flex Descriptor Structure .................................................................................................................... 9-28
Two-Dimensional DMA ....................................................................................................................... 9-30
DMA Operation Flow .......................................................................................................................... 9-32
DMA Startup ..................................................................................................................................... 9-35
DMA Refresh ....................................................................................................................................... 9-37
To Stop DMA Transfers ...................................................................................................................... 9-38
Software Management of DMA .......................................................................................................... 9-39
Synchronization of Software and DMA ............................................................................................... 9-40
Single-Buffer DMA Transfers .............................................................................................................. 9-42
Continuous Transfers Using Auto Buffering ...................................................................................... 9-42
Descriptor Structures .......................................................................................................................... 9-44
Descriptor Queue Management ......................................................................................................... 9-45
Descriptor Queue Using Interrupts on Every Descriptor .................................................................... 9-46
Descriptor Queue Using Minimal Interrupts ..................................................................................... 9-47
More 2D DMA Examples .................................................................................................................... 9-49
Contents

Memory DMA ................................................................. 9-50
    MDMA Bandwidth ......................................................... 9-52
    MDMA Priority and Scheduling ....................................... 9-53
DMA Controller Errors (Aborts) ........................................... 9-54
DMA Performance: Prioritization and Optimization ............... 9-57
Prioritization and Traffic Control ....................................... 9-59
    DMA Traffic Control Counter Period (DMACx_TC_PER)  
    and Counter (DMACx_TC_CNT) Registers ....................... 9-61
Urgent DMA Transfers ..................................................... 9-64

**SPI COMPATIBLE PORT CONTROLLERS**

    Interface Signals .................................................... 10-4
        Serial Peripheral Interface Clock Signals (SCKx) .......... 10-4
        Serial Peripheral Interface Slave Select Input Signals  
        (SPIxSS) ................................................................ 10-4
        Master Out Slave In (MOSIx) ..................................... 10-5
        Master In Slave Out (MISOx) .................................... 10-5
        Interrupt Output .................................................... 10-6
SPI Registers .................................................................. 10-7
    SPI BAUD Rate (SPIx_BAUD) Register .......................... 10-8
    SPI Control (SPIx_CTL) Register .................................. 10-9
    SPI Flag (SPIx_FLG) Register ....................................... 10-11
        Slave Select Inputs .................................................. 10-13
        Use of FLS Bits in SPI0_FLG for Multiple Slave 
        SPI Systems ............................................................ 10-14
        Special Considerations for SPI1 and SPI2 Slave Control ... 10-15
<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPI Status (SPIx_STAT) Register</td>
<td>10-15</td>
</tr>
<tr>
<td>SPI Transmit Data Buffer (SPIx_TDBR) Register</td>
<td>10-17</td>
</tr>
<tr>
<td>SPI Receive Data Buffer (SPIx_RDBR) Register</td>
<td>10-18</td>
</tr>
<tr>
<td>SPI Receive Data Buffer Shadow (SPIx_SHADOW) Register</td>
<td>10-19</td>
</tr>
<tr>
<td>Register Functions</td>
<td>10-19</td>
</tr>
<tr>
<td>SPI Transfer Formats</td>
<td>10-20</td>
</tr>
<tr>
<td>SPI General Operation</td>
<td>10-23</td>
</tr>
<tr>
<td>Clock Signals</td>
<td>10-24</td>
</tr>
<tr>
<td>Master Mode Operation</td>
<td>10-24</td>
</tr>
<tr>
<td>Transfer Initiation From Master (Transfer Modes)</td>
<td>10-25</td>
</tr>
<tr>
<td>Slave Mode Operation</td>
<td>10-26</td>
</tr>
<tr>
<td>Slave Ready for a Transfer</td>
<td>10-28</td>
</tr>
<tr>
<td>Error Signals and Flags</td>
<td>10-28</td>
</tr>
<tr>
<td>Mode Fault Error (MODF)</td>
<td>10-28</td>
</tr>
<tr>
<td>Transmission Error (TXE)</td>
<td>10-30</td>
</tr>
<tr>
<td>Reception Error (RBSY)</td>
<td>10-30</td>
</tr>
<tr>
<td>Transmit Collision Error (TXCOL)</td>
<td>10-30</td>
</tr>
<tr>
<td>Beginning and Ending an SPI Transfer</td>
<td>10-30</td>
</tr>
<tr>
<td>DMA</td>
<td>10-32</td>
</tr>
<tr>
<td>DMA Functionality</td>
<td>10-32</td>
</tr>
<tr>
<td>Master Mode DMA Operation</td>
<td>10-33</td>
</tr>
<tr>
<td>Slave Mode DMA Operation</td>
<td>10-36</td>
</tr>
<tr>
<td>Timing</td>
<td>10-39</td>
</tr>
</tbody>
</table>
PARALLEL PERIPHERAL INTERFACE

PPI Registers ................................................................. 11-2
  PPI_CONTROL Register ........................................... 11-3
  PPI_STATUS Register .............................................. 11-8
  PPI_DELAY Register ............................................... 11-9
  PPI_COUNT Register .............................................. 11-11
  PPI_FRAME Register ............................................... 11-12
ITU-R 656 Modes ......................................................... 11-13
  ITU-R 656 Background .......................................... 11-13
  ITU-R 656 Input Modes .......................................... 11-17
    Entire Field ...................................................... 11-18
    Active Video Only ............................................ 11-19
    Vertical Blanking Interval (VBI) Only .................. 11-19
  ITU-R 656 Output Mode .......................................... 11-20
  Frame Synchronization in ITU-R 656 Modes .............. 11-20
General-Purpose PPI Modes ........................................ 11-21
  Data Input (RX) Modes .......................................... 11-22
    No Frame Syncs ............................................... 11-23
    1, 2, or 3 External Frame Syncs ......................... 11-24
    2 or 3 Internal Frame Syncs .............................. 11-25
  Data Output (TX) Modes .......................................... 11-25
    No Frame Syncs ............................................... 11-26
    1 or 2 External Frame Syncs .............................. 11-26
    1, 2, or 3 Internal Frame Syncs ....................... 11-27
Frame Synchronization in GP Modes ........................................ 11-28
Modes with Internal Frame Syncs ....................................... 11-28
Modes With External Frame Syncs ..................................... 11-30
DMA Operation .................................................................... 11-31
Data Transfer Scenarios ..................................................... 11-32

UART PORT CONTROLLERS

Serial Communications ......................................................... 12-2
UART Control and Status Registers ....................................... 12-2
  UART Line Control (UARTx_LCR) Register ....................... 12-3
  UART Modem Control (UARTx_MCR) Register ................... 12-4
  UART Line Status (UARTx_LSR) Register ......................... 12-5
  UART Transmit Holding (UARTx_THR) Register ............... 12-6
  UART Receive Buffer (UARTx_RBR) Register .......... 12-7
  UART Interrupt Enable (UARTx_IER) Register ............. 12-8
  UART Interrupt Identification (UARTx_IIR) Register .. 12-10
  UARTx_DLL and UARTx_DLH Registers ..................... 12-12
  UART Scratch (UARTx_SCR) Register ....................... 12-14
  UART Global Control (UARTx_GCTL) Register ............ 12-14
Non-DMA Mode ................................................................. 12-15
DMA Mode .......................................................................... 12-16
Mixing Modes ................................................................... 12-17
IrDA Support ...................................................................... 12-18
  IrDA Transmitter Description .................................. 12-18
  IrDA Receiver Description ....................................... 12-19
SERIAL PORT CONTROLLERS

SPORT Operation ................................................................. 13-8
SPORT Disable ..................................................................... 13-9
Setting SPORT Modes ....................................................... 13-10
Register Writes and Effective Latency ................................. 13-11
SPORT Transmit Configuration
(SPORTx_TCR1, SPORTx_TCR2) Registers ......................... 13-11
SPORT Receive Configuration
(SPORTx_RCR1, SPORTx_RCR2) Registers ......................... 13-18
Data Word Formats ............................................................. 13-22
SPORT Transmit Data (SPORTx_TX) Register ....................... 13-23
SPORT Receive Data (SPORTx_RX) Register ....................... 13-25
SPORT Status (SPORTx_STAT) Register .............................. 13-28
   SPORT RX, TX, and Error Interrupts ................................. 13-30
   PAB Errors .................................................................... 13-31
SPORT Transmit Serial Clock Divider
(SPORTx_TCLKDIV, SPORTx_RCLKDIV) Registers ................. 13-31
SPORT Transmit Frame Sync Divider
(SPORTx_TFSDIV, SPORTx_RFSDIV) Register ....................... 13-32
Clock and Frame Sync Frequencies ..................................... 13-34
   Maximum Clock Rate Restrictions ................................. 13-35
   Frame Sync & Clock Example ......................................... 13-35
Word Length ...................................................................... 13-36
Bit Order .......................................................................... 13-36
Data Type .......................................................................... 13-36
Companding ........................................................................................................ 13-37
Clock Signal Options .......................................................................................... 13-38
Frame Sync Options ............................................................................................ 13-39
  Framed Versus Unframed ................................................................................ 13-39
  Internal Versus External Frame Syncs ........................................................... 13-40
  Active Low Versus Active High Frame Syncs .............................................. 13-41
  Sampling Edge for Data and Frame Syncs .................................................... 13-41
  Early Versus Late Frame Syncs
    (Normal Versus Alternate Timing) ............................................................. 13-43
  Data Independent Transmit Frame Sync ...................................................... 13-45
Moving Data Between SPORTs and Memory ....................................................... 13-46
Stereo Serial Operation ....................................................................................... 13-46
Multichannel Operation ...................................................................................... 13-50
  SPORT Multichannel Configuration (SPORTx_MCMCn)
    Registers ........................................................................................................ 13-53
  Multichannel Enable ....................................................................................... 13-54
  Frame Syncs in Multichannel Mode .............................................................. 13-55
  The Multichannel Frame ................................................................................ 13-57
  Multichannel Frame Delay ............................................................................. 13-58
  Window Size .................................................................................................... 13-58
  Window Offset ................................................................................................ 13-59
  SPORT Current Channel (SPORTx_CHNL) Register .................................... 13-59
  Other Multichannel Fields in SPORTx_MCMC2 ........................................... 13-60
Channel Selection Register .................................................. 13-61

SPORT Multichannel Receive Selection
(SPORTx_MRCSn) Registers ............................................. 13-62

SPORT Multichannel Transmit Selection
(SPORTx_MTCSn) Registers ........................................... 13-64

Multichannel DMA Data Packing ........................................ 13-65

Support for H.100 Standard Protocol ................................. 13-66

2X Clock Recovery Control ................................................. 13-66

SPORT Pin/Line Terminations ............................................. 13-67

Timing Examples ................................................................... 13-67

GENERAL-PURPOSE INPUT/OUTPUT PORT F

GPIO Port F Registers (MMRs) ............................................... 14-5

GPIO Port F Direction (PORTFIO_DIR) Register .................. 14-5

GPIO Port F Value Registers Overview ............................... 14-6

GPIO Port F Data (PORTFIO) Register ................................. 14-8

GPIO Port F Set (PORTFIO_SET), GPIO Port F Clear
(PORTFIO_CLEAR), and GPIO Port F Toggle
(PORTFIO_TOGGLE) Registers ............................................ 14-8

GPIO Port F Mask Interrupt Registers Overview .................. 14-11

GPIO Port F Interrupt Generation Flow .............................. 14-13

GPIO Port F Interrupt A (PORTFIO_MASKA,
PORTFIO_MASKA_CLEAR,
PORTFIO_MASKA_SET,
PORTFIO_MASKA_TOGGLE) Registers ......................... 14-15
Contents

GPIO Port F Interrupt B (PORTFIO_MASKB, PORTFIO_MASKB_CLEAR, PORTFIO_MASKB_SET, PORTFIO_MASKB_TOGGLE) Registers .......................... 14-17
GPIO Port F Polarity (PORTFIO_POLAR) Register ............. 14-18
GPIO Port F Interrupt Sensitivity (PORTFIO_EDGE) Register ............................................................................. 14-19
GPIO Port F Set on Both Edges (PORTFIO_BOTH) Register ............................................................................. 14-20
GPIO Port F Input Enable (PORTFIO_INEN) Register ..... 14-21
Performance/Throughput ................................................................. 14-22

GENERAL-PURPOSE INPUT/OUTPUT PORTS C, D, E

GPIO Memory-Mapped Registers (MMRs) ......................... 15-4
GPIO Function Enable (PORTxIO_FER) Register .............. 15-5
GPIO Direction (PORTxIO_DIR) Register ......................... 15-6
GPIO Input Enable (PORTxIO_INEN) Register ................. 15-9
GPIO Value Registers ................................................................. 15-11
GPIO Data (PORTxIO) Register ........................................ 15-12
GPIO Set (PORTxIO_SET), GPIO Clear (PORTxIO_CLEAR), and GPIO Toggle (PORTxIO_TOGGLE) Registers ................. 15-13
Performance/Throughput ................................................................. 15-19
TIMERS

General-Purpose Timers .............................................................. 16-1
Timer Registers ........................................................................... 16-4
  TIMER_ENABLE Register .......................................................... 16-4
  TIMER_DISABLE Register ....................................................... 16-5
  TIMER_STATUS Register .......................................................... 16-6
  TIMERx_CONFIG Registers .................................................. 16-8
  TIMERx_COUNTER Registers .............................................. 16-9
  TIMERx_PERIOD and TIMERx_WIDTH Registers ............ 16-10
Using the Timer ........................................................................ 16-13
  Pulse-Width Modulation (PWM_OUT) Mode .................... 16-15
    Output Pad Disable .............................................................. 16-17
    Single Pulse Generation ..................................................... 16-17
    Pulse-Width Modulation Waveform Generation .......... 16-17
    Stopping the Timer in PWM_OUT Mode ......................... 16-19
    Externally Clocked PWM_OUT ....................................... 16-20
    PULSE_HI Toggle Mode ....................................................... 16-21
  Pulse Width Count and Capture (WDTH_CAP) Mode ...... 16-25
    Autobaud Mode .................................................................... 16-34
    External Event (EXT_CLK) Mode ...................................... 16-35
    Using the Timers With the PPI .......................................... 16-37
    Interrupts ........................................................................... 16-37
    Illegal States ....................................................................... 16-38
  Summary ............................................................................. 16-42
Core Timer ............................................................................................................. 16-44
TCNTL Register .................................................................................................... 16-45
TCOUNT Register .................................................................................................. 16-46
TPERIOD Register ................................................................................................ 16-47
TSCALE Register .................................................................................................. 16-48
Watchdog Timer .................................................................................................... 16-48
Watchdog Timer Operation .................................................................................. 16-49
WDOG_CNT Register ............................................................................................ 16-49
WDOG_STAT Register .......................................................................................... 16-50
WDMOG_CTL Register .......................................................................................... 16-52

REAL-TIME CLOCK

Interfaces ............................................................................................................... 17-2
RTC Clock Requirements .................................................................................... 17-2
RTC Programming Model .................................................................................... 17-4
Register Writes .................................................................................................... 17-5
Write Latency ......................................................................................................... 17-6
Register Reads ....................................................................................................... 17-7
Deep Sleep ............................................................................................................. 17-7
Prescaler Enable ................................................................................................... 17-8
Event Flags ......................................................................................................... 17-8
Interrupts .............................................................................................................. 17-11
RTC Status (RTC_STAT) Register ....................................................................... 17-13
RTC Interrupt Control (RTC_ICTL) Register ................................................... 17-14
RTC Interrupt Status (RTC_ISTAT) Register .................................................... 17-15
RTC Stopwatch Count (RTC_SWCNT) Register ............................................................. 17-16
RTC Alarm (RTC_ALARM) Register ........................................................................ 17-18
RTC Prescaler Enable (RTC_PREN) Register ................................................................. 17-18
State Transitions Summary ......................................................................................... 17-20

EXTERNAL BUS INTERFACE UNIT

Overview ......................................................................................................................... 18-1
  Block Diagram ........................................................................................................... 18-2
  Internal Memory Interfaces ....................................................................................... 18-4
  External Memory Interfaces ....................................................................................... 18-5
  EBIU Programming Model ......................................................................................... 18-7
  Error Detection ........................................................................................................... 18-8
  Asynchronous Memory Interface .............................................................................. 18-9
    Asynchronous Memory Address Decode ................................................................. 18-9
    EBIU_AMGCTL Register .......................................................................................... 18-10
    EBIU_AMBCTL0 and EBIU_AMBCTL1 Registers ..................................................... 18-11
    Avoiding Bus Contention ....................................................................................... 18-15
      ARDY Input Control ............................................................................................. 18-16
    Programmable Timing Characteristics ................................................................. 18-17
      Asynchronous Accesses by Core Instructions ...................................................... 18-17
      Asynchronous Reads ........................................................................................... 18-17
      Asynchronous Writes ........................................................................................... 18-19
CONTROLLER AREA NETWORK (CAN) MODULE

Overview ................................................................. 19-1
Low Power Features ....................................................... 19-4
   CAN Wake-Up From Hibernate State ......................... 19-4
   CAN Built-In Sleep Mode ........................................ 19-5
CAN Module Control and Configuration Registers .......... 19-6
   CAN Control (CAN_CONTROL) Register .................. 19-6
   CAN Status (CAN_STATUS) Register ...................... 19-11
   CAN Clock (CAN_CLOCK) Register ......................... 19-14
   CAN Timing (CAN_TIMING) Register ..................... 19-15
   CAN Debug (CAN_DEBUG) Register ....................... 19-17
Data Storage ............................................................ 19-21
Mailbox Identifier Word Registers ............................................. 19-22
  CAN Mailbox Identifier 1 (CAN_MBxx_ID1) Registers ...... 19-23
  CAN Mailbox Identifier 0 (CAN_MBxx_ID0) Registers ...... 19-25
  CAN Mailbox Time Stamp (CAN_MBxx_TIMESTAMP) Registers ................................................................. 19-26
  CAN Mailbox Length (CAN_MBxx_LENGTH) Registers ... 19-27
  CAN Mailbox Data (CAN_MBxx_DATAx) Registers .......... 19-28
Mailbox Area .................................................................................. 19-33
Mailbox Types ................................................................................. 19-34
Mailbox Control ............................................................................. 19-35
  CAN Mailbox Configuration (CAN_MCx) and Direction (CAN_MDx) Registers ................................................................. 19-35
Receive Logic .................................................................................... 19-38
  Acceptance Filter/Data Acceptance Filter ......................... 19-39
  CAN Acceptance Mask (CAN_AMxx) Registers ................... 19-40
Receive Control Registers ............................................................... 19-45
  CAN Receive Message Pending (CAN_RMPx) Register ....... 19-45
  CAN Receive Message Lost (CAN_RMLx) Register .............. 19-46
  CAN Overwrite Protection/Single Shot Transmission (CAN_OPSSx) Register ................................................................. 19-48
Transmit Logic ................................................................................. 19-49
  Retransmission ........................................................................ 19-50
  Single Shot Transmission ......................................................... 19-50
  Transmit Priority Defined by Mailbox Number .................... 19-51
Contents

Transmit Control Registers ................................................................. 19-51
  CAN Transmission Request Set
    (CAN_TRSx) Registers .............................................................. 19-51
  CAN Transmission Request Reset
    (CAN_TRRx) Registers .............................................................. 19-53
  CAN Abort Acknowledge (CAN_AAx) Register .............................. 19-56
  CAN Transmission Acknowledge
    (CAN_TAx) Register ................................................................. 19-57
  CAN Mailbox Temporary Disable (CAN_MBTD) Register ............... 19-59
  CAN Remote Frame Handling
    (CAN_RFHx) Registers ............................................................. 19-60
CAN Interrupts .................................................................................... 19-62
  CAN Interrupt (CAN_INTR) Register ........................................... 19-62
Mailbox Interrupts ............................................................................. 19-64
  CAN Mailbox Interrupt Mask (CAN_MBIMx) Registers .......... 19-65
  CAN Mailbox Interrupt Mask Flag
    (CAN_MBTIFx) Registers ............................................................ 19-67
  CAN Mailbox Receive Interrupt Flag
    (CAN_MBRIFx) Registers ............................................................ 19-68
Global Interrupt .................................................................................. 19-70
  Global Interrupt Logic ................................................................. 19-74
  CAN Global Interrupt Mask (CAN_GIM) Register .................. 19-74
  CAN Global Interrupt Status (CAN_GIS) Register .................. 19-75
  CAN Global Interrupt Flag (CAN_GIF) Register ....................... 19-75
Universal Counter Module ........................................................ 19-77
Time Stamp Mode ........................................................................ 19-77
Watchdog Mode ........................................................................... 19-78
Auto Transmit Mode .................................................................... 19-79
Event Counter Mode ..................................................................... 19-79
CAN Universal Counter Configuration
(CAN_UCCNF) Register .......................................................... 19-80
CAN Universal Counter (CAN_UCCNT) Register .................. 19-83
CAN Universal Counter Reload/Capture
(CAN_UCRC) Register .......................................................... 19-84
Programmable Warning Limit for RXECNT and TXECNT ...... 19-84
CAN Errors and Warnings ............................................................ 19-85
CAN Error Counter (CAN_CEC) Register ............................ 19-85
CAN Error Status (CAN_ESR) Register .................................. 19-86
CAN Error Counter Warning Level
(CAN_EWR) Register .......................................................... 19-88

TWO-WIRE INTERFACE CONTROLLERS

Overview .................................................................................... 20-1
Architecture .................................................................................. 20-2
Register Descriptions .................................................................. 20-4
    TWI Control (TWIx_CONTROL) Registers .......................... 20-4
    TWI Clock Divider (TWIx_CLKDIV) Registers ................. 20-6
    TWI Slave Mode Control (TWIx_SLAVE_CTRL) Registers .... 20-7
Contents

TWI Slave Mode Address (TWIx_SLAVE_ADDR) Registers ................................................................. 20-9
TWI Slave Mode Status (TWIx_SLAVE_STAT) Registers ................................................................. 20-9
TWI Master Mode Control (TWIx_MASTER_CTRL) Registers ......................................................... 20-11
TWI Master Mode Address (TWIx_MASTER_ADDR) Registers ....................................................... 20-14
TWI Master Mode Status (TWIx_MASTER_STAT) Registers .......................................................... 20-14
TWI FIFO Control (TWIx_FIFO_CTRL) Registers .............................................................. 20-18
TWI FIFO Status (TWIx_FIFO_STAT) Registers .............................................................. 20-20
TWI Interrupt Mask (TWIx_INT_ENABLE) Registers ............................................................... 20-22
TWI Interrupt Status (TWIx_INT_STAT) Registers ............................................................... 20-24
TWI FIFO Transmit Data Single Byte (TWIx_XMT_DATA8) Registers ........................................... 20-27
TWI FIFO Transmit Data Double Byte (TWIx_XMT_DATA16) Registers ........................................ 20-27
TWI FIFO Receive Data Single Byte (TWIx_RCV_DATA8) Registers ............................................. 20-28
TWI FIFO Receive Data Double Byte (TWIx_RCV_DATA16) Registers ........................................ 20-29
Data Transfer Mechanics ........................................................................................................... 20-30
Clock Generation and Synchronization ....................................................................................... 20-31
Bus Arbitration .......................................................................................................................... 20-32
Start and Stop Conditions ........................................................................................................ 20-32
Contents

General Call Support .......................................................... 20-33
Fast Mode ........................................................................... 20-34
Programming Examples ............................................................ 20-34
General Setup ..................................................................... 20-35
Slave Mode ......................................................................... 20-35
Master Mode Clock Setup ................................................... 20-36
Master Mode Transmit ........................................................ 20-37
Master Mode Receive .......................................................... 20-38
Repeated Start Condition .................................................... 20-39
  Transmit/Receive Repeated Start Sequence ...................... 20-39
  Receive/Transmit Repeated Start Sequence ...................... 20-40
Clock Stretching ............................................................. 20-42
  Clock Stretching During FIFO Underflow ...................... 20-42
  Clock Stretching During FIFO Overflow ........................ 20-44
  Clock Stretching During Repeated Start Condition ........ 20-45
Electrical Specifications ............................................................ 20-47

SYSTEM DESIGN

Pin Descriptions ......................................................................... 21-1
  Recommendations for Unused Pins ....................................... 21-1
Resetting the Processor ............................................................... 21-1
Booting the Processor ................................................................. 21-2
Managing Clocks ........................................................................ 21-4
  Managing Core and System Clocks ......................................... 21-4
Configuring and Servicing Interrupts ........................................... 21-4
## Contents

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semaphores</td>
<td>21-5</td>
</tr>
<tr>
<td>Example Code for Query Semaphore</td>
<td>21-6</td>
</tr>
<tr>
<td>Data Delays, Latencies and Throughput</td>
<td>21-6</td>
</tr>
<tr>
<td>Bus Priorities</td>
<td>21-7</td>
</tr>
<tr>
<td>External Memory Design Issues</td>
<td>21-7</td>
</tr>
<tr>
<td>Example Asynchronous Memory Interfaces</td>
<td>21-7</td>
</tr>
<tr>
<td>Using SDRAMs Smaller Than 16M Byte</td>
<td>21-8</td>
</tr>
<tr>
<td>Managing SDRAM Refresh During PLL Transitions</td>
<td>21-8</td>
</tr>
<tr>
<td>Avoiding Bus Contention</td>
<td>21-10</td>
</tr>
<tr>
<td>High Frequency Design Considerations</td>
<td>21-11</td>
</tr>
<tr>
<td>Point-to-Point Connections on Serial Ports</td>
<td>21-11</td>
</tr>
<tr>
<td>Signal Integrity</td>
<td>21-12</td>
</tr>
<tr>
<td>Decoupling Capacitors and Ground Planes</td>
<td>21-12</td>
</tr>
<tr>
<td>Oscilloscope Probes</td>
<td>21-14</td>
</tr>
<tr>
<td>Recommended Reading</td>
<td>21-14</td>
</tr>
</tbody>
</table>

### BLACKFIN PROCESSOR DEBUG

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Watchpoint Unit</td>
<td>22-1</td>
</tr>
<tr>
<td>Instruction Watchpoints</td>
<td>22-4</td>
</tr>
<tr>
<td>Instruction Watchpoint Address (WPIAn) Registers</td>
<td>22-5</td>
</tr>
<tr>
<td>Instruction Watchpoint Address Count (WPIACNTn) Registers</td>
<td>22-6</td>
</tr>
<tr>
<td>Instruction Watchpoint Address Control (WPIACTL) Register</td>
<td>22-7</td>
</tr>
<tr>
<td>Data Address Watchpoints</td>
<td>22-10</td>
</tr>
</tbody>
</table>
Contents

Data Watchpoint Address (WPDAAn) Registers ...................... 22-11
Data Watchpoint Address Count Value (WPDACNTn) Registers ................................................................. 22-11
Data Watchpoint Address Control (WPDACTL) Register .................................................................................. 22-12
Watchpoint Status (WPSTAT) Register ........................................... 22-12
Trace Unit ........................................................................................................ 22-14
Trace Buffer Control (TBUFCTL) Register ......................... 22-16
Trace Buffer Status (TBUFSTAT) Register ............................ 22-17
Trace Buffer (TBUF) Register ............................................................... 22-17
Code to Recreate the Execution Trace in Memory .................. 22-18
Performance Monitoring Unit ......................................................... 22-19
Performance Monitor Counter (PFCNTRn) Registers .......... 22-19
Performance Monitor Control (PFCTL) Register .................. 22-20
Event Monitor Table ........................................................................ 22-22
Cycle Counter ........................................................................................ 22-23
CYCLES and CYCLES2 Registers ................................................. 22-24
Product Identification Register .................................................. 22-25
DSP Device ID (DSPID) Register ................................................... 22-25

BLACKFIN PROCESSOR CORE MMR ASSIGNMENTS

L1 Data Memory Controller Registers .................................................. A-1
L1 Instruction Memory Controller Registers ................................ A-4
Interrupt Controller Registers ......................................................... A-7
Core Timer Registers .......................................................................... A-9
Contents

Debug, MP, and Emulation Unit Registers ........................................ A-9
Trace Unit Registers ........................................................................ A-10
Watchpoint and Patch Registers ..................................................... A-10
Performance Monitor Registers ....................................................... A-12

SYSTEM MMR ASSIGNMENTS

Dynamic Power Management Registers ............................................ B-2
System Reset and Interrupt Control
  Registers ...................................................................................... B-2
Watchdog Timer Registers .............................................................. B-3
Real-Time Clock Registers .............................................................. B-4
Parallel Peripheral Interface (PPI) Registers ..................................... B-4
UART Controller Registers .............................................................. B-5
SPI Controller Registers ................................................................. B-8
Timer Registers ............................................................................... B-10
GPIO Port C, D, and E Registers .................................................... B-11
GPIO Port F Registers ..................................................................... B-13
SPORT Controller Registers .......................................................... B-14
DMA/Memory DMA Control Registers .......................................... B-22
External Bus Interface Unit Registers .............................................. B-25
CAN Registers ................................................................................ B-26
Two-Wire Interface Registers ........................................................ B-36
TEST FEATURES

JTAG Standard ............................................................................. C-1
Boundary-Scan Architecture ......................................................... C-2
Instruction Register ...................................................................... C-4
Public Instructions ........................................................................ C-6
EXTEST – Binary Code 00000 ..................................................... C-6
SAMPLE/PRELOAD – Binary Code 10000 ................................. C-6
BYPASS – Binary Code 11111 .................................................... C-6
Boundary-Scan Register ............................................................. C-7

NUMERIC FORMATS

Unsigned or Signed: Two’s-Complement Format ......................... D-1
Integer or Fractional ................................................................. D-1
Binary Multiplication .................................................................. D-5
Fractional Mode and Integer Mode ........................................... D-6
Block Floating-Point Format ....................................................... D-7

INDEX
Thank you for purchasing and developing systems using Blackfin® processors from Analog Devices, Inc.

Purpose of This Manual

*ADSP-BF538/ADSP-BF538F Blackfin Processor Hardware Reference* contains information about the architecture for the ADSP-BF538 processors. The architectural descriptions cover functional blocks, buses, and ports, including all features and processes that they support.

For programming information, see *Blackfin Processor Programming Reference*. For timing, electrical, and package specifications, see *ADSP-BF538/ADSP-BF538F Embedded Processor Data Sheet*.

Intended Audience

The primary audience for this manual is a programmer who is familiar with Analog Devices processors. The manual assumes the audience has a working knowledge of the appropriate processor architecture and instruction set. Programmers who are unfamiliar with Analog Devices processors can use this manual, but should supplement it with other texts, such as hardware and programming reference manuals that describe their target architecture.
Manual Contents

This manual contains:

- Chapter 1, “Introduction”
  Provides a high level overview of the processor. Architectural descriptions include functional blocks, buses, and ports, including features and processes they support.

- Chapter 2, “Computational Units”
  Describes the arithmetic/logic units (ALUs), multiplier/accumulator units (MACs), shifter, and the set of video ALUs. The chapter also discusses data formats, data types, and register files.

- Chapter 3, “Operating Modes and States”
  Describes the three operating modes of the processor: Emulation mode, Supervisor mode, and User mode. The chapter also describes Idle state and Reset state.

- Chapter 4, “Program Sequencer”
  Describes the operation of the program sequencer, which controls program flow by providing the address of the next instruction to be executed. The chapter also discusses loops, subroutines, jumps, interrupts, and exceptions.

- Chapter 5, “Data Address Generators”
  Describes the Data Address Generators (DAGs), addressing modes, how to modify DAG and Pointer registers, memory address alignment, and DAG instructions.

- Chapter 6, “Memory”
  Describes L1 memories. In particular, details their memory architecture, memory model, memory transaction model, and memory-mapped registers (MMRs). Discusses the instruction, data, and scratchpad memory, which are part of the Blackfin processor core.
• Chapter 7, “Chip Bus Hierarchy”
  Describes on-chip buses, including how data moves through the system. The chapter also discusses the system memory map, major system components, and the system interconnects.

• Chapter 8, “Dynamic Power Management”
  Describes system reset and power-up configuration, system clocking and control, and power management.

• Chapter 9, “Direct Memory Access”
  Describes the peripheral DMA and memory DMA controllers. The peripheral DMA section discusses direct, block data movements between a peripheral with DMA access and internal or external memory spaces.

  The memory DMA section discusses memory-to-memory transfer capabilities among the processor memory spaces and the L1, external synchronous, and asynchronous memories.

• Chapter 10, “SPI Compatible Port Controllers”
  Describes the serial peripheral interface (SPI) ports that provide an I/O interface to a variety of SPI compatible peripheral devices.

• Chapter 11, “Parallel Peripheral Interface”
  Describes the parallel peripheral interface (PPI) of the processor. The PPI is a half-duplex, bidirectional port accommodating up to 16 bits of data and used for digital video and data converter applications.

• Chapter 12, “UART Port Controllers”
  Describes the Universal Asynchronous Receiver/Transmitter (UART) ports, which convert data between serial and parallel formats and includes modem control and interrupt handling hardware. The UARTs support the half-duplex IrDA® SIR protocol as a mode-enabled feature.
• Chapter 13, “Serial Port Controllers”
  Describes the independent, synchronous serial port controllers that
  provide an I/O interface to a variety of serial peripheral devices.

• Chapter 14, “General-Purpose Input/Output Port F”
  Describes the GPIO Port F, including how to configure the pins as
  inputs and outputs and how to generate interrupts.

• Chapter 15, “General-Purpose Input/Output Ports C, D, E”
  Describes the general-purpose I/O pins, including how to configure
  the pins as inputs and outputs.

• Chapter 16, “Timers”
  Describes the general-purpose timers that can be configured in any
  of three modes; the core timer that can generate periodic interrupts
  for a variety of timing functions; and the watchdog timer that can
  implement software watchdog functions, such as generating events
  to the Blackfin processor core.

• Chapter 17, “Real-Time Clock”
  Describes a set of digital watch features of the processor, including
  time of day, alarm, and stopwatch countdown.

• Chapter 18, “External Bus Interface Unit”
  Describes the External Bus Interface Unit of the processor. The
  chapter also discusses the asynchronous memory interface, the
  SDRAM controller (SDC), related registers, and SDC configuration
  and commands.

• Chapter 19, “Controller Area Network (CAN) Module”
  Describes the CAN module, a low bit rate serial interface intended
  for use in applications where bit rates are typically up to 1Mbit/s.

• Chapter 20, “Two-Wire Interface Controllers”
  Describes the Two-Wire Interface (TWI) controllers, which allow
  a device to interface to an Inter IC bus as specified by the Philips
• Chapter 21, “System Design”  
Describes how to use the processor as part of an overall system. It includes information about interfacing the processor to external memory chips, bus timing and latency numbers, semaphores, and a discussion of the treatment of unused pins.

• Chapter 22, “Blackfin Processor Debug”  
Describes the Blackfin processor debug functionality, which can be used for software debugging and complements some services often found in an operating system.

• Appendix A, “Blackfin Processor Core MMR Assignments”  
Lists the core memory-mapped registers, their addresses, and cross-references to text.

• Appendix B, “System MMR Assignments”  
Lists the system memory-mapped registers, their addresses, and cross-references to text.

• Appendix C, “Test Features”  
Describes test features for the processor; discusses the JTAG standard, boundary-scan architecture, instruction and boundary registers, and public instructions.

• Appendix D, “Numeric Formats”  
Describes various aspects of the 16-bit data format. The chapter also describes how to implement a block floating-point format in software.
What’s New in This Manual

This is Revision 1.2 of the *ADSP-BF538/ADSP-BF538F Blackfin Processor Hardware Reference*. This revision corrects minor typographical errors and the following issues:

- UART not half-duplex in Chapter 1, “Introduction”
- System reset code example in Chapter 3, “Operating Modes and States”
- `RETI` instructions need not be first in nested interrupts and complete table of hardware conditions causing hardware interrupts in Chapter 4, “Program Sequencer”
- Core priority over DMA when accessing L1 SRAM in Chapter 7, “Chip Bus Hierarchy”
- Removal of reference to datasheet, note on programming the `STOPCK` bit, and description of the `WAKE` bit in Chapter 8, “Dynamic Power Management”
- Obsolete DMA error address range deleted in Chapter 9, “Direct Memory Access”
- Termination of SPI TX DMA operations in Chapter 10, “SPI Compatible Port Controllers”
- Behavior on startup when using an external clock and receiver and transmitter enable bit names standardized on `RSPEN` and `TSPEN` in Chapter 13, “Serial Port Controllers”
- Note on the `TINT` bit in the `TCNTL` register in Chapter 16, “Timers”
- Sampling the `ARDY` pin when it is asserted and note on timing dependencies for the `TRP` and `TRAS` settings in the `EBIU_SDGCTL` register in Chapter 18, “External Bus Interface Unit”
• Detection of recessive-to-dominant edges and note on CAN_GIS and CAN_GIF programming in Chapter 19, “Controller Area Network (CAN) Module”

• Coverage of previously undocumented clock stretching behavior—and miscellaneous changes across Chapter 20, “Two-Wire Interface Controllers”

• Clarification of watchpoint ranges in Chapter 22, “Blackfin Processor Debug”

Technical Support

You can reach Analog Devices processors and DSP technical support in the following ways:

• Post your questions in the processors and DSP support community at EngineerZone®:
  http://ez.analog.com/community/dsp

• Submit your questions to technical support directly at:
  http://www.analog.com/support

• E-mail your questions about processors, DSPs, and tools development software from CrossCore® Embedded Studio or VisualDSP++®:

  Choose Help > Email Support. This creates an e-mail to processor.tools.support@analog.com and automatically attaches your CrossCore Embedded Studio or VisualDSP++ version information and license.dat file.

• E-mail your questions about processors and processor applications to:
  processor.support@analog.com or processor.china@analog.com (Greater China support)
• In the USA only, call 1-800-ANALOGD (1-800-262-5643)

• Contact your Analog Devices sales office or authorized distributor. Locate one at: www.analog.com/adi-sales

• Send questions by mail to: Processors and DSP Technical Support Analog Devices, Inc. Three Technology Way P.O. Box 9106 Norwood, MA 02062-9106 USA

Supported Processors

The name “Blackfin” refers to a family of 16-bit, embedded processors. Refer to the CCES or VisualDSP++ online help for a complete list of supported processors.

Product Information

Product information can be obtained from the Analog Devices Web site and the CCES or VisualDSP++ online help.
Analog Devices Web Site


To access a complete technical library for each processor family, go to http://www.analog.com/processors/technical_library. The manuals selection opens a list of current manuals related to the product as well as a link to the previous revisions of the manuals. When locating your manual title, note a possible errata check mark next to the title that leads to the current correction report against the manual.

Also note, myAnalog is a free feature of the Analog Devices Web site that allows customization of a Web page to display only the latest information about products you are interested in. You can choose to receive weekly e-mail notifications containing updates to the Web pages that meet your interests, including documentation errata against all manuals. myAnalog provides access to books, application notes, data sheets, code examples, and more.

Visit myAnalog to sign up. If you are a registered user, just log on. Your user name is your e-mail address.

EngineerZone

EngineerZone is a technical support forum from Analog Devices, Inc. It allows you direct access to ADI technical support engineers. You can search FAQs and technical information to get quick answers to your embedded processing and DSP design questions.

Use EngineerZone to connect with other DSP developers who face similar design challenges. You can also use this open forum to share knowledge and collaborate with the ADI support team and your peers. Visit http://ez.analog.com to sign up.
# Notation Conventions

Text conventions in this manual are identified and described as follows.

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Close command</strong></td>
<td>Titles in reference sections indicate the location of an item within the IDE environment’s menu system (for example, the Close command appears on the File menu).</td>
</tr>
<tr>
<td>{(this</td>
<td>that)}</td>
</tr>
<tr>
<td>[this</td>
<td>that]</td>
</tr>
<tr>
<td>[this,...]</td>
<td>Optional item lists in syntax descriptions appear within brackets delimited by commas and terminated with an ellipsis; read the example as an optional comma-separated list of this.</td>
</tr>
<tr>
<td>.SECTION</td>
<td>Commands, directives, keywords, and feature names are in text with letter gothic font.</td>
</tr>
<tr>
<td>filename</td>
<td>Non-keyword placeholders appear in text with italic style format.</td>
</tr>
</tbody>
</table>

- **Note:** For correct operation, ...
  An A Note provides supplementary information on a related topic. In the online version of this book, the word Note appears instead of this symbol.

- **Caution:** Incorrect device operation may result if ...
  A Caution identifies conditions or inappropriate usage of the product that could lead to undesirable results or product damage. In the online version of this book, the word Caution appears instead of this symbol.

- **Warning:** Injury to device users may result if ...
  A Warning identifies conditions or inappropriate usage of the product that could lead to conditions that are potentially hazardous for devices users. In the online version of this book, the word Warning appears instead of this symbol.
Register Diagram Conventions

Register diagrams use the following conventions:

- The descriptive name of the register appears at the top, followed by the short form of the name in parentheses.

- If the register is read-only (RO), write-1-to-set (W1S), or write-1-to-clear (W1C), this information appears under the name. Read/write is the default and is not noted. Additional descriptive text may follow.

- If any bits in the register do not follow the overall read/write convention, this is noted in the bit description after the bit name.

- If a bit has a short name, the short name appears first in the bit description, followed by the long name in parentheses.

- The reset value appears in binary in the individual bits and in hexadecimal to the right of the register.

- Bits marked x have an unknown reset value. Consequently, the reset value of registers that contain such bits is undefined or dependent on pin values at reset.

- Shaded bits are reserved.

To ensure upward compatibility with future implementations, write back the value that is read for reserved bits in a register, unless otherwise specified.
The following figure shows an example of these conventions.

**Timer Configuration Registers (TIMERx_CONFIG)**

- **ERR_TYP[1:0] (Error Type) - RO**
  - 00 - No error.
  - 01 - Counter overflow error.
  - 10 - Period register programming error.
  - 11 - Pulse width register programming error.

- **EMU_RUN (Emulation Behavior Select)**
  - 0 - Timer counter stops during emulation.
  - 1 - Timer counter runs during emulation.

- **TOGGLE_HI (PWM_OUT PULSE_HI Toggle Mode)**
  - 0 - The effective state of PULSE_HI is the programmed state.
  - 1 - The effective state of PULSE_HI alternates each period.

- **CLK_SEL (Timer Clock Select)**
  - This bit must be set to 1, when operating the PPI in GP Output modes.
  - 0 - Use system clock SCLK for counter.
  - 1 - Use PWM_CLK to clock counter.

- **OUT_DIS (Output Pad Disable)**
  - 0 - Enable pad in PWM_OUT mode.
  - 1 - Disable pad in PWM_OUT mode.

**Figure 1. Register Diagram Example**
The ADSP-BF538 Blackfin® processor is derived from the ADSP-BF533 processor, offering similar performance and ease of use capabilities, but with enhanced peripheral features, targeted for the automotive and industrial markets. Common peripherals share the same features and functions.

Any time a processor is referenced by name (for example, ADSP-BF538), the information provided applies to the processor derivatives with on-chip flash memory as well (for example, ADSP-BF538F4 and ADSP-BF538F8).

The Blackfin processor core architecture combines a dual-MAC signal processing engine, an orthogonal RISC-like microprocessor instruction set, flexible single instruction multiple data (SIMD) capabilities, and multimedia features into a single instruction set architecture.

Blackfin products feature dynamic power management, the ability to vary both the voltage and frequency of operation, which optimizes the power consumption profile to the specific task.

## Purpose of this Manual

This Blackfin processor hardware reference provides architectural information about enhanced Blackfin processors that include the ADSP-BF538 processors. The architectural descriptions cover functional blocks, buses, and ports, including all features and processes that they support.
Purpose of this Manual

For programming information, see *Blackfin Processor Programming Reference*. For timing, electrical, and package specifications, see *ADSP-BF538/ADSP-BF538F Embedded Processor Data Sheet*.

Table 1-1 can be used to identify chapters from the *ADSP-BF533 Blackfin Processor Hardware Reference* that are applicable to ADSP-BF538 Blackfin products.

For programmers familiar with the ADSP-BF533/BF532/BF531 processors, the ADSP-BF538 is very similar, as they are built from the same processor core. The ADSP-BF538 uses many of the same peripherals that are found on the ADSP-BF533/BF532/BF531 (see Table 1-1).

Table 1-1 is intended as a guide that can be used to identify which chapters of this manual are the same/similar or new, when compared to the *ADSP-BF533 Blackfin Processor Hardware Reference* chapters; such that an experienced programmer does not need to read every chapter of this manual to understand the operation of the ADSP-BF538.

- **No changes**—means that the reader can refer directly to *ADSP-BF533 Blackfin Processor Hardware Reference* for this chapter.

- **Changed**—means that *ADSP-BF533 Blackfin Processor Hardware Reference* chapter has been copied into this book, but some changes have been made or features added.

- **New**—means that this is an entirely new chapter and this is the only source of reference for the material.
### Table 1-1. Guide to Hardware Reference Chapter Differences

<table>
<thead>
<tr>
<th>ADSP-BF538</th>
<th>ADSP-BF533</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter</td>
<td>Chapter</td>
<td>Chapter</td>
</tr>
<tr>
<td>Number</td>
<td>Title</td>
<td>Number</td>
</tr>
<tr>
<td>1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Computational Units</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Operating Modes and States</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>Program Sequencer</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>Data Address Generators</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>Memory</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>Chip Bus Hierarchy</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>Direct Memory Access</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>SPI Compatible Port Controllers</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>Parallel Peripheral Interface</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>Serial Port Controllers</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>UART Port Controller</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>General-Purpose I/O Port F</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>General-Purpose I/O Ports C, D, and E</td>
<td>New</td>
</tr>
<tr>
<td>16</td>
<td>Timers</td>
<td>15</td>
</tr>
<tr>
<td>17</td>
<td>Real Time Clock</td>
<td>16</td>
</tr>
</tbody>
</table>
Peripherals

The processor system peripherals include the following:

- Parallel peripheral interface (PPI)
- Serial ports (SPORTs)
- Serial peripheral interfaces (SPI)
- Controller area network (CAN)
- Two-wire interfaces (TWI)—for connection to an $I^2C$ network
- General-purpose timers
- Universal asynchronous receiver transmitters (UART)
- Real-time clock (RTC)
- Watchdog timer
- General-purpose I/O
These peripherals are connected to the core via several high bandwidth buses, as shown in Figure 1-1.

Figure 1-1. Processor Block Diagram

All of the peripherals, except for general-purpose I/O, real-time clock, CAN, timers, and TWI are supported by a flexible DMA structure. There are also four separate memory DMA channels dedicated to data transfers between the processor memory spaces, which include external SDRAM and asynchronous memory. Multiple on-chip buses provide enough bandwidth to keep the processor core running even when there is also activity on all of the on-chip and external peripherals.

Core Architecture

The processor core contains two 16-bit multipliers, two 40-bit accumulators, two 40-bit arithmetic logic units (ALUs), four 8-bit video ALUs, and a 40-bit shifter, as shown in Figure 1-2. The computational units process 8-, 16-, or 32-bit data from the register file.
The compute register file contains eight 32-bit registers. When performing compute operations on 16-bit operand data, the register file operates as 16 independent 16-bit registers. All operands for compute operations come from the multiported register file and instruction constant fields.
Each MAC can perform a 16- by 16-bit multiply per cycle, with accumulation to a 40-bit result. Signed and unsigned formats, rounding, and saturation are supported.

The ALUs perform a traditional set of arithmetic and logical operations on 16-bit or 32-bit data. Many special instructions are included to accelerate various signal processing tasks. These include bit operations such as field extract and population count, modulo 232 multiply, divide primitives, saturation and rounding, and sign/exponent detection. The set of video instructions include byte alignment and packing operations, 16-bit and 8-bit adds with clipping, 8-bit average operations, and 8-bit subtract/absolute value/accumulate (SAA) operations. Also provided are the compare/select and vector search instructions. For some instructions, two 16-bit ALU operations can be performed simultaneously on register pairs (a 16-bit high half and 16-bit low half of a compute register). By also using the second ALU, quad 16-bit operations are possible.

The 40-bit shifter can deposit data and perform shifting, rotating, normalization, and extraction operations.

A program sequencer controls the instruction execution flow, including instruction alignment and decoding. For program flow control, the sequencer supports PC relative and indirect conditional jumps (with static branch prediction), and subroutine calls. Hardware is provided to support zero-overhead looping. The architecture is fully interlocked, meaning that there are no visible pipeline effects when executing instructions with data dependencies.

The address arithmetic unit provides two addresses for simultaneous dual fetches from memory. It contains a multiported register file consisting of four sets of 32-bit index, modify, length, and base registers (for circular buffering), and eight additional 32-bit pointer registers (for C-style indexed stack manipulation).
Core Architecture

Blackfin products support a modified Harvard architecture in combination with a hierarchical memory structure. Level 1 (L1) memories typically operate at the full processor speed with little or no latency. At the L1 level, the instruction memory holds instructions only. The two data memories hold data, and a dedicated scratchpad data memory stores stack and local variable information.

In addition, multiple L1 memory blocks are provided, which may be configured as a mix of SRAM and cache. The memory management unit (MMU) provides memory protection for individual tasks that may be operating on the core and may protect system registers from unintended access.

The architecture provides three modes of operation: user, supervisor, and emulation. User mode has restricted access to a subset of system resources, thus providing a protected software environment. Supervisor and emulation modes have unrestricted access to the system and core resources.

The Blackfin instruction set is optimized so that 16-bit opcodes represent the most frequently used instructions. Complex DSP instructions are encoded into 32-bit opcodes as multifunction instructions. Blackfin products support a limited multi-issue capability, where a 32-bit instruction can be issued in parallel with two 16-bit instructions. This allows the programmer to use many of the core resources in a single instruction cycle.

The Blackfin assembly language uses an algebraic syntax. The architecture is optimized for use with the C compiler.
Memory Architecture

The Blackfin architecture structures memory as a single, unified 4G byte address space using 32-bit addresses. All resources, including internal memory, external memory, and I/O control registers, occupy separate sections of this common address space. The memory portions of this address space are arranged in a hierarchical structure to provide a good cost/performance balance of some very fast, low latency on-chip memory as cache or SRAM, and larger, lower-cost and lower performance off-chip memory systems. Table 1-2 shows the memory allocation for the ADSP-BF538.

Table 1-2. Memory Comparison

<table>
<thead>
<tr>
<th>Type of Memory</th>
<th>Memory size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction SRAM/Cache</td>
<td>16 KB</td>
</tr>
<tr>
<td>Instruction SRAM</td>
<td>64 KB</td>
</tr>
<tr>
<td>Instruction ROM</td>
<td>-</td>
</tr>
<tr>
<td>Data SRAM/Cache</td>
<td>32 KB</td>
</tr>
<tr>
<td>Data SRAM</td>
<td>32 KB</td>
</tr>
<tr>
<td>Scratchpad</td>
<td>4 KB</td>
</tr>
<tr>
<td>Total</td>
<td>148 KB</td>
</tr>
</tbody>
</table>

The L1 memory system is the primary highest performance memory available to the core. The off-chip memory system, accessed through the External Bus Interface Unit (EBIU), provides expansion with SDRAM, flash memory, and SRAM, optionally accessing up to 132M bytes of physical memory.

The memory DMA controller provides high bandwidth data movement capability. It can perform block transfers of code or data between the internal memory and the external memory spaces.
Memory Architecture

Internal Memory

The processor has three blocks of on-chip memory that provide high bandwidth access to the core:

L1 instruction memory, consisting of SRAM and a 4-way set-associative cache. On ROM-enabled parts, this also includes a user-definable ROM region. This memory is accessed at full processor speed.

L1 data memory, consisting of SRAM and/or a 2-way set-associative cache. This memory block is accessed at full processor speed.

L1 scratchpad RAM, which runs at the same speed as the L1 memories but is only accessible as data SRAM and cannot be configured as cache memory.

External Memory

External (off-chip) memory is accessed via the EBIU. This 16-bit interface provides a glueless connection to a bank of synchronous DRAM (SDRAM) and as many as four banks of asynchronous memory devices including flash memory, EPROM, ROM, SRAM, and memory-mapped I/O devices.

The PC133-compliant SDRAM controller can be programmed to interface to up to 128M bytes of SDRAM.

The asynchronous memory controller can be programmed to control up to four banks of devices. Each bank occupies a 1M byte segment regardless of the size of the devices used, so that these banks are only contiguous if each is fully populated with 1M byte of memory.
I/O Memory Space

Blackfin products do not define a separate I/O space. All resources are mapped through the flat 32-bit address space. On-chip I/O devices have their control registers mapped into memory-mapped registers (MMRs) at addresses near the top of the 4G byte address space. These are separated into two smaller blocks: one contains the control MMRs for all core functions and the other contains the registers needed for setup and control of the on-chip peripherals outside of the core. The MMRs are accessible only in supervisor mode. They appear as reserved space to on-chip peripherals.

Event Handling

The event controller on the processor handles all asynchronous and synchronous events to the processor. The processor event handling supports both nesting and prioritization. Nesting allows multiple event service routines to be active simultaneously. Prioritization ensures that servicing a higher priority event takes precedence over servicing a lower priority event. The controller provides support for five different types of events:

- **Emulation**
  
  Causes the processor to enter emulation mode, allowing command and control of the processor via the JTAG interface.

- **Reset**
  
  Resets the processor.

- **Nonmaskable Interrupt (NMI)**
  
  The software watchdog timer or the NMI input signal to the processor generates this event. The NMI event is frequently used as a power-down indicator to initiate an orderly shutdown of the system.
Event Handling

- Exceptions

Synchronous to program flow. That is, the exception is taken before the instruction is allowed to complete. Conditions such as data alignment violations and undefined instructions cause exceptions.

- Interrupts

Asynchronous to program flow. These are caused by input pins, timers, and other peripherals.

Each event has an associated register to hold the return address and an associated return-from-event instruction. When an event is triggered, the state of the processor is saved on the supervisor stack.

The processor event controller consists of two stages: the core event controller (CEC) and the system interrupt controllers (SIC). The CEC works with the SIC to prioritize and control all system events. Conceptually, interrupts from the peripherals arrive at the SIC and are routed directly into the general-purpose interrupts of the CEC.

Core Event Controller (CEC)

The CEC supports nine general-purpose interrupts (IVG15–7), in addition to the dedicated interrupt and exception events. Of these general-purpose interrupts, the two lowest-priority interrupts (IVG15–14) are recommended to be reserved for software interrupt handlers, leaving seven prioritized interrupt inputs to support peripherals.
System Interrupt Controllers (SICx)

The system interrupt controllers provide the mapping and routing of events from the many peripheral interrupt sources to the prioritized general-purpose interrupt inputs of the CEC. Although the processor provides a default mapping, the user can alter the mappings and priorities of interrupt events by writing the appropriate values into the Interrupt Assignment Registers (SIC_IARx).

DMA Support

The processor has two independent DMA controllers that support automated data transfers with minimal overhead for the core. DMA transfers can occur between the internal memories and any of its DMA-capable peripherals. Additionally, DMA transfers can be accomplished between any of the DMA-capable peripherals and external devices connected to the external memory interfaces, including the SDRAM controller and the asynchronous memory controller. DMA-capable peripherals include the SPORTs, SPI ports, UARTs, and PPI. Each individual DMA-capable peripheral has at least one dedicated DMA channel.

The DMA controllers support both 1-dimensional (1D) and 2-dimensional (2D) DMA transfers. DMA transfer initialization can be implemented from registers or from sets of parameters called descriptor blocks.

The 2D DMA capability supports arbitrary row and column sizes up to 64K elements by 64K elements, and arbitrary row and column step sizes up to +/- 32K elements. Furthermore, the column step size can be less than the row step size, allowing implementation of interleaved data streams. This feature is especially useful in video applications where data can be de-interleaved on the fly.
Examples of DMA types supported include:

- A single, linear buffer that stops upon completion
- A circular, auto-refreshing buffer that interrupts on each full or fractionally full buffer
- 1-D or 2-D DMA using a linked list of descriptors
- 2-D DMA using an array of descriptors specifying only the base DMA address within a common page

In addition to the dedicated peripheral DMA channels, there are four separate memory DMA channels provided for transfers between the various memories of the system. This enables transfers of blocks of data between any of the memories—including external SDRAM, ROM, SRAM, and flash memory—with minimal processor intervention. Memory DMA transfers can be controlled by a very flexible descriptor-based methodology or by a standard register-based autobuffer mechanism.

External Bus Interface Unit

The external bus interface unit on the processor interfaces with a wide variety of industry-standard memory devices. The controller consists of an SDRAM controller and an asynchronous memory controller.

PC133 SDRAM Controller

The SDRAM controller provides an interface to a single bank of industry-standard SDRAM devices or DIMMs. Fully compliant with the PC133 SDRAM standard, the bank can be configured to contain between 16 and 128M bytes of memory.
A set of programmable timing parameters is available to configure the SDRAM bank to support slower memory devices. The memory bank is 16 bits wide for minimum device count and lower system cost.

**Asynchronous Controller**

The asynchronous memory controller provides a configurable interface for up to four separate banks of memory or I/O devices. Each bank can be independently programmed with different timing parameters. This allows connection to a wide variety of memory devices, including SRAM, ROM, and flash EPROM, as well as I/O devices that interface with standard memory control lines. Each bank occupies a 1M byte window in the processor address space, but if not fully populated, these are not made contiguous by the memory controller. The banks are 16 bits wide, for interfacing to a range of memories and I/O devices.

**Parallel Peripheral Interface**

The processor provides a parallel peripheral interface (PPI) that can connect directly to parallel A/D and D/A converters, video encoders and decoders, and other general-purpose peripherals. The PPI consists of a dedicated input clock pin, up to 3 frame synchronization pins, and up to 16 data pins. The input clock supports parallel data rates up to SCLK/2, while the synchronization signals can be configured as either inputs or outputs.

The PPI supports a variety of general-purpose and ITU-R 656 modes of operation. In general-purpose mode, the PPI provides half-duplex, bidirectional data transfer with up to 16 bits of data. Up to 3 frame synchronization signals are also provided for controlling DMA transfers. In ITU-R 656 mode, the PPI provides half-duplex, bidirectional data transfer with up to 10 bits of data. Additionally, on-chip decode of embedded start-of-line (SOL) and start-of-field (SOF) preamble packets is supported.
Parallel Peripheral Interface

General-Purpose Mode Descriptions

The GP modes of the PPI are intended to suit a wide variety of data capture and transmission applications. Three distinct sub-modes are supported:

- Input mode – Frame syncs and data are inputs into the PPI.
- Frame capture mode – Frame syncs are outputs from the PPI, but data are inputs.
- Output mode – Frame syncs and data are outputs from the PPI.

Input Mode

This mode is intended for ADC applications, as well as video communication with hardware signaling. In its simplest form, PPI_FS1 is an external frame sync input that controls when to read data. The PPI_DELAY MMR allows for a delay (in PPI_CLK cycles) between reception of this frame sync and the initiation of data reads. The number of input data samples is user-programmable and defined by the contents of the PPI_COUNT register. Data widths of 8, 10, 11, 12, 13, 14, 15, and 16 bits are supported, as programmed by the PPI_CONTROL register.

Frame Capture Mode

This mode allows the video source(s) to act as a slave (for example, for frame capture). The processor controls when to read from the video source(s). PPI_FS1 is an HSYNC output and PPI_FS2 is a VSYNC output.
Output Mode

This mode is used for transmitting video or other data with up to three output frame syncs. Typically, a single frame sync is appropriate for data converter applications, whereas two or three frame syncs could be used for sending video with hardware signaling.

ITU-R 656 Mode Descriptions

The ITU-R 656 modes of the PPI are intended to suit a wide variety of video capture, processing, and transmission applications. Three distinct sub-modes are supported:

- Active Video Only Mode
- Vertical Blanking Only Mode
- Entire Field Mode

Active Video Only Mode

This mode is used when only the active video portion of a field is of interest and not any of the blanking intervals. The PPI does not read in any data between the end of active video (EAV) and start of active video (SAV) preamble symbols, or any data present during the vertical blanking intervals. In this mode, the control byte sequences are not stored to memory; they are filtered by the PPI. After synchronizing to the start of Field 1, the PPI ignores incoming samples until it sees an SAV code. The user specifies the number of active video lines per frame (in PPI_COUNT register).

Vertical Blanking Interval Mode

In this mode, the PPI only transfers vertical blanking interval (VBI) data, as well as horizontal blanking information and control byte sequences on VBI lines.
Entire Field Mode

In this mode, the entire incoming bit stream is read in through the PPI. This includes active video, control preamble sequences, and ancillary data that may be embedded in horizontal and vertical blanking intervals. Data transfer starts immediately after synchronization to Field 1.

Serial Ports (SPORTs)

The processor incorporates four identical dual-channel synchronous serial ports (SPORT0, SPORT1, SPORT2 and SPORT3) for serial and multi-processor communications. The SPORTs support the following features:

- Bidirectional, I²S capable operation
  
  Each SPORT has two sets of independent transmit and receive pins, enabling 16 channels of I²S stereo audio.

- Buffered (8 deep) transmit and receive ports
  
  Each port has a data register for transferring data words to and from other processor components and shift registers for shifting data in and out of the data registers.

- Clocking
  
  Each transmit and receive port can either use an external serial clock or can generate its own in a wide range of frequencies.

- Word length
  
  Each SPORT supports serial data words from 3 to 32 bits in length, transferred in most significant bit first or least significant bit-first format.
Introduction

- Framing

Each transmit and receive port can run with or without frame sync signals for each data word. Frame sync signals can be generated internally or externally, active high or low, and with either of two pulse widths and early or late frame sync.

- Companding in hardware

Each SPORT can perform A-law or μ-law companding according to ITU recommendation G.711. Companding can be selected on the transmit and/or receive channel of the SPORT without additional latencies.

- DMA operations with single cycle overhead

Each SPORT can automatically receive and transmit multiple buffers of memory data. The processor can link or chain sequences of DMA transfers between a SPORT and memory.

- Interrupts

Each transmit and receive port generates an interrupt upon completing the transfer of a data word or after transferring an entire data buffer or buffers through DMA.

- Multichannel capability

Each SPORT supports 128 channels out of a 1024-channel window and is compatible with the H.100, H.110, MVIP-90, and HMVIP standards.
The processor has three SPI-compatible ports that enable the processor to communicate with multiple SPI-compatible devices.

The SPI interface uses three pins for transferring data: two data pins and a clock pin. An SPI chip-select input pin lets other SPI devices select the processor as a slave. SPI chip-select output pins let the processor select other SPI devices. SPI0 has one chip-select input pin and seven chip-select output pins. All are reconfigurable GPIO port F pins. The remaining two SPI instantiations have one chip-select input pin and one chip-select output pin. All of these chip-selects are reconfigurable GPIO port D pins.

Using these pins, the SPI port provides a full-duplex, synchronous, serial interface, which supports both master and slave modes and multimaster environments.

The SPI port baud rate and clock phase/polarities are programmable, and each SPI port has an integrated DMA controller, configurable to support either transmit or receive data streams. The SPI DMA controllers can only service unidirectional accesses at any given time.

During transfers, the SPI ports simultaneously transmit and receive by serially shifting data in and out of their two serial data lines. The serial clock line synchronizes the shifting and sampling of data on the two serial data lines.
Timers

There are four general-purpose programmable timer units in the processor. Three timers have an external pin that can be configured either as a pulse width modulator (PWM) or timer output, as an input to clock the timer, or as a mechanism for measuring pulse widths of external events. These timer units can be synchronized to an external clock input connected to the PF1 pin, an external clock input to the PPI_CLK pin, or to the internal SCLK.

The timer units can be used in conjunction with UART0 to measure the width of the pulses in the data stream to provide an autobaud detect function for a serial channel.

The timers can generate interrupts to the processor core to provide periodic events for synchronization, either to the processor clock or to a count of external signals.

In addition to the three general-purpose programmable timers, a fourth timer is also provided. This extra timer is clocked by the internal processor clock and is typically used as a system tick clock for generation of operating system periodic interrupts.

UART Ports

The processor has three full-duplex Universal Asynchronous Receiver/Transmitter (UART) ports, which are fully compatible with PC-standard UARTs. The UART ports provide a simplified UART interface to other peripherals or hosts, providing full duplex, DMA-supported, asynchronous transfers of serial data. The UART ports include support for 5 to 8 data bits; 1 or 2 stop bits; and none, even, or odd parity. The UART ports support two modes of operation:
UART Ports

- **Programmed I/O (PIO)**

  The processor sends or receives data by writing or reading I/O-mapped UART registers. The data is double buffered on both transmit and receive.

- **Direct memory access (DMA)**

  The DMA controller transfers both transmit and receive data. This reduces the number and frequency of interrupts required to transfer data to and from memory. Each UART has two dedicated DMA channels, one for transmit and one for receive. These DMA channels have lower priority than most DMA channels because of their relatively low service rates.

The UART port baud rate, serial data format, error code generation and status, and interrupts can be programmed to support the following:

- **Wide range of bit rates**
- **Data formats from 7 to 12 bits per frame**
- **Generation of maskable interrupts to the processor by both transmit and receive operations**

In conjunction with the general-purpose timer functions, autobaud detection is supported by UART0.

The capabilities of the UARTs are further extended with support for the Infrared Data Association (IrDA®) Serial Infrared Physical Layer Link Specification (SIR) protocol.
Controller Area Network Port

The controller area network port (CAN) provides a two-wire interface for communication with other CAN compliant devices. Features of the CAN port include error detection, multimastering, prioritization of messages through arbitration, and a 32-entry mailbox RAM. Transfer rates typically approach 1M bps.

Two-Wire Interface Port

The processor has two TWI (two-wire interface) ports that support synchronous serial transfers over a two-wire system with $I^2C$ compliant devices. Features include simultaneous master and slave operation, multimaster arbitration, 400K bps data rates, master clock synchronization, and 7-bit addressing.

Real-Time Clock

The Blackfin real-time clock (RTC) provides a robust set of digital watch features, including current time, stopwatch, and alarm. The RTC is clocked by a 32.768 KHz crystal external to the processor. The RTC peripheral has dedicated power supply pins, so that it can remain powered up and clocked even when the rest of the processor is in a low power state. The RTC provides several programmable interrupt options, including interrupt per second, minute, hour, or day clock ticks, interrupt on programmable stopwatch countdown, or interrupt at a programmed alarm time.

The 32.768 KHz input clock frequency is divided down to a 1 Hz signal by a prescaler. The counter function of the timer consists of four counters: a 60 second counter, a 60 minute counter, a 24 hours counter, and a 32768 day counter.
When enabled, the alarm function generates an interrupt when the output of the timer matches the programmed value in the alarm control register. There are two alarms: The first alarm is for a time of day. The second alarm is for a day and time of that day.

The stopwatch function counts down from a programmed value, with one minute resolution. When the stopwatch is enabled and the counter under-flows, an interrupt is generated.

Like the other peripherals, the RTC can wake up the processor from a low power state upon generation of any RTC wake-up event.

Watchdog Timer

The processor includes a 32-bit timer that can be used to implement a software watchdog function. A software watchdog can improve system availability by forcing the processor to a known state through generation of a hardware reset, non-maskable interrupt (NMI), or general-purpose interrupt, if the timer expires before being reset by software. The programmer initializes the count value of the timer, enables the appropriate interrupt, then enables the timer. Thereafter, the software must reload the counter before it counts to zero from the programmed value. This protects the system from remaining in an unknown state where software that would normally reset the timer has stopped running due to an external noise condition or software error.

If configured to generate a hardware reset, the watchdog timer resets both the CPU and the peripherals. After a reset, software can determine if the watchdog was the source of the hardware reset by interrogating a status bit in the watchdog control register.

The timer is clocked by the system clock ($SCLK$), at a maximum frequency of $f_{SCLK}$.
General-Purpose I/O

There are up to 54 general-purpose I/O (GPIO) pins on the processor which span four ports—C, D, E, and F.

The GPIO ports C, D, and E functionality is muxed with peripheral pins. By default, the peripheral function is selected. Through software, the GPIO functionality can be selected for the pin instead.

The GPIO pins may be individually selected on a pin by pin basis; so that, for example, if all the pins of a SPORT are not required, the remainder may be used as GPIO. GPIO interrupt sensitivity registers – The two GPIO interrupt sensitivity registers specify whether individual PFx pins are level or edge sensitive and specify—if edge sensitive—whether just the rising edge or both the rising and falling edges of the signal are significant. One register selects the type of sensitivity, and one register selects which edges are significant for edge sensitivity.

Clock Signals

The processor can be clocked by an external crystal, a sine wave input, or a buffered, shaped clock derived from an external clock oscillator.

This external clock connects to the Blackfin CLKIN pin. CLKIN input cannot be halted, changed, or operated below the specified frequency during normal operation. This clock signal should be a TTL-compatible signal.

The core clock (CCLK) and system peripheral clock (SCLK) are derived from the input clock (CLKIN) signal. An on-chip PLL is capable of multiplying the CLKIN signal by a user programmable multiplication factor. The default multiplier is 10x, but it can be modified by a software instruction sequence. On-the-fly frequency changes can be made by simply writing to the PLL_DIV register.
All on-chip peripherals are clocked by the system clock ($SCLK$). The system clock frequency is programmable by means of the $SSEL[3:0]$ bits of the PLL\_DIV register.

The CAN clock is derived from the system clock ($SCLK$), through a further divisor. Careful selection of the input clock and $SCLK$ is important to obtain the correct CAN clock frequency.

**Dynamic Power Management**

The processor provides four operating modes, each with a different performance/power profile. In addition, dynamic power management provides the control functions to dynamically alter the processor core supply voltage to further reduce power dissipation. Control of clocking to each of the peripherals also reduces power consumption.

**Full On Operating Mode (Maximum Performance)**

In the full on mode, the phase-locked loop (PLL) is enabled, not bypassed, providing the maximum operational frequency. This is the normal execution state in which maximum performance can be achieved. The processor core and all enabled peripherals run at full speed.

**Active Operating Mode (Moderate Power Savings)**

In the active mode, the PLL is enabled, but bypassed. Because the PLL is bypassed, the Blackfin core clock ($CCLK$) and system clock ($SCLK$) run at the input clock ($CLKIN$) frequency. In this mode, the $CLKIN$ to VCO multiplier ratio can be changed, although the changes are not realized until the full on mode is entered. DMA access is available to appropriately configured L1 memories.
Introduction

In the active mode, it is possible to disable the PLL through the PLL control register (PLL_CTL). If disabled, the PLL must be re-enabled before transitioning to the full on or sleep modes.

Sleep Operating Mode (High Power Savings)

The sleep mode reduces power consumption by disabling the clock to the processor core. The sleep mode reduces power dissipation by disabling the clock to the processor core (CCLK). The PLL and system clock (SCLK), however, continue to operate in this mode. Typically an external event or RTC activity wakes up the processor. When in the sleep mode, assertion of any interrupt causes the processor to sense the value of the bypass bit (BYPASS) in the PLL control register (PLL_CTL). If bypass is disabled, the processor transitions to the full on mode. If bypass is enabled, the processor transitions to the active mode.

When in the sleep mode, system DMA access to L1 memory is not supported.

Deep Sleep Operating Mode (Maximum Power Savings)

The deep sleep mode maximizes power savings by disabling the clocks to the processor core and to all synchronous systems. Asynchronous systems, such as the RTC, may still be running, but can not access internal resources or external memory. This powered-down mode can only be exited by assertion of the reset interrupt or by an asynchronous interrupt generated by the RTC. When in deep sleep mode, assertion of the reset interrupt or the RTC asynchronous interrupt causes the processor to transition to the active mode.
Hibernate State

For lowest possible power dissipation, this state allows the internal supply ($V_{DDINT}$) to be powered down, while keeping the I/O supply ($V_{DDEXT}$) running. Although not strictly an operating mode like the four modes detailed above, it is illustrative to view it as such.

The processor can be programmed to wake up from hibernate by reset, the RTC, a general-purpose event, or the CAN.

Voltage Regulation

The processor provides an on-chip voltage regulator that can generate internal voltage levels from an external 2.25 V to 3.6 V supply. The regulator controls the internal logic voltage levels and is programmable with the voltage regulator control register ($VR_{CTL}$) in increments of 50 mV. The regulator can also be disabled and bypassed at user discretion.

Boot Modes

The processor has three mechanisms for automatically loading internal L1 instruction memory after a reset. A fourth mode is provided to execute from external memory, bypassing the boot sequence:

- Execute from 16-bit external memory—Execution starts from address $0x2000\ 0000$ with 16-bit packing. The boot ROM is bypassed in this mode. All configuration settings are set for the slowest device possible (3-cycle hold time; 15-cycle R/W access times; 4-cycle setup).
Introduction

- Boot from 8-bit or 16-bit flash memory—The 8-bit flash boot routine located in boot ROM memory space is set up using asynchronous memory bank 0. All configuration settings are set for the slowest device possible (3-cycle hold time; 15-cycle R/W access times; 4-cycle setup). For ADSP-BF538F processors, the on-chip flash memory can be booted from when the flash is mapped to asynchronous bank 0.

- Boot from an SPI host in SPI slave Mode—The SPI0 is configured as an SPI slave device and a host is used to boot the processor.

- Boot from an 8-/16-/24-bit addressable SPI in SPI master mode—Support for Atmel AT45DB041B, AT45DB081B, AT45D161B Data Flash® devices. The SPI0 uses the PF2 output pin to select a single SPI EEPROM device.

For each of the boot modes, a 10-byte header is first read from an external memory device. The header specifies the number of bytes to be transferred and the memory destination address. Multiple memory blocks may be loaded by any boot sequence. Once all blocks are loaded, program execution commences from the start of L1 instruction SRAM (0xFFFFA0 0000).

In addition, bit 4 of the reset configuration register can be set by application code to bypass the normal boot sequence during a software reset. For this case, the processor jumps directly to the beginning of L1 instruction memory.

To augment the boot modes, a secondary software loader is provided that adds additional booting mechanisms. This secondary loader provides the capability to boot from 16-bit flash memory, fast flash, variable baud rate memory, and other sources.
Instruction Set Description

The Blackfin processor family assembly language instruction set employs an algebraic syntax designed for ease of coding and readability. The instructions have been specifically tuned to provide a flexible, densely encoded instruction set that compiles to a very small final memory size. The instruction set also provides fully featured multifunction instructions that allow the programmer to use many of the processor core resources in a single instruction. Coupled with many features more often seen on microcontrollers, this instruction set is very efficient when compiling C and C++ source code. In addition, the architecture supports both user (algorithm/application code) and supervisor (O/S kernel, device drivers, debuggers, ISRs) modes of operation, allowing multiple levels of access to core resources.

The assembly language, which takes advantage of the Blackfin unique architecture, offers the following advantages:

- Seamlessly integrated DSP/CPU features are optimized for both 8-bit and 16-bit operations.
- A multi-issue load/store modified-Harvard architecture, which supports two 16-bit MAC or four 8-bit ALU + two load/store + two pointer updates per cycle.
- All registers, I/O, and memory are mapped into a unified 4G byte memory space, providing a simplified programming model.
- Microcontroller features, such as arbitrary bit and bit-field manipulation, insertion, and extraction; integer operations on 8-, 16-, and 32-bit data-types; and separate user and supervisor stack pointers.
- Code density enhancements include intermixing of 16- and 32-bit instructions with no mode switching or code segregation. Frequently used instructions are encoded in 16 bits.
Development Tools

The processor is supported by a complete set of software and hardware development tools, including Analog Devices' emulators and the Cross-Core Embedded Studio or VisualDSP++ development environment. (The emulator hardware that supports other Analog Devices processors also emulates the processor.)

The development environments support advanced application code development and debug with features such as:

- Create, compile, assemble, and link application programs written in C++, C, and assembly
- Load, run, step, halt, and set breakpoints in application programs
- Read and write data and program memory
- Read and write core and peripheral registers
- Plot memory

Analog Devices DSP emulators use the IEEE 1149.1 JTAG test access port to monitor and control the target board processor during emulation. The emulator provides full speed emulation, allowing inspection and modification of memory, registers, and processor stacks. Nonintrusive in-circuit emulation is assured by the use of the processor JTAG interface—the emulator does not affect target system loading or timing.

Software tools also include Board Support Packages (BSPs). Hardware tools also include standalone evaluation systems (boards and extenders). In addition to the software and hardware development tools available from Analog Devices, third parties provide a wide range of tools supporting the Blackfin processors. Third party software tools include DSP libraries, real-time operating systems, and block diagram design tools.
Development Tools
2  COMPUTATIONAL UNITS

The processor’s computational units perform numeric processing for DSP and general control algorithms. The six computational units are two arithmetic/logic units (ALUs), two multiplier/accumulator (multiplier) units, a shifter, and a set of video ALUs. These units get data from registers in the data register file. Computational instructions for these units provide fixed-point operations, and each computational instruction can execute every cycle.

The computational units handle different types of operations. The ALUs perform arithmetic and logic operations. The multipliers perform multiplication and execute multiply/add and multiply/subtract operations. The shifter executes logical shifts and arithmetic shifts and performs bit packing and extraction. The video ALUs perform single-instruction, multiple-data (SIMD) logical operations on specific 8-bit data operands.

Data moving in and out of the computational units goes through the data register file, which consists of eight registers, each 32 bits wide. In operations requiring 16-bit operands, the registers are paired, providing sixteen possible 16-bit registers.

The processor’s assembly language provides access to the data register file. The syntax lets programs move data to and from these registers and specify a computation’s data format at the same time.

Figure 2-1 provides a graphical guide to the other topics in this chapter. An examination of each computational unit provides details about its operation and is followed by a summary of computational instructions. Studying the details of the computational units, register files, and data
buses leads to a better understanding of proper data flow for computations. Next, details about the processor’s advanced parallelism reveal how to take advantage of multifunction instructions.

Figure 2-1 shows the relationship between the data register file and the computational units—multipliers, ALUs, and shifter.

Figure 2-1. Processor Core Architecture

Single function multiplier, ALU, and shifter instructions have unrestricted access to the data registers in the data register file. Multifunction operations may have restrictions that are described in the section for that particular operation.
Computational Units

Two additional registers, A0 and A1, provide 40-bit accumulator results. These registers are dedicated to the ALUs and are used primarily for multiply-and-accumulate functions.

The traditional modes of arithmetic operations, such as fractional and integer, are specified directly in the instruction. Rounding modes are set from the $\text{ASTAT}$ register, which also records status and conditions for the results of the computational operations.

Using Data Formats

Blackfin processors are primarily 16-bit, fixed-point machines. Most operations assume a two’s-complement number representation, while others assume unsigned numbers or simple binary strings. Other instructions support 32-bit integer arithmetic, with further special features supporting 8-bit arithmetic and block floating point. For detailed information about each number format, see Appendix D, “Numeric Formats”.

In the Blackfin processor family arithmetic, signed numbers are always in two’s-complement format. These processors do not use signed-magnitude, one’s-complement, binary-coded decimal (BCD), or excess-n formats.

Binary String

The binary string format is the least complex binary notation; in it, 16 bits are treated as a bit pattern. Examples of computations using this format are the logical operations NOT, AND, OR, XOR. These ALU operations treat their operands as binary strings with no provision for sign bit or binary point placement.
Using Data Formats

Unsigned

Unsigned binary numbers may be thought of as positive and having nearly twice the magnitude of a signed number of the same length. The processor treats the least significant words of multiple precision numbers as unsigned numbers.

Signed Numbers: Two’s-Complement

In Blackfin processor arithmetic, the word signed refers to two’s-complement numbers. Most Blackfin processor family operations presume or support two’s-complement arithmetic.

Fractional Representation: 1.15

Blackfin processor arithmetic is optimized for numerical values in a fractional binary format denoted by 1.15 (“one dot fifteen”). In the 1.15 format, 1 sign bit (the most significant bit (MSB)) and 15 fractional bits represent values from –1 to 0.999969.

Figure 2-2 shows the bit weighting for 1.15 numbers as well as some examples of 1.15 numbers and their decimal equivalents.

<table>
<thead>
<tr>
<th>1.15 NUMBER (HEXADECIMAL)</th>
<th>DECIMAL EQUIVALENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0001</td>
<td>0.000031</td>
</tr>
<tr>
<td>0x7FFF</td>
<td>0.999969</td>
</tr>
<tr>
<td>0xFFFF</td>
<td>–0.000031</td>
</tr>
<tr>
<td>0x8000</td>
<td>–1.000000</td>
</tr>
</tbody>
</table>

Figure 2-2. Bit Weighting for 1.15 Numbers
Register Files

The processor’s computational units have three definitive register groups—a data register file, a pointer register file, and set of data address generator (DAG) registers.

- The data register file receives operands from the data buses for the computational units and stores computational results.
- The pointer register file has pointers for addressing operations.
- The DAG registers are dedicated registers that manage zero-overhead circular buffers for DSP operations.

For more information, see Chapter 5, “Data Address Generators”.

The processor register files appear in Figure 2-3.

![Figure 2-3. Register Files](image-url)
In the processor, a word is 32 bits long; H denotes the high order 16 bits of a 32-bit register; L denotes the low order 16 bits of a 32-bit register. For example, $A0.W$ contains the lower 32 bits of the 40-bit $A0$ register; $A0.L$ contains the lower 16 bits of $A0.W$, and $A0.H$ contains the upper 16 bits of $A0.W$.

**Data Register File**

The data register file consists of eight registers, each 32 bits wide. Each register may be viewed as a pair of independent 16-bit registers. Each is denoted as the low half or high half. Thus the 32-bit register $R0$ may be regarded as two independent register halves, $R0.L$ and $R0.H$.

Three separate buses (two read, one write) connect the register File to the L1 data memory, each bus being 32 bits wide. Transfers between the data register file and the data memory can move up to four 16-bit words of valid data in each cycle.

**Accumulator Registers**

In addition to the data register file, the processor has two dedicated, 40-bit accumulator registers. Each can be referred to as its 16-bit low half ($A_n.L$) or high half ($A_n.H$) plus its 8-bit extension ($A_n.X$). Each can also be referred to as a 32-bit register ($A_n.W$) consisting of the lower 32 bits, or as a complete 40-bit result register ($A_n$).
Pointer Register File

The general-purpose address pointer registers, also called P-registers, are organized as:

- 6-entry, P-register files $P[5:0]$
- Frame pointers (FP) used to point to the current procedure’s activation record
- Stack pointer registers (SP) used to point to the last used location on the runtime stack. See mode dependent registers in Chapter 3, “Operating Modes and States”.

P-registers are 32 bits wide. Although P-registers are primarily used for address calculations, they may also be used for general integer arithmetic with a limited set of arithmetic operations; for instance, to maintain counters. However, unlike the data registers, P-register arithmetic does not affect the arithmetic register (ASTAT) register status flags.

DAG Register Set

DSP instructions primarily use the data address generator (DAG) register set for addressing. The DAG register set consists of these registers:

- $I[3:0]$ contain index addresses
- $M[3:0]$ contain modify values
- $B[3:0]$ contain base addresses
- $L[3:0]$ contain length values

All DAG registers are 32 bits wide.
Register Files

The I (index) registers and B (base) registers always contain addresses of 8-bit bytes in memory. The index registers contain an effective address. The M (modify) registers contain an offset value that is added to one of the Index registers or subtracted from it.

The B and L (length) registers define circular buffers. The B register contains the starting address of a buffer, and the L register contains the length in bytes. Each L and B register pair is associated with the corresponding I register. For example, L0 and B0 are always associated with I0. However, any M register may be associated with any I register. For example, I0 may be modified by M3. For more information, see Chapter 5, “Data Address Generators”.

Register File Instruction Summary

Table 2-1 lists the register file instructions. For more information about assembly language syntax, see Blackfin Processor Programming Reference.

In Table 2-1, note the meaning of these symbols:

- **Allreg** denotes: R[7:0], P[5:0], SP, FP, I[3:0], M[3:0], B[3:0], L[3:0], A0.X, A0.W, A1.X, A1.W, ASTAT, RETS, RETI, RETX, RETN, RETE, LC[1:0], LT[1:0], LB[1:0], USP, SEQSTAT, SYSCFG, CYCLES, and CYCLES2.

- **An** denotes either ALU result register A0 or A1.

- **Dreg** denotes any data register file register.

- **Sysreg** denotes the system registers: ASTAT, SEQSTAT, SYSCFG, RETI, RETX, RETN, RETE, or RETS, LC[1:0], LT[1:0], LB[1:0], CYCLES, and CYCLES2.

- **Preg** denotes any pointer register, FP, or SP register.

- **Dreg_even** denotes R0, R2, R4, or R6.
Computational Units

- Dreg_odd denotes R1, R3, R5, or R7.
- DPreg denotes any data register file register or any pointer register, FP, or SP register.
- Dreg_lo denotes the lower 16 bits of any data register file register.
- Dreg_hi denotes the upper 16 bits of any data register file register.
- An.L denotes the lower 16 bits of accumulator A0.W or A1.W.
- An.H denotes the upper 16 bits of accumulator A0.W or A1.W.
- Dreg_byte denotes the low order 8 bits of each data register.
- Option (X) denotes sign-extended.
- Option (Z) denotes zero-extended.
- * Indicates the flag may be set or cleared, depending on the result of the instruction.
- ** Indicates the flag is cleared.
- – Indicates no effect.

Table 2-1. Register File Instruction Summary

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ASTAT Status Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AZ</td>
</tr>
<tr>
<td>allreg = allreg ; ¹</td>
<td>–</td>
</tr>
<tr>
<td>An = An ;</td>
<td>–</td>
</tr>
<tr>
<td>An = Dreg ;</td>
<td>–</td>
</tr>
<tr>
<td>Dreg_even = A0 ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_odd = A1 ;</td>
<td>*</td>
</tr>
</tbody>
</table>
### Table 2-1. Register File Instruction Summary (Cont’d)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ASTAT Status Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AZ</td>
</tr>
<tr>
<td>Dreg_even = A0, Dreg_odd = A1 ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_odd = A1, Dreg_even = A0 ;</td>
<td>*</td>
</tr>
<tr>
<td>IF CC Dpreg = Dpreg ;</td>
<td>-</td>
</tr>
<tr>
<td>IF ! CC Dpreg = Dpreg ;</td>
<td>-</td>
</tr>
<tr>
<td>Dreg = Dreg_lo (Z) ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = Dreg_lo (X) ;</td>
<td>*</td>
</tr>
<tr>
<td>An.X = Dreg_lo ;</td>
<td>-</td>
</tr>
<tr>
<td>Dreg_lo = An.X ;</td>
<td>-</td>
</tr>
<tr>
<td>An.L = Dreg_lo ;</td>
<td>-</td>
</tr>
<tr>
<td>An.H = Dreg_hi ;</td>
<td>-</td>
</tr>
<tr>
<td>Dreg_lo = A0 ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_hi = A1 ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_hi = A1 ; Dreg_lo = A0 ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_lo = A0 ; Dreg_hi = A1 ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = Dreg_byte (Z) ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = Dreg_byte (X) ;</td>
<td>*</td>
</tr>
</tbody>
</table>

1 Warning: Not all register combinations are allowed. For details, see the functional description of the Move Register instruction in *Blackfin Processor Programming Reference.*
Data Types

The processor supports 32-bit words, 16-bit half words, and bytes. The 32- and 16-bit words can be integer or fractional, but bytes are always integers. Integer data types can be signed or unsigned, but fractional data types are always signed.

Table 2-3 illustrates the formats for data that resides in memory, in the register file, and in the accumulators. In the table, the letter \( d \) represents one bit, and the letter \( s \) represents one signed bit.

Some instructions manipulate data in the registers by sign-extending or zero-extending the data to 32 bits:

- Instructions zero-extend unsigned data
- Instructions sign-extend signed 16-bit half words and 8-bit bytes

Other instructions manipulate data as 32-bit numbers. In addition, two 16-bit half words or four 8-bit bytes can be manipulated as 32-bit values. For details, refer to the instructions in Blackfin Processor Programming Reference.

In Table 2-2, note the meaning of these symbols:

- \( s \) = sign bit(s)
- \( d \) = data bit(s)
- “.” = decimal point by convention; however, a decimal point does not literally appear in the number.
- Italics denotes data from a source other than adjacent bits.
### Table 2-2. Data Formats

<table>
<thead>
<tr>
<th>Format</th>
<th>Representation in Memory</th>
<th>Representation in 32-Bit Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>32.0 Unsigned Word</td>
<td>dddd dddd dddd dddd dddd</td>
<td>dddd dddd dddd dddd dddd dddd dddd</td>
</tr>
<tr>
<td>32.0 Signed Word</td>
<td>sddd dddd dddd dddd dddd</td>
<td>sddd dddd dddd dddd dddd dddd dddd</td>
</tr>
<tr>
<td>16.0 Unsigned Half Word</td>
<td>dddd dddd dddd dddd</td>
<td>0000 0000 0000 0000 dddd dddd dddd</td>
</tr>
<tr>
<td>16.0 Signed Half Word</td>
<td>sddd dddd dddd dddd</td>
<td>ssss ssss ssss ssss dddd dddd dddd</td>
</tr>
<tr>
<td>8.0 Unsigned Byte</td>
<td>dddd dddd</td>
<td>0000 0000 0000 0000 0000 0000 0000 dddd dddd</td>
</tr>
<tr>
<td>8.0 Signed Byte</td>
<td>sddd dddd</td>
<td>ssss ssss ssss ssss ssss ssss dddd dddd</td>
</tr>
<tr>
<td>0.16 Unsigned Fraction</td>
<td>.ddd dddd dddd dddd</td>
<td>0000 0000 0000 0000 .ddd dddd dddd dddd</td>
</tr>
<tr>
<td>1.15 Signed Fraction</td>
<td>s.ddd dddd dddd dddd</td>
<td>ssss ssss ssss ssss ssss ssss ssss ssss dddd dddd</td>
</tr>
<tr>
<td>0.32 Unsigned Fraction</td>
<td>.ddd dddd dddd dddd dddd dddd dddd dddd</td>
<td>.ddd dddd dddd dddd dddd dddd dddd dddd dddd</td>
</tr>
<tr>
<td>1.31 Signed Fraction</td>
<td>s.ddd dddd dddd dddd dddd dddd dddd dddd dddd dddd</td>
<td>s.ddd dddd dddd dddd dddd dddd dddd dddd dddd dddd</td>
</tr>
<tr>
<td>Packed 8.0 Unsigned Byte</td>
<td>dddd dddd dddd dddd dddd dddd</td>
<td>dddd dddd dddd dddd dddd dddd dddd dddd dddd</td>
</tr>
<tr>
<td>Packed 0.16 Unsigned Fraction</td>
<td>.ddd dddd dddd dddd .ddd dddd dddd dddd dddd dddd</td>
<td>.ddd dddd dddd dddd .ddd dddd dddd dddd dddd dddd</td>
</tr>
<tr>
<td>Packed 1.15 Signed Fraction</td>
<td>s.ddd dddd dddd dddd s.ddd dddd dddd dddd dddd</td>
<td>s.ddd dddd dddd dddd s.ddd dddd dddd dddd dddd dddd</td>
</tr>
</tbody>
</table>


**Endian Byte Order**

Both internal and external memory are accessed in little endian byte order. For more information, see “Memory Transaction Model” on page 6-62.

**ALU Data Types**

Operations on each ALU treat operands and results as either 16- or 32-bit binary strings, except the signed division primitive (DIVS). ALU result status bits treat the results as signed, indicating status with the overflow flags (AV0, AV1) and the negative flag (AN). Each ALU has its own sticky overflow flag, AV0S and AV1S. Once set, these bits remain set until cleared by writing directly to the ASTAT register. An additional V flag is set or cleared depending on the transfer of the result from both accumulators to the register file. Furthermore, the sticky VS bit is set with the V bit and remains set until cleared.

The logic of the overflow bits (V, VS, AV0, AV0S, AV1, AV1S) is based on two’s-complement arithmetic. A bit or set of bits is set if the most significant bit (MSB) changes in a manner not predicted by the signs of the operands and the nature of the operation. For example, adding two positive numbers must generate a positive result; a change in the sign bit signifies an overflow and sets AVn, the corresponding overflow flags. Adding a negative and a positive number may result in either a negative or positive result, but cannot cause an overflow.

The logic of the carry bits (AC0, AC1) is based on unsigned magnitude arithmetic. The bit is set if a carry is generated from bit 16 (the MSB). The carry bits (AC0, AC1) are most useful for the lower word portions of a multiword operation.

ALU results generate status information. For more information about using ALU status, see “ALU Instruction Summary” on page 2-29.
Multipliers Data Types

Each multiplier produces results that are binary strings. The inputs are interpreted according to the information given in the instruction itself (whether it is signed multiplied by signed, unsigned multiplied by unsigned, a mixture, or a rounding operation). The 32-bit result from the multipliers is assumed to be signed; it is sign-extended across the full 40-bit width of the \( A0 \) or \( A1 \) registers.

The processor supports two modes of format adjustment: the fractional mode for fractional operands (1.15 format with 1 sign bit and 15 fractional bits) and the integer mode for integer operands (16.0 format).

When the processor multiplies two 1.15 operands, the result is a 2.30 (2 sign bits and 30 fractional bits) number. In the fractional mode, the multiplier automatically shifts the multiplier product left one bit before transferring the result to the multiplier result register \( (A0, A1) \). This shift of the redundant sign bit causes the multiplier result to be in 1.31 format, which can be rounded to 1.15 format. The resulting format appears in Figure 2-4.

In the integer mode, the left shift does not occur. For example, if the operands are in the 16.0 format, the 32-bit multiplier result would be in 32.0 format. A left shift is not needed and would change the numerical representation. This result format appears in Figure 2-5.

Multiplier results generate status information when they update accumulators or when they are transferred to a destination register in the register file. For more information, see “Multiplier Instruction Summary” on page 2-40.


**Shifter Data Types**

Many operations in the shifter are explicitly geared to signed (two’s-complement) or unsigned values—logical shifts assume unsigned magnitude or binary string values, and arithmetic shifts assume two’s-complement values.

The exponent logic assumes two’s-complement numbers. The exponent logic supports block floating point, which is also based on two’s-complement fractions.

Shifter results generate status information. For more information about using shifter status, see “Shifter Instruction Summary” on page 2-54

**Arithmetic Formats Summary**

Table 2-3, Table 2-4, Table 2-5, and Table 2-6 summarize some of the arithmetic characteristics of computational operations.

Table 2-3. ALU Arithmetic Formats

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operand Formats</th>
<th>Result Formats</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>Signed or unsigned</td>
<td>Interpret flags</td>
</tr>
<tr>
<td>Subtraction</td>
<td>Signed or unsigned</td>
<td>Interpret flags</td>
</tr>
<tr>
<td>Logical</td>
<td>Binary string</td>
<td>Same as operands</td>
</tr>
<tr>
<td>Division</td>
<td>Explicitly signed or unsigned</td>
<td>Same as operands</td>
</tr>
</tbody>
</table>
Using Multiplier Integer and Fractional Formats

For multiply-and-accumulate functions, the processor provides two choices—fractional arithmetic for fractional numbers (1.15) and integer arithmetic for integers (16.0).
For fractional arithmetic, the 32-bit product output is format adjusted—sign-extended and shifted one bit to the left—before being added to accumulator $A_0$ or $A_1$. For example, bit 31 of the product lines up with bit 32 of $A_0$ (which is bit 0 of $A_0.X$), and bit 0 of the product lines up with bit 1 of $A_0$ (which is bit 1 of $A_0.W$). The least significant bit (LSB) is zero-filled. The fractional multiplier result format appears in Figure 2-4.

For integer arithmetic, the 32-bit product register is not shifted before being added to $A_0$ or $A_1$. Figure 2-5 shows the integer mode result placement.

With either fractional or integer operations, the multiplier output product is fed into a 40-bit adder/subtracter which adds or subtracts the new product with the current contents of the $A_0$ or $A_1$ register to produce the final 40-bit result.


Data Types

Rounding Multiplier Results

On many multiplier operations, the processor supports multiplier results rounding (RND option). Rounding is a means of reducing the precision of a number by removing a lower order range of bits from that number’s representation and possibly modifying the remaining portion of the number to more accurately represent its former value. For example, the original number will have N bits of precision, whereas the new number will have only M bits of precision (where N>M). The process of rounding, then, removes N – M bits of precision from the number.

The RND_MOD bit in the ASTAT register determines whether the RND option provides biased or unbiased rounding. For unbiased rounding, set RND_MOD bit = 0. For biased rounding, set RND_MOD bit = 1.

For most algorithms, unbiased rounding is preferred.
Unbiased Rounding

The convergent rounding method returns the number closest to the original. In cases where the original number lies exactly halfway between two numbers, this method returns the nearest even number, the one containing an LSB of 0. For example, when rounding the 3-bit, two’s-complement fraction 0.25 (binary 0.01) to the nearest 2-bit, two’s-complement fraction, the result would be 0.0, because that is the even-numbered choice of 0.5 and 0.0. Since it rounds up and down based on the surrounding values, this method is called unbiased rounding.

Unbiased rounding uses the ALU capability of rounding the 40-bit result at the boundary between bit 15 and bit 16. Rounding can be specified as part of the instruction code. When rounding is selected, the output register contains the rounded 16-bit result; the accumulator is never rounded.

The accumulator uses an unbiased rounding scheme. The conventional method of biased rounding adds a 1 into bit position 15 of the adder chain. This method causes a net positive bias because the midway value (when \( A0.L/A1.L = 0x8000 \)) is always rounded upward.

The accumulator eliminates this bias by forcing bit 16 in the result output to 0 when it detects this midway point. Forcing bit 16 to 0 has the effect of rounding odd \( A0.L/A1.L \) values upward and even values downward, yielding a large sample bias of 0, assuming uniformly distributed values.

The following examples use \( x \) to represent any bit pattern (not all zeros). The example in Figure 2-6 shows a typical rounding operation for \( A0 \); the example also applies for \( A1 \).

The compensation to avoid net bias becomes visible when all lower 15 bits are 0 and bit 15 is 1 (the midpoint value) as shown in Figure 2-7.

In Figure 2-7, \( A0 \) bit 16 is forced to 0. This algorithm is employed on every rounding operation, but is evident only when the bit patterns shown in the lower 16 bits of the next example are present.
Biased Rounding

The *round-to-nearest* method also returns the number closest to the original. However, by convention, an original number lying exactly halfway between two numbers always rounds up to the larger of the two. For example, when rounding the 3-bit, two’s-complement fraction 0.25 (binary 0.01) to the nearest 2-bit, two’s-complement fraction, this method returns 0.5 (binary 0.1). The original fraction lies exactly midway between 0.5 and 0.0 (binary 0.0), so this method rounds up. Because it always rounds up, this method is called *biased* rounding.

The `RND_MOD` bit in the `ASTAT` register enables biased rounding. When the `RND_MOD` bit is cleared, the `RND` option in multiplier instructions uses the normal, unbiased rounding operation, as discussed in “Unbiased Rounding” on page 2-19.
Computational Units

When the RND_MOD bit is set (=1), the processor uses biased rounding instead of unbiased rounding. When operating in biased rounding mode, all rounding operations with A0.L/A1.L set to 0x8000 round up, rather than only rounding odd values up. For an example of biased rounding, see Table 2-7.

Figure 2-7. Avoiding Net Bias in Unbiased Multiplier Rounding

When the RND_MOD bit is set (=1), the processor uses biased rounding instead of unbiased rounding. When operating in biased rounding mode, all rounding operations with A0.L/A1.L set to 0x8000 round up, rather than only rounding odd values up. For an example of biased rounding, see Table 2-7.
Data Types

Table 2-7. Biased Rounding in Multiplier Operation

<table>
<thead>
<tr>
<th>A0/A1 Before RND</th>
<th>Biased RND Result</th>
<th>Unbiased RND Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00 0000 8000</td>
<td>0x00 0001 8000</td>
<td>0x00 0000 0000</td>
</tr>
<tr>
<td>0x00 0001 8000</td>
<td>0x00 0002 0000</td>
<td>0x00 0002 0000</td>
</tr>
<tr>
<td>0x00 0000 8001</td>
<td>0x00 0001 0001</td>
<td>0x00 0001 0001</td>
</tr>
<tr>
<td>0x00 0001 8001</td>
<td>0x00 0002 0001</td>
<td>0x00 0002 0001</td>
</tr>
<tr>
<td>0x00 0000 7FFF</td>
<td>0x00 0000 FFFF</td>
<td>0x00 0000 FFFF</td>
</tr>
<tr>
<td>0x00 0001 7FFF</td>
<td>0x00 0001 FFFF</td>
<td>0x00 0001 FFFF</td>
</tr>
</tbody>
</table>

Biased rounding affects the result only when the A0.L/A1.L register contains 0x8000; all other rounding operations work normally. This mode allows more efficient implementation of bit specified algorithms that use biased rounding (for example, the global system for mobile communications (GSM) speech compression routines).

Truncation

Another common way to reduce the significant bits representing a number is to simply mask off the N – M lower bits. This process is known as truncation and results in a relatively large bias. Instructions that do not support rounding revert to truncation. The RND_MOD bit in ASTAT has no effect on truncation.

Special Rounding Instructions

The ALU provides the ability to round the arithmetic results directly into a data register with biased or unbiased rounding as described above. It also provides the ability to round on different bit boundaries. The options RND12, RND, and RND20 extract 16-bit values from bit 12, bit 16 and bit 20, respectively, and perform biased rounding regardless of the state of the RND_MOD bit in ASTAT.
For example:

\[ R3.L = R4 \text{ (RND)}; \] performs biased rounding at bit 16, depositing the result in a half word.

\[ R3.L = R4 + R5 \text{ (RND12)}; \] performs an addition of two 32-bit numbers, biased rounding at bit 12, depositing the result in a half word.

\[ R3.L = R4 + R5 \text{ (RND20)}; \] performs an addition of two 32-bit numbers, biased rounding at bit 20, depositing the result in a half word.

### Using Computational Status

The multiplier, ALU, and shifter update the overflow and other status flags in the processor’s Arithmetic register (\text{ASTAT}) register. To use status conditions from computations in program sequencing, use conditional instructions to test the \text{CC} flag in the \text{ASTAT} register after the instruction executes. This method permits monitoring each instruction’s outcome. The \text{ASTAT} register is a 32-bit register, with some bits reserved. To ensure compatibility with future implementations, writes to this register should write back the values read from these reserved bits.

### ASTAT Register

\text{Figure 2-8} describes the arithmetic register (\text{ASTAT}) register. The processor updates the status bits in \text{ASTAT}, indicating the status of the most recent ALU, multiplier, or shifter operation.
Figure 2-8. Arithmetic Status Register
Arithmetic Logic Unit (ALU)

The two ALUs perform arithmetic and logical operations on fixed-point data. ALU fixed-point instructions operate on 16-, 32-, and 40-bit fixed-point operands and output 16-, 32-, or 40-bit fixed-point results. ALU instructions include:

- Fixed-point addition and subtraction of registers
- Addition and subtraction of immediate values
- Accumulation and subtraction of multiplier results
- Logical AND, OR, NOT, XOR, bitwise XOR, negate
- Functions: ABS, MAX, MIN, round, division primitives

ALU Operations

Primary ALU operations occur on ALU0, while parallel operations occur on ALU1, which performs a subset of ALU0 operations.

Table 2-8 describes the possible inputs and outputs of each ALU.

Table 2-8. Inputs and Outputs of Each ALU

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two or four 16-bit operands</td>
<td>One or two 16-bit results</td>
</tr>
<tr>
<td>Two 32-bit operands</td>
<td>One 32-bit result</td>
</tr>
<tr>
<td>32-bit result from the multiplier</td>
<td>Combination of 32-bit result from the multiplier with a 40-bit accumulation result</td>
</tr>
</tbody>
</table>

Combining operations in both ALUs can result in four 16-bit results, two 32-bit results, or two 40-bit results generated in a single instruction.
Arithmetic Logic Unit (ALU)

**Single 16-Bit Operations**

In single 16-bit operations, any two 16-bit register halves may be used as the input to the ALU. An addition, subtraction, or logical operation produces a 16-bit result that is deposited into an arbitrary destination register half. ALU0 is used for this operation, because it is the primary resource for ALU operations.

For example:

\[ R3.H = R1.H + R2.L \text{ (NS)}; \] adds the 16-bit contents of \( R1.H \) (\( R1 \) high half) to the contents of \( R2.L \) (\( R2 \) low half) and deposits the result in \( R3.H \) (\( R3 \) high half) with no saturation.

**Dual 16-Bit Operations**

In dual 16-bit operations, any two 32-bit registers may be used as the input to the ALU, considered as pairs of 16-bit operands. An addition, subtraction, or logical operation produces two 16-bit results that are deposited into an arbitrary 32-bit destination register. ALU0 is used for this operation, because it is the primary resource for ALU operations.

For example:

\[ R3 = R1 +|– R2 \text{ (S)}; \] adds the 16-bit contents of \( R2.H \) (\( R2 \) high half) to the contents of \( R1.H \) (\( R1 \) high half) and deposits the result in \( R3.H \) (\( R3 \) high half) with saturation.

The instruction also subtracts the 16-bit contents of \( R2.L \) (\( R2 \) low half) from the contents of \( R1.L \) (\( R1 \) low half) and deposits the result in \( R3.L \) (\( R3 \) low half) with saturation (see Figure 2-10).
Quad 16-Bit Operations

In quad 16-bit operations, any two 32-bit registers may be used as the inputs to ALU0 and ALU1, considered as pairs of 16-bit operands. A small number of addition or subtraction operations produces four 16-bit results that are deposited into two arbitrary, 32-bit destination registers. Both ALU0 and ALU1 are used for this operation. Because there are only two 32-bit data paths from the data register file to the arithmetic units, the same two pairs of 16-bit inputs are presented to ALU1 as to ALU0. The instruction construct is identical to that of a dual 16-bit operation, and input operands must be the same for both ALUs.

For example:

\[ R3 = R0 + R1, R2 = R0 - R1 (S); \]

performs four operations:

- Adds the 16-bit contents of \( R1.H \) (\( R1 \) high half) to the 16-bit contents of \( R0.H \) (\( R0 \) high half) and deposits the result in \( R3.H \) with saturation.
- Adds \( R1.L \) to \( R0.L \) and deposits the result in \( R3.L \) with saturation.
- Subtracts the 16-bit contents of \( R1.H \) (\( R1 \) high half) from the 16-bit contents of the \( R0.H \) (\( R0 \) high half) and deposits the result in \( R2.H \) with saturation.
- Subtracts \( R1.L \) from \( R0.L \) and deposits the result in \( R2.L \) with saturation.

 Explicitly, the four equivalent instructions are:

\[
\begin{align*}
R3.H &= R0.H + R1.H (S); \\
R3.L &= R0.L + R1.L (S); \\
R2.H &= R0.H - R1.H (S); \\
R2.L &= R0.L - R1.L (S); 
\end{align*}
\]
Arithmetic Logic Unit (ALU)

Single 32-Bit Operations

In single 32-bit operations, any two 32-bit registers may be used as the input to the ALU, considered as 32-bit operands. An addition, subtraction, or logical operation produces a 32-bit result that is deposited into an arbitrary 32-bit destination register. ALU0 is used for this operation, because it is the primary resource for ALU operations.

In addition to the 32-bit input operands coming from the data register file, operands may be sourced and deposited into the pointer register file, consisting of the eight registers $P[5:0], SP, FP$.

Instructions may not intermingle pointer registers with data registers.

For example:

R3 = R1 + R2 (NS); adds the 32-bit contents of R2 to the 32-bit contents of R1 and deposits the result in R3 with no saturation.

R3 = R1 + R2 (S); adds the 32-bit contents of R1 to the 32-bit contents of R2 and deposits the result in R3 with saturation.

Dual 32-Bit Operations

In dual 32-bit operations, any two 32-bit registers may be used as the input to ALU0 and ALU1, considered as a pair of 32-bit operands. An addition or subtraction produces two 32-bit results that are deposited into two 32-bit destination registers. Both ALU0 and ALU1 are used for this operation. Because only two 32-bit data paths go from the data register file to the arithmetic units, the same two 32-bit input registers are presented to ALU0 and ALU1.

For example:

R3 = R1 + R2, R4 = R1 - R2 (NS); adds the 32-bit contents of R2 to the 32-bit contents of R1 and deposits the result in R3 with no saturation.
The instruction also subtracts the 32-bit contents of \( R2 \) from that of \( R1 \) and deposits the result in \( R4 \) with no saturation.

A specialized form of this instruction uses the ALU 40-bit result registers as input operands, creating the sum and differences of the \( A0 \) and \( A1 \) registers.

For example:

\[
R3 = A0 + A1, \quad R4 = A0 - A1 \ (S); \quad \text{transfers to the result registers two 32-bit, saturated, sum and difference values of the ALU registers.}
\]

**ALU Instruction Summary**

Table 2-9 lists the ALU instructions. For more information about assembly language syntax and the effect of ALU instructions on the status flags, see *Blackfin Processor Programming Reference*.

In Table 2-9, note the meaning of these symbols:

- Dreg denotes any data register file register.
- Preg denotes any pointer register, \( FP \), or \( SP \) register.
- Dreg_lo_hi denotes any 16-bit register half in any data register file register.
- Dreg_lo denotes the lower 16 bits of any data register file register.
- imm7 denotes a signed, 7-bit wide, immediate value.
- \( A_n \) denotes either ALU result register \( A0 \) or \( A1 \).
- DIVS denotes a divide sign primitive.
- DIVQ denotes a divide quotient primitive.
- MAX denotes the maximum, or most positive, value of the source registers.
Arithmetic Logic Unit (ALU)

- MIN denotes the minimum value of the source registers.
- ABS denotes the absolute value of the upper and lower halves of a single 32-bit register.
- RND denotes rounding a half word.
- RND12 denotes saturating the result of an addition or subtraction and rounding the result on bit 12.
- RND20 denotes saturating the result of an addition or subtraction and rounding the result on bit 20.
- SIGNBITS denotes the number of sign bits in a number, minus one.
- EXPADJ denotes the lesser of the number of sign bits in a number minus one, and a threshold value.
- * Indicates the flag may be set or cleared, depending on the results of the instruction.
- ** Indicates the flag is cleared.
- – Indicates no effect.
- $d$ indicates $A_0$ contains the dividend MSB exclusive-OR divisor MSB.
### Table 2-9. ALU Instruction Summary

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ASTAT Status Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AZ</td>
</tr>
<tr>
<td>Preg = Preg + Preg ;</td>
<td></td>
</tr>
<tr>
<td>Preg += Preg ;</td>
<td></td>
</tr>
<tr>
<td>Preg -= Preg ;</td>
<td></td>
</tr>
<tr>
<td>Dreg = Dreg + Dreg ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = Dreg – Dreg (S) ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = Dreg + Dreg, Dreg = Dreg – Dreg ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_lo_hi = Dreg_lo_hi + Dreg_lo_hi ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_lo_hi = Dreg_lo_hi – Dreg_lo_hi (S) ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = Dreg ++ Dreg ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = Dreg ++ Dreg, Dreg = Dreg – Dreg ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = Dreg –+ Dreg + Dreg ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = Dreg – Dreg – Dreg ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = An + An, Dreg = An – An ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg += imm7 ;</td>
<td>*</td>
</tr>
<tr>
<td>Preg += imm7 ;</td>
<td></td>
</tr>
<tr>
<td>Dreg = ( A0 += A1 ) ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_lo_hi = ( A0 += A1) ;</td>
<td>*</td>
</tr>
<tr>
<td>A0 += A1 ;</td>
<td>*</td>
</tr>
</tbody>
</table>
### Arithmetic Logic Unit (ALU)

#### Table 2-9. ALU Instruction Summary (Cont’d)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ASTAT Status Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AZ</td>
</tr>
<tr>
<td>A0 –= A1 ;</td>
<td>*</td>
</tr>
<tr>
<td>DIVS ( Dreg, Dreg ) ;</td>
<td>*</td>
</tr>
<tr>
<td>DIVQ ( Dreg, Dreg ) ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = MAX ( Dreg, Dreg ) (V) ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = MIN ( Dreg, Dreg ) (V) ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = ABS Dreg (V) ;</td>
<td>*</td>
</tr>
<tr>
<td>An = ABS An ;</td>
<td>*</td>
</tr>
<tr>
<td>An = ABS An, An = ABS An ;</td>
<td>*</td>
</tr>
<tr>
<td>An = –An ;</td>
<td>*</td>
</tr>
<tr>
<td>An = –An, An =– An ;</td>
<td>*</td>
</tr>
<tr>
<td>An = An (S) ;</td>
<td>*</td>
</tr>
<tr>
<td>An = An (S), An = An (S) ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_lo_hi = Dreg (RND) ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_lo_hi = Dreg + Dreg (RND12) ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_lo_hi = Dreg – Dreg (RND12) ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_lo = SIGNBITS Dreg ;</td>
<td>–</td>
</tr>
</tbody>
</table>

![Image of the table](image-url)
Figure 2-9 shows a more detailed diagram of the arithmetic units and the data register file, which appears in Figure 2-1.

ALU0 is described here for convenience. ALU1 is very similar—a subset of ALU0.

Each ALU performs 40-bit addition for the accumulation of the multiplier results, as well as 32-bit and dual 16-bit operations. Each ALU has two 32-bit input ports that can be considered a pair of 16-bit operands or a single 32-bit operand. For single 16-bit operations, any of the four possible 16-bit operands may be used with any of the other 16-bit operands presented at the input to the ALU.
As shown in Figure 2-10, for dual 16-bit operations, the high halves and low halves are paired, providing four possible combinations of addition and subtraction.

(A) H + H, L + L    (B) H + H, L – L
(C) H – H, L + L    (D) H – H, L – L
Dual 16-Bit Cross Options

For dual 16-bit operations, the results may be crossed. “Crossing the results” changes the location in the result register for the result of a calculation. Usually, the result from the high side calculation is placed in the high half of the result register, and the result from the low side calculation is placed in the low half of the result register. With the cross option, the high result is placed in the low half of the destination register, and the low result is placed in the high half of the destination register (see Figure 2-11). This is particularly useful when dealing with complex math and portions of the Fast Fourier Transform (FFT). The cross option applies to ALU0 only.
Arithmetic Logic Unit (ALU)

Each ALU generates six status signals: the zero (AZ) status, the negative (AN) status, the carry (ACn) status, the sticky overflow (AVnS) status, the immediate overflow (AVn) status, and the quotient (AQ) status. All arithmetic status signals are latched into the arithmetic status register (ASTAT) at the end of the cycle. For the effect of ALU instructions on the status flags, see Table 2-9.

Depending on the instruction, the inputs can come from the data register file, the pointer register file, or the arithmetic result registers. Arithmetic on 32-bit operands directly support multiprecision operations in the ALU.

ALU Division Support Features

The ALU supports division with two special divide primitives. These instructions (DIVS, DIVQ) let programs implement a non-restoring, conditional (error checking), addition/subtraction/division algorithm.

The division can be either signed or unsigned, but both the dividend and divisor must be of the same type. Details about using division and programming examples are available in Blackfin Processor Programming Reference.
Special SIMD Video ALU Operations

Four 8-bit video ALUs enable the processor to process video information with high efficiency. Each video ALU instruction may take from one to four pairs of 8-bit inputs and return one to four 8-bit results. The inputs are presented to the video ALUs in two 32-bit words from the data register file. The possible operations include:

- Quad 8-bit add or subtract
- Quad 8-bit average
- Quad 8-bit pack or unpack
- Quad 8-bit subtract-absolute-accumulate
- Byte align

For more information about the operation of these instructions, see Blackfin Processor Programming Reference.

Multiply Accumulators (Multipliers)

The two multipliers (MAC0 and MAC1) perform fixed-point multiplication and multiply and accumulate operations. Multiply and accumulate operations are available with either cumulative addition or cumulative subtraction.

Multiplier fixed-point instructions operate on 16-bit fixed-point data and produce 32-bit results that may be added or subtracted from a 40-bit accumulator.
Inputs are treated as fractional or integer, unsigned or two’s-complement. Multiplier instructions include:

- Multiplication
- Multiply and accumulate with addition, rounding optional
- Multiply and accumulate with subtraction, rounding optional
- Dual versions of the above

**Multiplier Operation**

Each multiplier has two 32-bit inputs from which it derives the two 16-bit operands. For single multiply and accumulate instructions, these operands can be any data registers in the data register file. Each multiplier can accumulate results in its accumulator register, $A_1$ or $A_0$. The accumulator results can be saturated to 32 or 40 bits. The multiplier result can also be written directly to a 16- or 32-bit destination register with optional rounding.

Each multiplier instruction determines whether the inputs are either both in integer format or both in fractional format. The format of the result matches the format of the inputs. In MAC0, both inputs are treated as signed or unsigned. In MAC1, there is a mixed-mode option.

If both inputs are fractional and signed, the multiplier automatically shifts the result left one bit to remove the redundant sign bit. Unsigned fractional, integer, and mixed modes do not perform a shift for sign bit correction. Multiplier instruction options specify the data format of the inputs. See “Multiplier Instruction Options” on page 2-42 for more information.
Placing Multiplier Results in Multiplier Accumulator Registers

As shown in Figure 2-9, each multiplier has a dedicated accumulator, $A_0$ or $A_1$. Each accumulator register is divided into three sections—$A_0.L/A_1.L$ (bits 15:0), $A_0.H/A_1.H$ (bits 31:16), and $A_0.X/A_1.X$ (bits 39:32).

When the multiplier writes to its result accumulator registers, the 32-bit result is deposited into the lower bits of the combined accumulator register, and the MSB is sign-extended into the upper eight bits of the register ($A_0.X/A_1.X$).

Multiplier output can be deposited not only in the $A_0$ or $A_1$ registers, but also in a variety of 16- or 32-bit data registers in the data register file.

Rounding or Saturating Multiplier Results

On a multiply and accumulate operation, the accumulator data can be saturated and, optionally, rounded for extraction to a register or register half. When a multiply deposits a result only in a register or register half, the saturation and rounding works the same way. The rounding and saturation operations work as follows.

- Rounding is applied only to fractional results except for the $IH$ option, which applies rounding and high half extraction to an integer result.

  For the $IH$ option, the rounded result is obtained by adding $0x8000$ to the accumulator (for MAC) or multiply result (for mult) and then saturating to 32-bits. For more information, see “Rounding Multiplier Results” on page 2-18.

- If an overflow or underflow has occurred, the saturate operation sets the specified result register to the maximum positive or negative value. For more information, see the following section.
Multiply Accumulators (Multipliers)

Saturating Multiplier Results on Overflow

The following bits in ASTAT indicate multiplier overflow status:

- Bit 16 (AV0) and bit 18 (AV1) record overflow condition (whether the result has overflowed 32 bits) for the A0 and A1 accumulators, respectively.

  If the bit is cleared (=0), no overflow or underflow has occurred. If the bit is set (=1), an overflow or underflow has occurred. The AV0S and AV1S bits are sticky bits.

- Bit 24 (V) and bit 25 (VS) are set if overflow occurs in extracting the accumulator result to a register.

Multiplier Instruction Summary

Table 2-10 lists the multiplier instructions. For more information about assembly language syntax and the effect of multiplier instructions on the status flags, see Blackfin Processor Programming Reference.

In Table 2-10, note the meaning of these symbols:

- Dreg denotes any data register file register.
- Dreg_lo_hi denotes any 16-bit register half in any data register file register.
- Dreg_lo denotes the lower 16 bits of any data register file register.
- Dreg_hi denotes the upper 16 bits of any data register file register.
- \( A_n \) denotes either MAC accumulator register A0 or A1.
- * Indicates the flag may be set or cleared, depending on the results of the instruction.
- – Indicates no effect.
Multiplier instruction options are described in “Multiplier Instruction Options” on page 2-42.

Table 2-10. Multiplier Instruction Summary

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ASTAT Status Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AV0</td>
</tr>
<tr>
<td>Dreg_lo = Dreg_lo_hi * Dreg_lo_hi ;</td>
<td>–</td>
</tr>
<tr>
<td>Dreg_hi = Dreg_lo_hi * Dreg_lo_hi ;</td>
<td>–</td>
</tr>
<tr>
<td>Dreg = Dreg_lo_hi * Dreg_lo_hi ;</td>
<td>–</td>
</tr>
<tr>
<td>An = Dreg_lo_hi * Dreg_lo_hi ;</td>
<td>*</td>
</tr>
<tr>
<td>An += Dreg_lo_hi * Dreg_lo_hi ;</td>
<td>*</td>
</tr>
<tr>
<td>An -= Dreg_lo_hi * Dreg_lo_hi ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_lo = ( A0 = Dreg_lo_hi * Dreg_lo_hi ) ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_lo = ( A0 += Dreg_lo_hi * Dreg_lo_hi ) ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_lo = ( A0 -= Dreg_lo_hi * Dreg_lo_hi ) ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_hi = ( A1 = Dreg_lo_hi * Dreg_lo_hi ) ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_hi = ( A1 += Dreg_lo_hi * Dreg_lo_hi ) ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_hi = ( A1 -= Dreg_lo_hi * Dreg_lo_hi ) ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = ( An = Dreg_lo_hi * Dreg_lo_hi ) ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = ( An += Dreg_lo_hi * Dreg_lo_hi ) ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = ( An -= Dreg_lo_hi * Dreg_lo_hi ) ;</td>
<td>*</td>
</tr>
</tbody>
</table>
Multiplier Instruction Options

The following descriptions of multiplier instruction options provide an overview. Not all options are available for all instructions. For information about how to use these options with their respective instructions, see Blackfin Processor Programming Reference.

**default**
No option; input data is signed fraction.

(IS)
Input data operands are signed integer. No shift correction is made.

(FU)
Input data operands are unsigned fraction. No shift correction is made.

(IU)
Input data operands are unsigned integer. No shift correction is made.

(T)
Input data operands are signed fraction. When copying to the destination half register, truncates the lower 16 bits of the accumulator contents.

(TFU)
Input data operands are unsigned fraction. When copying to the destination half register, truncates the lower 16 bits of the accumulator contents.

(ISS2)
If multiplying and accumulating to a register:

Input data operands are signed integer. When copying to the destination register, accumulator contents are scaled (multiplied x2 by a one-place shift-left). If scaling produces a signed value larger than 32 bits, the number is saturated to its maximum positive or negative value.
If multiplying and accumulating to a half register:

When copying the lower 16 bits to the destination half register, the accumulator contents are scaled. If scaling produces a signed value greater than 16 bits, the number is saturated to its maximum positive or negative value.

(IH)

This option indicates integer multiplication with high half word extraction. The accumulator is saturated at 32 bits, and bits [31:16] of the accumulator are rounded, and then copied into the destination half register.

(W32)

Input data operands are signed fraction with no extension bits in the accumulators at 32 bits. Left-shift correction of the product is performed, as required. This option is used for legacy GSM speech vocoder algorithms written for 32-bit accumulators. For this option only, this special case applies: \(0x8000 \times 0x8000 = 0x7FFF\).

(M)

Operation uses mixed-multiply mode. Valid only for MAC1 versions of the instruction. Multiplies a signed fraction by an unsigned fractional operand with no left-shift correction. Operand one is signed; operand two is unsigned. MAC0 performs an unmixed multiply on signed fractions by default, or another format as specified. That is, MAC0 executes the specified signed/signed or unsigned/unsigned multiplication. The (M) option can be used alone or in conjunction with one other format option.
Multiplier Data Flow Details

Figure 2-12 shows the register files and ALUs, along with the multiplier/accumulators.

Figure 2-12. Register Files and ALUs

Each multiplier has two 16-bit inputs, performs a 16-bit multiplication, and stores the result in a 40-bit accumulator or extracts to a 16-bit or 32-bit register. Two 32-bit words are available at the MAC inputs, providing four 16-bit operands to choose from.
One of the operands must be selected from the low half or the high half of one 32-bit word. The other operand must be selected from the low half or the high half of the other 32-bit word. Thus, each MAC is presented with four possible input operand combinations. The two 32-bit words can contain the same register information, giving the options for squaring and multiplying the high half and low half of the same register. Figure 2-13 show these possible combinations.

Figure 2-13. Four Possible Combinations of MAC Operations

The 32-bit product is passed to a 40-bit adder/subtractor, which may add or subtract the new product from the contents of the accumulator result register or pass the new product directly to the data register file results register. For results, the A0 and A1 registers are 40 bits wide. Each of these registers consists of smaller 32- and 8-bit registers—A0.W, A1.W, A0.X, and A1.X.
Multiply Accumulators (Multipliers)

Some example instructions:

A0 = R3.L * R4.H ;

In this instruction, the MAC0 multiplier/accumulator performs a multiply and puts the result in the accumulator register.


In this instruction, the MAC1 multiplier/accumulator performs a multiply and accumulates the result with the previous results in the A1 accumulator.

Multiply Without Accumulate

The multiplier may operate without the accumulation function. If accumulation is not used, the result can be directly stored in a register from the data register file or the accumulator register. The destination register may be 16 bits or 32 bits. If a 16-bit destination register is a low half, then MAC0 is used; if it is a high half, then MAC1 is used. For a 32-bit destination register, either MAC0 or MAC1 is used.

If the destination register is 16 bits, then the word that is extracted from the multiplier depends on the data type of the input.

- If the multiplication uses fractional operands or the IH option, then the high half of the result is extracted and stored in the 16-bit destination registers (see Figure 2-14).

- If the multiplication uses integer operands, then the low half of the result is extracted and stored in the 16-bit destination registers. These extractions provide the most useful information in the resultant 16-bit word for the data type chosen (see Figure 2-15).
Computational Units

For example, this instruction uses fractional, unsigned operands:

\[ R0.L = R1.L \times R2.L \text{ (FU)}; \]

The instruction deposits the upper 16 bits of the multiply answer with rounding and saturation into the lower half of \( R0 \), using MAC0. This instruction uses unsigned integer operands:

\[ R0.H = R2.H \times R3.H \text{ (IU)}; \]

The instruction deposits the lower 16 bits of the multiply answer with any required saturation into the high half of \( R0 \), using MAC1:

\[ R0 = R1.L \times R2.L; \]

Regardless of operand type, the preceding operation deposits 32 bits of the multiplier answer with saturation into \( R0 \), using MAC0.
### Special 32-Bit Integer MAC Instruction

The processor supports a multi cycle 32-bit MAC instruction:

\[
\text{Dreg } *= \text{Dreg}
\]

The single instruction multiplies two 32-bit integer operands and provides a 32-bit integer result, destroying one of the input operands.

The instruction takes multiple cycles to execute. Refer to the product data sheet and *Blackfin Processor Programming Reference* for more information about the exact operation of this instruction. This macro function is interruptable and does not modify the data in either accumulator register \(A0\) or \(A1\).
Dual MAC Operations

The processor has two 16-bit MACs. Both MACs can be used in the same operation to double the MAC throughput. The same two 32-bit input registers are offered to each MAC unit, providing each with four possible combinations of 16-bit input operands. Dual MAC operations are frequently referred to as vector operations, because a program could store vectors of samples in the four input operands and perform vector computations.

An example of a dual multiply and accumulate instruction is:

\[
\text{A1} += \text{R1.H} \times \text{R2.L}, \ \text{A0} += \text{R1.L} \times \text{R2.H};
\]

This instruction represents two multiply and accumulate operations.

- In one operation (MAC1) the high half of \( \text{R1} \) is multiplied by the low half of \( \text{R2} \) and added to the contents of the \( \text{A1} \) accumulator.

- In the second operation (MAC0) the low half of \( \text{R1} \) is multiplied by the high half of \( \text{R2} \) and added to the contents of \( \text{A0} \).

The results of the MAC operations may be written to registers in a number of ways: as a pair of 16-bit halves, as a pair of 32-bit registers, or as an independent 16-bit half register or 32-bit register.

For example:

\[
\text{R3.H} = (\text{A1} += \text{R1.H} \times \text{R2.L}), \ \text{R3.L} = (\text{A0} += \text{R1.L} \times \text{R2.L});
\]

In this instruction, the 40-bit accumulator is packed into a 16-bit half register. The result from MAC1 must be transferred to a high half of a destination register and the result from MAC0 must be transferred to the low half of the same destination register.

The operand type determines the correct bits to extract from the accumulator and deposit in the 16-bit destination register. See “Multiply Without Accumulate” on page 2-46.
Barrel Shifter (Shifter)

R3 = (A1 += R1.H * R2.L), R2 = (A0 += R1.L * R2.L);

In this instruction, the 40-bit accumulators are packed into two 32-bit registers. The registers must be register pairs (R[1:0], R[3:2], R[5:4], R[7:6]).


This instruction is an example of one accumulator—but not the other—being transferred to a register. Either a 16- or 32-bit register may be specified as the destination register.

Barrel Shifter (Shifter)

The shifter provides bitwise shifting functions for 16-, 32-, or 40-bit inputs, yielding a 16-, 32-, or 40-bit output. These functions include arithmetic shift, logical shift, rotate, and various bit test, set, pack, unpack, and exponent detection functions. These shift functions can be combined to implement numerical format control, including full floating-point representation.

Shifter Operations

The shifter instructions (>>>, >>, <<, ASHIFT, LSHIFT, ROT) can be used various ways, depending on the underlying arithmetic requirements. The ASHIFT and >>> instructions represent the arithmetic shift. The LSHIFT, <<, and >> instructions represent the logical shift.

The arithmetic shift and logical shift operations can be further broken into subsections. Instructions that are intended to operate on 16-bit single or paired numeric values (as would occur in many DSP algorithms) can use the instructions ASHIFT and LSHIFT. These are typically three-operand instructions.
Instructions that are intended to operate on a 32-bit register value and use two operands, such as instructions frequently used by a compiler, can use the >>> and >> instructions.

Arithmetic shift, logical shift, and rotate instructions can obtain the shift argument from a register or directly from an immediate value in the instruction. For details about shifter related instructions, see “Shifter Instruction Summary” on page 2-54.

Two-Operand Shifts

Two-operand shift instructions shift an input register and deposit the result in the same register.

Immediate Shifts

An immediate shift instruction shifts the input bit pattern to the right (downshift) or left (upshift) by a given number of bits. Immediate shift instructions use the data value in the instruction itself to control the amount and direction of the shifting operation.

The following example shows the input value downshifted.

R0 contains 0000 B6A3;
R0 >>= 0x04;

results in
R0 contains 0000 0B6A;

The following example shows the input value upshifted.

R0 contains 0000 B6A3;
R0 <<= 0x04;

results in
R0 contains 000B 6A30;
Barrel Shifter (Shifter)

Register Shifts

Register-based shifts use a register to hold the shift value. The entire 32-bit register is used to derive the shift value, and when the magnitude of the shift is greater than or equal to 32, then the result is either 0 or –1.

The following example shows the input value upshifted.

R0 contains 0000 B6A3;
R2 contains 0000 0004;
R0 <<= R2;

results in
R0 contains 000B 6A30;

Three-Operand Shifts

Three-operand shifter instructions shift an input register and deposit the result in a destination register.

Immediate Shifts

Immediate shift instructions use the data value in the instruction itself to control the amount and direction of the shifting operation.

The following example shows the input value downshifted.

R0 contains 0000 B6A3;
R1 = R0 >> 0x04;

results in
R1 contains 0000 0B6A;

The following example shows the input value upshifted.

R0.L contains B6A3;
R1.H = R0.L << 0x04;
results in  
R1.H contains 6A30;

Register Shifts

Register-based shifts use a register to hold the shift value. When a register is used to hold the shift value (for ASHIFT, LSHIFT or ROT), then the shift value is always found in the low half of a register (Rn.L). The bottom six bits of Rn.L are masked off and used as the shift value.

The following example shows the input value upshifted.

RO contains 0000 B6A3;  
R2.L contains 0004;  
R1 = RO ASHIFT by R2.L;  
results in  
R1 contains 000B 6A30;

The following example shows the input value rotated. Assume the condition code (CC) bit is set to 0. For more information about CC, see “Condition Code Flag” on page 4-12.

RO contains ABCD EF12;  
R2.L contains 0004;  
R1 = RO ROT by R2.L;  
results in  
R1 contains BCDE F125;  
Note the CC bit is included in the result, at bit 3.

Bit Test, Set, Clear, Toggle

The shifter provides the method to test, set, clear, and toggle specific bits of a data register. All instructions have two arguments—the source register and the bit field value. The test instruction does not change the source register. The result of the test instruction resides in the CC bit.
Barrel Shifter (Shifter)

The following examples show a variety of operations.

```c
BITCLR ( R0, 6 );
BITSET ( R2, 9 );
BTTGL ( R3, 2 );
CC = BTTST ( R3, 0 );
```

Field Extract and Field Deposit

If the shifter is used, a source field may be deposited anywhere in a 32-bit destination field. The source field may be from 1 bit to 16 bits in length. In addition, a 1- to 16-bit field may be extracted from anywhere within a 32-bit source field.

Two register arguments are used for these functions. One holds the 32-bit destination or 32-bit source. The other holds the extract/deposit value, its length, and its position within the source.

Shifter Instruction Summary

Table 2-11 lists the shifter instructions. For more information about assembly language syntax and the effect of shifter instructions on the status flags, see Blackfin Processor Programming Reference.

In Table 2-11, note the meaning of these symbols:

- Dreg denotes any data register file register.
- Dreg_lo denotes the lower 16 bits of any data register file register.
- Dreg_hi denotes the upper 16 bits of any data register file register.
- * Indicates the flag may be set or cleared, depending on the results of the instruction.
- * 0 Indicates versions of the instruction that send results to accumulator AO set or clear AV0.
- * 1 Indicates versions of the instruction that send results to accumulator A1 set or clear AV1.
- ** Indicates the flag is cleared.
- *** Indicates CC contains the latest value shifted into it.
- – Indicates no effect.

Table 2-11. Shifter Instruction Summary

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ASTAT Status Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AZ</td>
</tr>
<tr>
<td>BITCLR (Dreg, uimm5) ;</td>
<td>*</td>
</tr>
<tr>
<td>BITSET (Dreg, uimm5) ;</td>
<td>**</td>
</tr>
<tr>
<td>BITTGL (Dreg, uimm5) ;</td>
<td>*</td>
</tr>
<tr>
<td>CC = BITTST (Dreg, uimm5) ;</td>
<td>–</td>
</tr>
<tr>
<td>CC = !BITTST (Dreg, uimm5) ;</td>
<td>–</td>
</tr>
<tr>
<td>Dreg = DEPOSIT (Dreg, Dreg) ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = EXTRACT (Dreg) ;</td>
<td>*</td>
</tr>
<tr>
<td>BITMUX (Dreg, Dreg, A0) ;</td>
<td>–</td>
</tr>
<tr>
<td>Dreg_lo = ONES Dreg ;</td>
<td>–</td>
</tr>
<tr>
<td>Dreg = PACK (Dreg_lo_hi, Dreg_lo_hi);</td>
<td>–</td>
</tr>
<tr>
<td>Dreg &gt;&gt;&gt;= uimm5 ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg &gt;&gt;= uimm5 ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg &lt;&lt;= uimm5 ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = Dreg &gt;&gt;&gt; uimm5 ;</td>
<td>*</td>
</tr>
</tbody>
</table>
### Barrel Shifter (Shifter)

#### Table 2-11. Shifter Instruction Summary (Cont’d)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ASTAT Status Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AZ</td>
</tr>
<tr>
<td>Dreg = Dreg &gt;&gt; uimm5 ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = Dreg &lt;&lt; uimm5 ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = Dreg &gt;&gt;&gt; uimm4 (V) ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = Dreg &gt;&gt; uimm4 (V) ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = Dreg &lt;&lt; uimm4 (V) ;</td>
<td>*</td>
</tr>
<tr>
<td>An = An &gt;&gt;&gt; uimm5 ;</td>
<td>*</td>
</tr>
<tr>
<td>An = An &gt;&gt; uimm5 ;</td>
<td>*</td>
</tr>
<tr>
<td>An = An &lt;&lt; uimm5 ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_lo_hi = Dreg_lo_hi &gt;&gt;&gt; uimm4 ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_lo_hi = Dreg_lo_hi &gt;&gt; uimm4 ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg_lo_hi = Dreg_lo_hi &lt;&lt; uimm4 ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg &gt;&gt;&gt;= Dreg ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg &gt;&gt;= Dreg ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg &lt;&lt;= Dreg ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = ASHIFT Dreg BY Dreg_lo ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = LSHIFT Dreg BY Dreg_lo ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = ROT Dreg BY imm6 ;</td>
<td>–</td>
</tr>
<tr>
<td>Dreg = ASHIFT Dreg BY Dreg_lo (V) ;</td>
<td>*</td>
</tr>
<tr>
<td>Dreg = LSHIFT Dreg BY Dreg_lo (V) ;</td>
<td>*</td>
</tr>
</tbody>
</table>
Table 2-11. Shifter Instruction Summary (Cont’d)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ASTAT Status Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AZ</td>
</tr>
<tr>
<td>Dreg_lo_hi = ASHIFT Dreg_lo_hi BY Dreg_lo ;</td>
<td>*</td>
</tr>
</tbody>
</table>
| Dreg_lo_hi = LSHIFT Dreg_lo_hi BY Dreg_lo ; | *    | *   | -   | -        | -   | -   | -    | -   | -    | **|/|-
| An = An ASHIFT BY Dreg_lo ;   | *    | *   | -   | 0       | 1   | -   | -    | -   | -    | -  | -  | -       |
| An = An ROT BY imm6 ;         | -    | -   | -   | -        | -   | -   | -    | -   | -    | -  | -  | -       |
| Preg = Preg >> 1 ;           | -    | -   | -   | -        | -   | -   | -    | -   | -    | -  | -  | -       |
| Preg = Preg >> 2 ;           | -    | -   | -   | -        | -   | -   | -    | -   | -    | -  | -  | -       |
| Preg = Preg << 1 ;           | -    | -   | -   | -        | -   | -   | -    | -   | -    | -  | -  | -       |
| Preg = Preg << 2 ;           | -    | -   | -   | -        | -   | -   | -    | -   | -    | -  | -  | -       |
| Dreg = ( Dreg + Dreg ) << 1 ; | *    | *   | *   | -        | -   | -   | -    | -   | -    | -  | -  | -       |
| Dreg = ( Dreg + Dreg ) << 2 ; | *    | *   | *   | -        | -   | -   | -    | -   | -    | -  | -  | -       |
| Preg = ( Preg + Preg ) << 1 ; | -    | -   | -   | -        | -   | -   | -    | -   | -    | -  | -  | -       |
| Preg = ( Preg + Preg ) << 2 ; | -    | -   | -   | -        | -   | -   | -    | -   | -    | -  | -  | -       |
| Preg = Preg + ( Preg << 1 ) ; | -    | -   | -   | -        | -   | -   | -    | -   | -    | -  | -  | -       |
| Preg = Preg + ( Preg << 2 ) ; | -    | -   | -   | -        | -   | -   | -    | -   | -    | -  | -  | -       |
Barrel Shifter (Shifter)
3 OPERATING MODES AND STATES

The processor supports the following three processor modes:

- User mode
- Supervisor mode
- Emulation mode

Emulation and supervisor modes have unrestricted access to the core resources. User mode has restricted access to certain system resources, thus providing a protected software environment.

User mode is considered the domain of application programs. Supervisor mode and emulation mode are usually reserved for the kernel code of an operating system.

The processor mode is determined by the event controller. When servicing an interrupt, a nonmaskable interrupt (NMI), or an exception, the processor is in supervisor mode. When servicing an emulation event, the processor is in emulation mode. When not servicing any events, the processor is in user mode.

The current processor mode may be identified by interrogating the \texttt{IPEND} memory-mapped register (MMR), as shown in Table 3-1.

\begin{itemize}
  \item MMRs cannot be read while the processor is in user mode.
\end{itemize}
Table 3-1. Identifying the Current Processor Mode

<table>
<thead>
<tr>
<th>Event</th>
<th>Mode</th>
<th>IPEND</th>
</tr>
</thead>
</table>
| Interrupt| Supervisor| $\geq 0x10$
but IPEND[0], IPEND[1], IPEND[2], and IPEND[3] = 0. |
| Exception| Supervisor| $\geq 0x08$
The core is processing an exception event if IPEND[0] = 0, IPEND[1] = 0, IPEND[2] = 0, IPEND[3] = 1, and IPEND[15:4] are 0’s or 1’s. |
| NMI      | Supervisor| $\geq 0x04$
The core is processing an NMI event if IPEND[0] = 0, IPEND[1] = 0, IPEND[2] = 1, and IPEND[15:2] are 0’s or 1’s. |
| Reset    | Supervisor| $= 0x02$
As the reset state is exited, IPEND is set to 0x02, and the reset vector runs in supervisor mode. |
| Emulation| Emulator  | $= 0x01$
The processor is in emulation mode if IPEND[0] = 1, regardless of the state of the remaining bits IPEND[15:1]. |
| None     | User      | $= 0x00$                                    |

In addition, the processor supports the following two non-processing states:

- Idle state
- Reset state

Figure 3-1 illustrates the processor modes and states as well as the transition conditions between them.
Operating Modes and States

User Mode

The processor is in user mode when it is not in reset or idle state, and when it is not servicing an interrupt, NMI, exception, or emulation event. User mode is used to process application level code that does not require explicit access to system registers. Any attempt to access restricted system registers causes an exception event. Table 3-2 lists the registers that may be accessed in user mode.

Figure 3-1. Processor Modes and States

(1) Normal exit from Reset is to Supervisor mode. However, emulation hardware may have initiated a reset. If so exit from Reset is to Emulation.
Table 3-2. Registers Accessible in User Mode

<table>
<thead>
<tr>
<th>Processor Registers</th>
<th>Register Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Registers</td>
<td>R[7:0], A[1:0]</td>
</tr>
<tr>
<td>Pointer Registers</td>
<td>P[5:0], SP, FP, I[3:0], M[3:0], L[3:0], B[3:0]</td>
</tr>
<tr>
<td>Sequencer and register Registers</td>
<td>RETS, LC[1:0], LT[1:0], LB[1:0], ASTAT, CYCLES, CYCLES2</td>
</tr>
</tbody>
</table>

**Protected Resources and Instructions**

System resources consist of a subset of processor registers, all MMRs, and a subset of protected instructions. These system and core MMRs are located starting at address 0xFFC0 0000. This region of memory is protected from user mode access. Any attempt to access MMR space in user mode causes an exception.

A list of protected instructions appears in Table 3-3. Any attempt to issue any of the protected instructions from user mode causes an exception event.

Table 3-3. Protected Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTI</td>
<td>Return from interrupt</td>
</tr>
<tr>
<td>RTX</td>
<td>Return from exception</td>
</tr>
<tr>
<td>RTN</td>
<td>Return from NMI</td>
</tr>
<tr>
<td>CLI</td>
<td>Disable interrupts</td>
</tr>
<tr>
<td>STI</td>
<td>Enable interrupts</td>
</tr>
<tr>
<td>RAISE</td>
<td>Force interrupt/reset</td>
</tr>
<tr>
<td>RTE</td>
<td>Return from emulation</td>
</tr>
<tr>
<td></td>
<td>Causes an exception only if executed outside emulation mode</td>
</tr>
</tbody>
</table>
**Protected Memory**

Additional memory locations can be protected from user mode access. A cacheability protection lookaside buffer (CPLB) entry can be created and enabled. See “Memory Management Unit” on page 6-43 for further information.

**Entering User Mode**

When coming out of reset, the processor is in supervisor mode because it is servicing a reset event. To enter user mode from the reset state, two steps must be performed. First, a return address must be loaded into the RETI register. Second, an RTI must be issued. The following example code shows how to enter user mode upon reset.

**Example Code to Enter User Mode Upon Reset**

Listing 3-1 provides code for entering user mode from reset.

Listing 3-1. Entering User Mode From Reset

```
P1.L = START ;  /* Point to start of user code */
P1.H = START ;
RETI = P1 ;
RTI ;  /* Return from Reset Event */

START :  /* Place user code here */
```
Return Instructions That Invoke User Mode

Table 3-4 provides a summary of return instructions that can be used to invoke user mode from various processor event service routines. When these instructions are used in service routines, the value of the return address must be first stored in the appropriate event $\text{RETx}$ register. In the case of an interrupt routine, if the service routine is interruptible, the return address is stored on the stack. For this case, the address can be found by popping the value from the stack into $\text{RETI}$. Once $\text{RETI}$ has been loaded, the $\text{RTI}$ instruction can be issued.

Note the stack pop is optional. If the $\text{RETI}$ register is not pushed/popped, then the interrupt service routine becomes non-interruptible, because the return address is not saved on the stack.

The processor remains in user mode until one of these events occurs:

- An interrupt, NMI, or exception event invokes supervisor mode.
- An emulation event invokes emulation mode.
- A reset event invokes the reset state.

Table 3-4. Return Instructions That Can Invoke User Mode

<table>
<thead>
<tr>
<th>Current Process Activity</th>
<th>Return Instruction to Use</th>
<th>Execution Resumes at Address in This Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt Service Routine</td>
<td>RTI</td>
<td>RETI</td>
</tr>
<tr>
<td>Exception Service Routine</td>
<td>RTX</td>
<td>RETX</td>
</tr>
<tr>
<td>Nonmaskable Interrupt Service Routine</td>
<td>RTN</td>
<td>RETN</td>
</tr>
<tr>
<td>Emulation Service Routine</td>
<td>RTE</td>
<td>RETE</td>
</tr>
</tbody>
</table>
Operating Modes and States

Supervisor Mode

The processor services all interrupt, NMI, and exception events in supervisor mode.

Supervisor mode has full, unrestricted access to all processor system resources, including all emulation resources, unless a CPLB has been configured and enabled. See “Memory Management Unit” on page 6-43 for a further description. Only supervisor mode can use the register alias USP, which references the user stack pointer in memory. This register alias is necessary because in supervisor mode, SP refers to the kernel stack pointer rather than to the user stack pointer.

Normal processing begins in supervisor mode from the reset state. Deasserting the RESET signal switches the processor from the reset state to supervisor mode where it remains until an emulation event or return instruction occurs to change the mode. Before the return instruction is issued, the RETI register must be loaded with a valid return address.

Non-OS Environments

For non-OS environments, application code should remain in supervisor mode so that it can access all core and system resources. When RESET is deasserted, the processor initiates operation by servicing the reset event. Emulation is the only event that can pre-empt this activity. Therefore, lower priority events cannot be processed.

One way of keeping the processor in supervisor mode and still allowing lower priority events to be processed is to set up and force the lowest priority interrupt (IVG15). Events and interrupts are described further in “Events and Sequencing” on page 4-18. After the low priority interrupt has been forced using the RAISE 15 instruction, RETI can be loaded with a return address that points to user code that can execute until IVG15 is issued. After RETI has been loaded, the RTI instruction can be issued to return from the reset event.
Supervisor Mode

The interrupt handler for IVG15 can be set to jump to the application code starting address. An additional RTI is not required. As a result, the processor remains in supervisor mode because IPEND[15] remains set. At this point, the processor is servicing the lowest priority interrupt. This ensures that higher priority interrupts can be processed.

Example Code for Supervisor Mode Coming Out of Reset

To remain in supervisor mode when coming out of the reset state, use code as shown in Listing 3-2.

Listing 3-2. Staying in Supervisor Mode Coming Out of Reset

```plaintext
PO.L = LO(EVT15) ; /* Point to IVG15 in Event Vector Table */
PO.H = HI(EVT15) ;
P1.L = START ;   /* Point to start of User code */

P1.H = START ;
[P0] = P1 ;   /* Place the address of start code in IVG15 of EVT */

PO.L = LO(IMASK) ;

RO = [P0] ;
R1.L = EVT_IVG15 & 0xFFFF ;

RO = RO | R1 ;
[P0] = RO ; /* Set (enable) IVG15 bit in Interrupt Mask register */
```
RAISE 15 ;  /* Invoke IVG15 interrupt */
PO.L = WAIT_HERE ;
PO.H = WAIT_HERE ;
RETI = PO ;  /* RETI loaded with return address */
RTI ;  /* Return from Reset Event */
WAIT_HERE :  /* Wait here till IVG15 interrupt is serviced */

JUMP WAIT_HERE ;

START:  /* IVG15 vectors here */
[--SP] = RETI ;  /* Enables interrupts and saves return address to stack */

Emulation Mode

The processor enters emulation mode if emulation mode is enabled and either of these conditions is met:

- An external emulation event occurs.
- The EMUEXCPT instruction is issued.

The processor remains in emulation mode until the emulation service routine executes an RTE instruction. If no interrupts are pending when the RTE instruction executes, the processor switches to user mode. Otherwise, the processor switches to supervisor mode to service the interrupt.

Emulation mode is the highest priority mode, and the processor has unrestricted access to all system resources.
Idle State

Idle state stops all processor activity at the user’s discretion, usually to conserve power during lulls in activity. No processing occurs during the Idle state. The Idle state is invoked by a sequential IDLE instruction. The IDLE instruction notifies the processor hardware that the Idle state is requested. The SSYNC instruction purges all speculative and transient states in the core and external system.

The processor remains in the idle state until a peripheral or external device, such as a SPORT or the real-time clock (RTC), generates an interrupt that requires servicing.

In Listing 3-3, core interrupts are disabled and the IDLE instruction is executed. When all the pending processes have completed, the core disables its clocks. Since interrupts are disabled, Idle state can be terminated only by asserting a WAKEUP signal. For more information, see “System Interrupt Wake-Up Enable (SIC_IWRx) Registers” on page 4-27. (While not required, an interrupt could also be enabled in conjunction with the WAKEUP signal.)

When the WAKEUP signal is asserted, the processor wakes up, and the STI instruction enables interrupts again.

Example Code for Transition to Idle State

To transition to the idle state, use code shown in Listing 3-3.

Listing 3-3. Transitioning to Idle State

```
CLI R0 ; /* disable interrupts */
IDLE ; /* drain pipeline and send core into IDLE state */
STI R0 ; /* re-enable interrupts after wakeup */
```
Reset State

Reset state initializes the processor logic. During reset state, application programs and the operating system do not execute. Clocks are stopped while in reset state.

The processor remains in the reset state as long as external logic asserts the external \texttt{RESET} signal. Upon deassertion, the processor completes the reset sequence and switches to supervisor mode, where it executes code found at the reset event vector.

Software in supervisor or emulation mode can invoke the reset state without involving the external \texttt{RESET} signal. This can be done by issuing the reset version of the \texttt{RAISE} instruction.

Application programs in user mode cannot invoke the reset state, except through a system call provided by an operating system kernel. Table 3-5 summarizes the state of the processor upon reset.
Table 3-5. Processor State Upon Reset

<table>
<thead>
<tr>
<th>Item</th>
<th>Description of Reset State</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Core</strong></td>
<td></td>
</tr>
<tr>
<td>Operating Mode</td>
<td>Supervisor mode in reset event, clocks stopped</td>
</tr>
<tr>
<td>Rounding Mode</td>
<td>Unbiased rounding</td>
</tr>
<tr>
<td>Cycle Counters</td>
<td>Disabled, zero</td>
</tr>
<tr>
<td>DAG Registers (I, L, B, M)</td>
<td>Random values (must be cleared at initialization)</td>
</tr>
<tr>
<td>Data and Address Registers</td>
<td>Random values (must be cleared at initialization)</td>
</tr>
<tr>
<td>IPEND, IMASK, ILAT</td>
<td>Cleared, interrupts globally disabled with IPEND bit 4</td>
</tr>
<tr>
<td>CPLBs</td>
<td>Disabled</td>
</tr>
<tr>
<td>L1 Instruction memory</td>
<td>SRAM (cache disabled)</td>
</tr>
<tr>
<td>L1 Data memory</td>
<td>SRAM (cache disabled)</td>
</tr>
<tr>
<td>Cache Validity Bits</td>
<td>Invalid</td>
</tr>
<tr>
<td><strong>System</strong></td>
<td></td>
</tr>
<tr>
<td>Booting Methods</td>
<td>Determined by the values of BMODE pins at reset</td>
</tr>
<tr>
<td>MSEL Clock Frequency</td>
<td>Reset value = 10</td>
</tr>
<tr>
<td>PLL Bypass Mode</td>
<td>Disabled</td>
</tr>
<tr>
<td>VCO/Core Clock Ratio</td>
<td>Reset value = 1</td>
</tr>
<tr>
<td>VCO/System Clock Ratio</td>
<td>Reset value = 5</td>
</tr>
<tr>
<td>Peripheral Clocks</td>
<td>Disabled</td>
</tr>
</tbody>
</table>

System Reset and Power-up

Table 3-6 describes the five types of resets. Note all resets, except system software, reset the core.
## Operating Modes and States

### Table 3-6. Resets

<table>
<thead>
<tr>
<th>Reset</th>
<th>Source</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware Reset</td>
<td>The \texttt{RESET} pin causes a hardware reset.</td>
<td>Resets both the core and the peripherals, including the dynamic power management controller (DPMC). Resets the no boot on software reset bit in \texttt{SYSCR}. For more information, see “\texttt{SYSCR Register}” on page 3-14.</td>
</tr>
<tr>
<td>System Software Reset</td>
<td>Writing \texttt{b#111} to bits [2:0] in the system MMR \texttt{SWRST} at address \texttt{0xFFC0 0100} causes a system software reset.</td>
<td>Resets only the peripherals, excluding the RTC (real-time clock) block and most of the DPMC. The DPMC resets only the no boot on software reset bit in \texttt{SYSCR}. Does not reset the core. Does not initiate a boot sequence.</td>
</tr>
<tr>
<td>Watchdog Timer Reset</td>
<td>Programming the watchdog timer appropriately causes a watchdog timer reset.</td>
<td>Resets both the core and the peripherals, excluding the RTC block and most of the DPMC. The software reset register (\texttt{SWRST}) can be read to determine whether the reset source was the watchdog timer.</td>
</tr>
<tr>
<td>Core Double-Fault Reset</td>
<td>If the core enters a double-fault state, and the core double fault reset enable bit (\texttt{DOUBLE_FAULT}) is set in the \texttt{SWRST} register, then a software reset occurs.</td>
<td>Resets both the core and the peripherals, excluding the RTC block and most of the DPMC. The \texttt{SWRST} register can be read to determine whether the reset source was core double fault.</td>
</tr>
<tr>
<td>Core-Only Software Reset</td>
<td>This reset is caused by executing a \texttt{RAISE1} instruction or by setting the software reset (\texttt{SYSRST}) bit in the core debug control register (\texttt{DBGCTL}) via emulation software through the JTAG port. The \texttt{DBGCTL} register is not visible to the memory map.</td>
<td>Resets only the core. The peripherals do not recognize this reset.</td>
</tr>
</tbody>
</table>
Hardware Reset

The processor chip reset is an asynchronous reset event. The \texttt{RESET} input pin must be deasserted to perform a hardware reset. For more information, see \textit{ADSP-BF538/ADSP-BF538F Embedded Processor Data Sheet}.

A hardware-initiated reset results in a system-wide reset that includes both core and peripherals. After the \texttt{RESET} pin is deasserted, the processor ensures that all asynchronous peripherals have recognized and completed a reset. After the reset, the processor transitions into the boot mode sequence configured by the \texttt{BMODE} state.

The \texttt{BMODE[1:0]} pins are dedicated mode control pins. No other functions are shared with these pins, and they may be permanently strapped by tying them directly to either \texttt{VDD} or \texttt{VSS}. The pins and the corresponding bits in \texttt{SYSCR} configure the boot mode that is employed after hardware reset or system software reset. See “Reset” on page 4-44 and Table 4-11 on page 4-47 for further information.

SYSCR Register

The values sensed from the \texttt{BMODE[1:0]} pins are latched into the system reset configuration register (\texttt{SYSCR}) upon the deassertion of the \texttt{RESET} pin. The values are made available for software access and modification after the hardware reset sequence. Software can modify only the no boot on software reset bit.

The various configuration parameters are distributed to the appropriate destinations from \texttt{SYSCR} (see Figure 3-2).
Operating Modes and States

Software Resets and Watchdog Timer

A software reset may be initiated in three ways:

- By the watchdog timer, if appropriately configured
- By setting the system software reset field in the software reset register (see Figure 3-3)
- By the RAISE1 instruction

The watchdog timer resets both the core and the peripherals. A system software reset results in a reset of the peripherals without resetting the core and without initiating a booting sequence.

The system software reset must be performed while executing from Level 1 memory (either as cache or as SRAM).

When L1 instruction memory is configured as cache, make sure the system software reset sequence has been read into the cache.
System Reset and Power-up

After either the watchdog or system software reset is initiated, the processor ensures that all asynchronous peripherals have recognized and completed a reset.

For a reset generated by the watchdog timer, the processors transitions into the boot mode sequence. The boot mode is configured by the state of the BMODE and the no boot on software reset control bits.

If the no boot on software reset bit in SYSCR is cleared, the reset sequence is determined by the BMODE[1:0] control bits.

SWRST Register

A software reset can be initiated by setting the system software reset field in the software reset register (SWRST). Bit 15 indicates whether a software reset has occurred since the last time SWRST was read. Bit 14 and bit 13, respectively, indicate whether the software watchdog timer or a core double fault has generated a software reset. Bits [15:13] are read-only and cleared when the register is read. Bits [3:0] are read/write.

When the BMODE pins are not set to b#00 and the no boot on software reset bit in SYSCR is set, the processor starts executing from the start of on-chip L1 memory. In this configuration, the core begins fetching instructions from the beginning of on-chip L1 memory.

When the BMODE pins are set to b#00 the core begins fetching instructions from address 0x2000 0000 (the beginning of ASYNC bank 0).
Operating Modes and States

Core-Only Software Reset

A core-only software reset is initiated by executing the **RAISE 1** instruction or by setting the software reset (**SYSRST**) bit in the core debug control register (**DBGCTL**) via emulation software through the JTAG port. (**DBGCTL** is not visible to the memory map.)

A core-only software reset affects only the state of the core. Note the system resources may be in an undetermined or even unreliable state, depending on the system activity during the reset period.

Core and System Reset

To perform a system and core reset, use the code sequence shown in **Listing 3-4**. As described in the code comments in the listing, the system soft reset takes five system clock cycles to complete, so a delay loop is needed. This code must reside in L1 memory for the system soft reset to work properly.
System Reset and Power-up

Listing 3-4. Core and System Reset

/* Issue system soft reset */
P0.L = LO(SWRST) ;
P0.H = HI(SWRST) ;
R0.L = 0x0007 ;
W[P0] = R0 ;
SSYNC ;

/* Wait for System reset to complete (needs to be 5 SCLKs). */
/* Assuming a worst case CCLK:SCLK ratio (15:1), use 5*15 = 75 */
/* as the loop count. */
P1 = 75;
LSETUP(start, end) LCO = P1 ;
start:
end:
NOP :

/* Clear system soft reset */
R0.L = 0x0000 ;
W[P0] = R0 ;
SSYNC ;

/* Core reset - forces reboot */
RAISE 1 ;
Operating Modes and States

Booting Methods

The internal boot ROM includes a small boot kernel that can either be bypassed or used to load user code from an external memory device. See Table 4-10 on page 4-45 for further information. The boot kernel reads the BMODE[1:0] pin state at reset to identify the download source (see Table 4-7 on page 4-24). When in boot mode 0, the processor is set to execute from 16-bit wide external memory at address 0x2000 0000 (ASYNC bank 0).

Several boot methods are available in which user code can be loaded from an external memory device or a host device (as in the case of SPI slave mode booting). For these modes, the boot kernel sets up the selected peripheral based on the BMODE[1:0] pin settings.

For each boot mode, user code read in from the memory device is placed at the starting location of L1 memory. Additional sections are read into internal memory as specified within headers in the loader file. The boot kernel terminates the boot process with a jump to the start of the L1 instruction memory space. The processor then begins execution from this address.

If booting from the serial peripheral interface (SPI0), general-purpose flag pin 2 (PF2) is used as the SPI-chip select. This line must be connected for proper operation.

A core-only software reset also vectors the core to the boot ROM. Only the core is reset with the core-only software reset; this reset does not affect the rest of the system. The boot ROM kernel detects a no boot on software reset condition in SYSCR to avoid initiating a download. If this bit is set on a software reset, the processor skips the normal boot sequence and jumps to the beginning of L1 memory and begins execution.
The boot kernel assumes these conditions for the flash boot mode ($\text{BMODE} = 01$):

- asynchronous memory bank (AMB) 0 enabled
- 16-bit packing for AMB 0 enabled
- bank 0 RDY is set to active high
- bank 0 hold time (read/write deasserted to $\overline{AOE}$ deasserted) = 3 cycles
- bank 0 read/write access times = 15 cycles

For SPI master mode boot ($\text{BMODE} = 11$), the boot kernel assumes that the SPI0 baud rate is 500 kHz. SPI serial EEPROMs that are 8-bit, 16-bit, and 24-bit addressable are supported. The SPI uses the PF2 output pin to select a single SPI EEPROM device. The SPI0 controller submits successive read commands at addresses 0x00, 0x0000, and 0x000000 until a valid 8-, 16-, or 24-bit addressable EEPROM is detected. It then begins clocking data into the beginning of L1 instruction memory.

The MISOx pin must be pulled high for SPI master mode booting ($\text{BMODE} = 11$).

For each of the boot modes, 10-byte headers are first read from an external memory device. The header specifies the number of bytes to be transferred and the memory destination address. Once all blocks are loaded, program execution commences from the start of L1 instruction SRAM.

For SPI slave mode boot ($\text{BMODE} = 10$), the hardware configuration shown in Figure 3-4 is assumed.
Operating Modes and States

The user-defined GPIO port \( \text{PF}_x \) is an output on the Blackfin processor and an input on the host device. This pin allows the processor to hold off the host device from sending data during certain sections of the boot process. When this pin is deasserted, the host can continue to send bytes to the processor.

![Figure 3-4. SPI Slave Boot Mode](image)

Figure 3-4. SPI Slave Boot Mode
4 PROGRAM SEQUENCER

In the processor, the program sequencer controls program flow, constantly providing the address of the next instruction to be executed by other parts of the processor. Program flow in the chip is mostly linear, with the processor executing program instructions sequentially.

The linear flow varies occasionally when the program uses nonsequential program structures, such as those illustrated in Figure 4-1. Nonsequential structures direct the processor to execute an instruction that is not at the next sequential address. These structures include:

- **Loops.** One sequence of instructions executes several times with zero overhead.

- **Subroutines.** The processor temporarily interrupts sequential flow to execute instructions from another part of memory.

- **Jumps.** Program flow transfers permanently to another part of memory.

- **Interrupts and Exceptions.** A runtime event or instruction triggers the execution of a subroutine.

- **Idle.** An instruction causes the processor to stop operating and hold its current state until an interrupt occurs. Then, the processor services the interrupt and continues normal execution.
The sequencer manages execution of these program structures by selecting the address of the next instruction to execute.

Figure 4-1. Program Flow Variations
The fetched address enters the instruction pipeline, ending with the program counter (PC). The pipeline contains the 32-bit addresses of the instructions currently being fetched, decoded, and executed. The PC couples with the \texttt{RETo} registers, which store return addresses. All addresses generated by the sequencer are 32-bit memory instruction addresses.

To manage events, the sequencer’s event controller handles interrupt and event processing, determines whether an interrupt is masked, and generates the appropriate event vector address.

In addition to providing data addresses, the data address generators (DAGs) can provide instruction addresses for the sequencer’s indirect branches.

The sequencer evaluates conditional instructions and loop termination conditions. The loop registers support nested loops. The memory-mapped registers (MMRs) store information used to implement interrupt service routines.
Sequencer Related Registers

Table 4-1 lists the registers within the processor that are related to the sequencer. Except for the PC and SEQSTAT registers, all sequencer related registers are directly readable and writable. Manually pushing or popping registers to or from the stack is done using the explicit instructions [--SP] = Rn (for push) or Rn = [SP++] (for pop).

Table 4-1. Sequencer Related Registers

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEQSTAT</td>
<td>Sequencer status register</td>
</tr>
<tr>
<td>RETX</td>
<td>Return address registers: see “Events and Sequencing” on page 4-18.</td>
</tr>
<tr>
<td>RETN</td>
<td>Exception return</td>
</tr>
<tr>
<td>RETI</td>
<td>NMI return</td>
</tr>
<tr>
<td>RETE</td>
<td>Interrupt return</td>
</tr>
<tr>
<td>RETS</td>
<td>Emulation return</td>
</tr>
<tr>
<td>LC0, LC1</td>
<td>Subroutine return</td>
</tr>
<tr>
<td>LT0, LT1</td>
<td>Zero-overhead loop registers:</td>
</tr>
<tr>
<td>LB0, LB1</td>
<td>Loop counters</td>
</tr>
<tr>
<td>FP, SP</td>
<td>Loop tops</td>
</tr>
<tr>
<td></td>
<td>Loop bottoms</td>
</tr>
<tr>
<td>SYSCFG</td>
<td>Frame pointer and stack pointer: see “Data Address Generators” on page 5-1.</td>
</tr>
<tr>
<td>CYCLES, CYCLES2</td>
<td>System configuration register</td>
</tr>
<tr>
<td>PC</td>
<td>Cycle counters: see “Blackfin Processor Debug” on page 22-1.</td>
</tr>
</tbody>
</table>

Sequencer Status (SEQSTAT) Register

The sequencer status register (SEQSTAT) contains information about the current state of the sequencer as well as diagnostic information from the last event. SEQSTAT is a read-only register and is accessible only in supervisor mode.
Program Sequencer

Zero-Overhead Loop (LCx, LTx, LBx) Registers

Two sets of zero-overhead loop registers implement loops, using hardware counters instead of software instructions to evaluate loop conditions. After evaluation, processing branches to a new target address. Both sets of registers include the loop counter (LCx), loop top (LTx), and loop bottom (LBx) registers.

Table 4-2 describes the 32-bit loop register sets.
Table 4-2. Loop Registers

<table>
<thead>
<tr>
<th>Registers</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>LC0, LC1</td>
<td>Loop Counters</td>
<td>Maintain a count of the remaining iterations of the loop</td>
</tr>
<tr>
<td>LT0, LT1</td>
<td>Loop Tops</td>
<td>Hold the address of the first instruction within a loop</td>
</tr>
<tr>
<td>LB0, LB1</td>
<td>Loop Bottoms</td>
<td>Hold the address of the last instruction of the loop</td>
</tr>
</tbody>
</table>

**System Configuration (SYSCFG) Register**

The system configuration register (SYSCFG) controls the configuration of the processor. This register is accessible only from the supervisor mode.

**System Configuration Register (SYSCFG)**

![Figure 4-3. System Configuration Register](image)

- **CCEN (Cycle-Counter Enable)**
  - 0 - Disable 64-bit, free-running cycle counter
  - 1 - Enable 64-bit, free-running cycle counter

- **SSSTEP (Supervisor Single Step)**
  - When set, a Supervisor exception is taken after each instruction is executed. It applies only to User mode, or when processing interrupts in Supervisor mode. It is ignored if the core is processing an exception or higher-priority event. If precise exception timing is required, CSYNC must be used after setting this bit.
Instruction Pipeline

The program sequencer determines the next instruction address by examining both the current instruction being executed and the current state of the processor. If no conditions require otherwise, the processor executes instructions from memory in sequential order by incrementing the look ahead address.

The processor has a ten-stage instruction pipeline.

Table 4-3. Stages of Instruction Pipeline

<table>
<thead>
<tr>
<th>Pipeline Stage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Fetch 1 (IF1)</td>
<td>Start instruction memory access.</td>
</tr>
<tr>
<td>Instruction Fetch 2 (IF2)</td>
<td>Intermediate memory pipeline.</td>
</tr>
<tr>
<td>Instruction Fetch 3 (IF3)</td>
<td>Finish L1 instruction memory access.</td>
</tr>
<tr>
<td>Instruction Decode (DEC)</td>
<td>Align instruction, start instruction decode, and access pointer register file.</td>
</tr>
<tr>
<td>Address Calculation (AC)</td>
<td>Calculate data addresses and branch target address.</td>
</tr>
<tr>
<td>Execute 1 (EX1)</td>
<td>Start access of data memory.</td>
</tr>
<tr>
<td>Execute 2 (EX2)</td>
<td>Register file read.</td>
</tr>
<tr>
<td>Execute 3 (EX3)</td>
<td>Finish accesses of data memory and start execution of dual cycle instructions.</td>
</tr>
<tr>
<td>Execute 4 (EX4)</td>
<td>Execute single cycle instructions.</td>
</tr>
<tr>
<td>Write Back (WB)</td>
<td>Write states to data and pointer register files and process events.</td>
</tr>
</tbody>
</table>
Figure 4-4 shows a diagram of the pipeline.

![Processor Pipeline Diagram](image)

**Figure 4-4. Processor Pipeline**

The sequencer decodes and distributes operations to the instruction memory unit and instruction alignment unit. It also controls stalling and invalidating the instructions in the pipeline. The sequencer ensures that the pipeline is fully interlocked and that the programmer does not need to manage the pipeline.

The instruction fetch and branch logic generates 32-bit fetch addresses for the instruction memory unit. The instruction alignment unit returns instructions and their width information at the beginning of the DEC stage.

For each instruction type (16-, 32-, or 64-bit), the alignment unit ensures that the alignment buffers have enough valid data to be able to provide an instruction every cycle. Since the instructions can be 16, 32, or 64 bits wide, the alignment unit may not need to fetch data from the cache every cycle. For example, for a series of 16-bit instructions, the alignment unit gets data from the instruction memory unit once in 4 cycles. The alignment logic requests the next instruction address based on the status of the alignment buffers. The sequencer responds by generating the next fetch address in the next cycle, provided there is no change of flow.

The sequencer holds the fetch address until it receives a request from the alignment logic or until a change of flow occurs. It always increments the previous fetch address by 8 (the next 8 bytes). If a change of flow occurs, such as a branch or an interrupt, the sequencer communicates it to the instruction memory unit, which invalidates the data in the alignment unit.
The execution unit contains two 16-bit multipliers, two 40-bit ALUs, two 40-bit accumulators, one 40-bit shifter, a video unit (which adds 8-bit ALU support), and an 8-entry 32-bit data register file.

Register file reads occur in the EX2 pipeline stage (for operands). Writes occur in the WB stage (for stores). The multipliers and the video unit are active in the EX3 stage, and the ALUs and shifter are active in the EX4 stage. The accumulators are written at the end of the EX4 stage.

Any nonsequential program flow can potentially decrease the processor’s instruction throughput. Nonsequential program operations include:

- Jumps
- Subroutine calls and returns
- Interrupts and returns
- Loops

**Branches and Sequencing**

One type of nonsequential program flow that the sequencer supports is branching. A branch occurs when a **JUMP** or **CALL** instruction begins execution at a new location other than the next sequential address. For descriptions of how to use the **JUMP** and **CALL** instructions, see *Blackfin Processor Programming Reference*.

A **JUMP** or a **CALL** instruction transfers program flow to another memory location. The difference between a **JUMP** and a **CALL** is that a **CALL** automatically loads the return address into the ** RETS** register. The return address is the next sequential address after the **CALL** instruction. This push makes the address available for the **CALL** instruction’s matching return instruction, allowing easy return from the subroutine.
Branches and Sequencing

A return instruction causes the sequencer to fetch the instruction at the return address, which is stored in the RETS register (for subroutine returns). The types of return instructions are return from subroutine (RTS), return from interrupt (RTI), return from exception (RTX), return from emulation (RTE), and return from nonmaskable interrupt (RTN). Each return type has its own register for holding the return address.

JUMP instructions can be conditional, depending on the status of the CC bit of the ASTAT register. They are immediate and may not be delayed. The program sequencer can evaluate the CC status bit to decide whether to execute a branch. If no condition is specified, the branch is always taken.

Conditional JUMP instructions use static branch prediction to reduce the branch latency caused by the length of the pipeline.

Branches can be direct or indirect. A direct branch address is determined solely by the instruction word (for example, JUMP 0x30), while an indirect branch gets its address from the contents of a DAG register (for example, JUMP(P3)).

All types of JUMPS and CALLs can be PC-relative. The indirect JUMP and CALL can be absolute or PC-relative.

Direct Short and Long Jumps

The sequencer supports both short and long jumps. The target of the branch is a PC-relative address from the location of the instruction, plus an offset. The PC-relative offset for the short jump is a 13-bit immediate value that must be a multiple of two (bit zero must be a zero). The 13-bit value gives an effective dynamic range of –4096 to +4094 bytes.

The PC-relative offset for the long jump is a 25-bit immediate value that must also be a multiple of two (bit zero must be a zero). The 25-bit value gives an effective dynamic range of –16,777,216 to +16,777,214 bytes.
If, at the time of writing the program, the destination is known to be less than a 13-bit offset from the current PC value, then the \texttt{JUMP.S 0xnnnn} instruction may be used. If the destination requires more than a 13-bit offset, then the \texttt{JUMP.L 0xnnnnnnn} instruction must be used. If the destination offset is unknown and development tools must evaluate the offset, then use the instruction \texttt{JUMP 0xnnnnnnnn}. Upon disassembly, the instruction is replaced by the appropriate \texttt{JUMP.S} or \texttt{JUMP.L} instruction.

**Direct Call**

The \texttt{CALL} instruction is a branch instruction that copies the address of the instruction which would have executed next (had the \texttt{CALL} not executed) into the \texttt{RETS} register. The direct \texttt{CALL} instruction has a 25-bit PC-relative offset that must be a multiple of two (bit zero must be a zero). The 25-bit value gives an effective dynamic range of \(-16,777,216\) to \(+16,777,214\) bytes.

**Indirect Branch and Call**

The indirect \texttt{JUMP} and \texttt{CALL} instructions get their destination address from a data address generator (DAG) P-register. For the \texttt{CALL} instruction, the \texttt{RETS} register is loaded with the address of the instruction which would have executed next in the absence of the \texttt{CALL} instruction.

For example:

\begin{verbatim}
JUMP (P3) ;
CALL (P0) ;
\end{verbatim}
Branches and Sequencing

PC-Relative Indirect Branch and Call

The PC-relative indirect JUMP and CALL instructions use the contents of a P-register as an offset to the branch target. For the CALL instruction, the RETS register is loaded with the address of the instruction which would have executed next (had the CALL not executed).

For example:

JUMP (PC + P3);

CALL (PC + P0);

Condition Code Flag

The processor supports a condition code (CC) flag bit, which is used to resolve the direction of a branch. This flag may be accessed eight ways:

1. A conditional branch is resolved by the value in CC.
2. A data register value may be copied into CC, and the value in CC may be copied to a data register.
3. The BITTST instruction accesses the CC flag.
4. A status flag may be copied into CC, and the value in CC may be copied to a status flag.
5. CC may be set to the result of a pointer register comparison.
6. CC may be set to the result of a data register comparison.
7. Some shifter instructions (rotate or BXOR) use CC as a portion of the shift operand/result.
8. Test and set instructions can set and clear the CC bit.
These eight ways of accessing the \( \text{\texttt{CC}} \) bit are used to control program flow. The branch is explicitly separated from the instruction that sets the arithmetic flags. A single bit resides in the instruction encoding that specifies the interpretation for the value of \( \text{\texttt{CC}} \). The interpretation is to “branch on true” or “branch on false.”

The comparison operations have the form \( \text{\texttt{CC}} = \text{\texttt{expr}} \) where \( \text{\texttt{expr}} \) involves a pair of registers of the same type (for example, data registers or pointer registers, or a single register and a small immediate constant). The small immediate constant is a 3-bit (–4 through 3) signed number for signed comparisons and a 3-bit (0 through 7) unsigned number for unsigned comparisons.

The sense of \( \text{\texttt{CC}} \) is determined by equal (\( \text{\texttt{==}} \)), less than (\( \text{\texttt{<}} \)), and less than or equal to (\( \text{\texttt{<=}} \)). There are also bit test operations that test whether a bit in a 32-bit R-register is set.

**Conditional Branches**

The sequencer supports conditional branches. These are \texttt{JUMP} instructions whose execution branches or continues linearly depending on the value of the \( \text{\texttt{CC}} \) bit. The target of the branch is a PC-relative address from the location of the instruction plus an offset. The PC-relative offset is an 11-bit immediate value that must be a multiple of two (bit zero must be a zero). This gives an effective dynamic range of –1024 to +1022 bytes.

For example, the following instruction tests the \( \text{\texttt{CC}} \) flag and, if it is positive, jumps to a location identified by the label \texttt{dest_address}:

\[
\text{\texttt{IF CC JUMP dest_address ;}}
\]
Branches and Sequencing

Conditional Register Move

Register moves can be performed depending on whether the value of the CC flag is true or false (1 or 0). In some cases, using this instruction instead of a branch eliminates the cycles lost because of the branch. These conditional moves can be done between any R- or P-registers (including SP and FP).

Example code:

```
IF CC R0 = P0 :
```

Branch Prediction

The sequencer supports static branch prediction to accelerate execution of conditional branches. These branches are executed based on the state of the CC bit.

In the EX4 stage, the sequencer compares the actual CC bit value to the predicted value. If the value was predicted incorrectly, the branch is corrected, and the correct address is available for the WB stage of the pipeline.

The branch latency for conditional branches is as follows:

- If prediction was “not to take branch,” and branch was actually not taken: 0 CCLK cycles.
- If prediction was “not to take branch,” and branch was actually taken: 8 CCLK cycles.
- If prediction was “to take branch,” and branch was actually taken: 4 CCLK cycles.
- If prediction was “to take branch,” and branch was actually not taken: 8 CCLK cycles.
For all unconditional branches, the branch target address computed in the AC stage of the pipeline is sent to the instruction fetch address bus at the beginning of the EX1 stage. All unconditional branches have a latency of 4 CCLK cycles.

Consider the example in Table 4-4.

Table 4-4. Branch Prediction

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>If CC JUMP dest (bp)</td>
<td>This instruction tests the CC flag, and if it is set, jumps to a location, identified by the label, dest. If the CC flag is set, then the branch is correctly predicted and the branch latency is reduced. Otherwise the branch is incorrectly predicted and the branch latency increases.</td>
</tr>
</tbody>
</table>

Loops and Sequencing

The sequencer supports a mechanism of zero-overhead looping. The sequencer contains two loop units, each containing three registers. Each loop unit has a loop top register (LT0, LT1), a loop bottom register (LB0, LB1), and a loop count register (LC0, LC1).

When an instruction at address X is executed, and X matches the contents of LB0, then the next instruction executed will be from the address in LT0. In other words, when PC==LB0, then an implicit jump to LT0 is executed.

A loopback only occurs when the count is greater than or equal to 2. If the count is nonzero, then the count is decremented by 1. For example, consider the case of a loop with two iterations. At the beginning, the count is 2. Upon reaching the first loop end, the count is decremented to 1 and the program flow jumps back to the top of the loop (to execute a second time). Upon reaching the end of the loop again, the count is decremented to zero but no loopback occurs (because the body of the loop has already been executed twice).
Since there are two loop units, loop unit 1 is assigned higher priority so that it can be used as the inner loop in a nested loop structure. In other words, a loopback caused by loop unit 1 on a particular instruction (PC==LB1, LC1>=2) will prevent loop unit 0 from looping back on that same instruction, even if the address matches. Loop unit 0 is allowed to loop back only after the loop count 1 is exhausted.

The `LSETUP` instruction can be used to load all three registers of a loop unit at once. Each loop register can also be loaded individually with a register transfer, but this incurs a significant overhead if the loop count is nonzero (the loop is active) at the time of the transfer.

The following code example shows a loop that contains two instructions and iterates 32 times.

```
Loop
P5 = 0x20 ;
LSETUP ( lp_start, lp_end ) LCO = P5 ;
lp_start :
R5 = R0 + R1 ( ns ) || R2 = [P2++] || R3 = [ I1++] ;
lp_end : R5 = R5 + R2 ;
```

Two sets of loop registers are used to manage two nested loops:

- `LC[1:0]` – the loop count registers
- `LT[1:0]` – the loop top address registers
- `LB[1:0]` – the loop bottom address registers

When executing an `LSETUP` instruction, the program sequencer loads the address of the loop’s last instruction into `LBx` and the address of the loop’s first instruction into `LTx`. The top and bottom addresses of the loop are computed as PC-relative addresses from the `LSETUP` instruction plus an offset. In each case, the offset value is added to the location of the `LSETUP` instruction.
**Program Sequencer**

\( \text{LC0} \) and \( \text{LC1} \) are unsigned 32-bit registers, each supporting \( 2^{32} - 1 \) iterations through the loop.

When \( \text{LCx} = 0 \), the loop is disabled, and a single pass of the code executes.

The processor supports a four-location instruction loop buffer that reduces instruction fetches while in loops. If the loop code contains four or fewer instructions, then no fetches to instruction memory are necessary for any number of loop iterations, because the instructions are stored locally. The loop buffer effectively eliminates the instruction fetch time in loops with more than four instructions by allowing fetches to take place while instructions in the loop buffer are being executed.

A four-cycle latency occurs on the first loopback when the \( \text{LSETUP} \) specifies a nonzero start offset (\( \text{lps}_{\text{start}} \)). Therefore, zero start offsets are preferred.

The processor has no restrictions regarding which instructions can occur in a loop end position. Branches and calls are allowed in that position.

### Table 4-5. Loop Registers

<table>
<thead>
<tr>
<th>First/Last Address of the Loop</th>
<th>PC-Relative Offset Used to Compute the Loop Start Address</th>
<th>Effective Range of the Loop Start Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top / First</td>
<td>5-bit signed immediate; must be a multiple of 2.</td>
<td>0 to 30 bytes away from ( \text{LSETUP} ) instruction.</td>
</tr>
<tr>
<td>Bottom / Last</td>
<td>11-bit signed immediate; must be a multiple of 2.</td>
<td>0 to 2046 bytes away from ( \text{LSETUP} ) instruction (the defined loop can be 2046 bytes long).</td>
</tr>
</tbody>
</table>
Events and Sequencing

The event controller of the processor manages five types of activities:

- Emulation
- Reset
- Non-maskable interrupts (NMI)
- Exceptions
- Interrupts

Note the word event describes all five types. The event controller manages fifteen events in all: emulation, reset, NMI, exception, and 11 interrupts.

An interrupt is an event that changes normal processor instruction flow and is asynchronous to program flow. In contrast, an exception is a software initiated event whose effects are synchronous to program flow.

The event system is nested and prioritized. Consequently, several service routines may be active at any time, and a low priority event may be pre-empted by one of higher priority.

The processor employs a two-level event control mechanism. The processor system interrupt controllers (SICx) work with the core event controller (CEC) to prioritize and control all system interrupts. The SICx provides mapping between the many peripheral interrupt sources and the prioritized general-purpose interrupt inputs of the core. This mapping is programmable, and individual interrupt sources can be masked in the SICx.

The CEC supports nine general-purpose interrupts (IVG7 - IVG15) in addition to the dedicated interrupt and exception events that are described in Table 4-6. It is recommended that the lowest two priority interrupts
(IVG14 and IVG15) be reserved for software interrupt handlers, leaving seven prioritized interrupt inputs (IVG7 - IVG13) to support the system. Refer to the following table.

Table 4-6. System and Core Event Mapping

<table>
<thead>
<tr>
<th>Event Source</th>
<th>Core Event Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>core events</td>
<td></td>
</tr>
<tr>
<td>emulation (highest priority)</td>
<td>EMU</td>
</tr>
<tr>
<td>reset</td>
<td>RST</td>
</tr>
<tr>
<td>NMI</td>
<td>NMI</td>
</tr>
<tr>
<td>exception</td>
<td>EVX</td>
</tr>
<tr>
<td>reserved</td>
<td>–</td>
</tr>
<tr>
<td>hardware error</td>
<td>IVHW</td>
</tr>
<tr>
<td>core timer</td>
<td>IVTMR</td>
</tr>
<tr>
<td>system interrupts</td>
<td></td>
</tr>
<tr>
<td>PLL wake-up interrupt</td>
<td>IVG7</td>
</tr>
<tr>
<td>DMA controller 0 error (generic)</td>
<td></td>
</tr>
<tr>
<td>DMA controller 1 error (generic)</td>
<td></td>
</tr>
<tr>
<td>PPI error interrupt</td>
<td></td>
</tr>
<tr>
<td>SPORT0 error interrupt</td>
<td></td>
</tr>
<tr>
<td>SPORT1 error interrupt</td>
<td></td>
</tr>
<tr>
<td>SPORT2 error interrupt</td>
<td></td>
</tr>
<tr>
<td>SPORT3 error interrupt</td>
<td></td>
</tr>
<tr>
<td>SPI0 error interrupt</td>
<td></td>
</tr>
<tr>
<td>SPI1 error interrupt</td>
<td></td>
</tr>
<tr>
<td>SPI2 error interrupt</td>
<td></td>
</tr>
<tr>
<td>UART0 error interrupt</td>
<td></td>
</tr>
<tr>
<td>UART1 error interrupt</td>
<td></td>
</tr>
<tr>
<td>UART2 error interrupt</td>
<td></td>
</tr>
<tr>
<td>CAN error interrupt</td>
<td></td>
</tr>
<tr>
<td>real-time clock interrupts</td>
<td></td>
</tr>
<tr>
<td>DMA0 interrupt (PPI)</td>
<td>IVG8</td>
</tr>
</tbody>
</table>

Refer to the following table.
### Events and Sequencing

Table 4-6. System and Core Event Mapping (Cont’d)

<table>
<thead>
<tr>
<th>Event Source</th>
<th>Core Event Name</th>
</tr>
</thead>
</table>
| DMA1 interrupt (SPORT0 receive)  
DMA2 interrupt (SPORT0 transmit)  
DMA3 interrupt (SPORT1 receive)  
DMA4 interrupt (SPORT1 transmit)  
DMA8 interrupt (SPORT2 receive)  
DMA9 interrupt (SPORT2 transmit)  
DMA10 interrupt (SPORT3 receive)  
DMA11 interrupt (SPORT3 transmit)  
DMA12 interrupt (unassigned)  
DMA13 interrupt (unassigned) | IVG9 |
| DMA5 interrupt (SPI0)  
DMA14 interrupt (SPI1)  
DMA15 interrupt (SPI2)  
DMA6 interrupt (UART0 receive)  
DMA7 interrupt (UART0 transmit)  
DMA16 interrupt (UART1 receive)  
DMA17 interrupt (UART1 transmit)  
DMA18 interrupt (UART2 receive)  
DMA19 interrupt (UART2 transmit) | IVG10 |
| timer0, timer1, timer2 interrupts  
TWI0 interrupt  
TWI1 interrupt  
CAN receive interrupt  
CAN transmit interrupt | IVG11 |
| GPIO interrupt A/B | IVG12 |
| MDMA0 stream 0 (memory DMA)  
MDMA0 stream 1 (memory DMA)  
MDMA1 stream 0 (memory DMA)  
MDMA1 stream 1 (memory DMA)  
software watchdog timer | IVG13 |

Note the system interrupt to core event mappings shown are the default values at reset and can be changed by software.
System Interrupt Processing

Referring to Figure 4-5, note when an interrupt (interrupt A) is generated by an interrupt-enabled peripheral:

- **SIC_ISRx** logs the request and keeps track of system interrupts that are asserted but not yet serviced (that is, an interrupt service routine has not yet cleared the interrupt).

- **SIC_IWRx** checks to see if it should wake up the core from an idled state based on this interrupt request.

- **SIC_IMASKx** masks off or enables interrupts from peripherals at the system level. If interrupt A is not masked, the request proceeds to Step 4.

- **ILAT** adds interrupt A to its log of interrupts latched by the core but not yet actively being serviced.

- **IMASK** masks off or enables events of different core priorities. If the **IVGx** event corresponding to interrupt A is not masked, the process proceeds to Step 7.

The event vector table (EVT) is accessed to look up the appropriate vector for interrupt A’s interrupt service routine (ISR).

When the event vector for interrupt A has entered the core pipeline, the appropriate **IPEND** bit is set, which clears the respective **ILAT** bit. Thus, **IPEND** tracks all pending interrupts, as well as those being presently serviced.
When the interrupt service routine for interrupt A has been executed, the RTI instruction clears the appropriate `IPEND` bit. However, the relevant `SIC_ISRx` bit is not cleared unless the interrupt service routine clears the mechanism that generated interrupt A, or if the process of servicing the interrupt clears this bit.

It should be noted that emulation, reset, NMI, and exception events, as well as hardware error (`IVHW`) and core timer (`IVTMR`) interrupt requests, enter the interrupt processing chain at the `ILAT` level and are not affected by the system-level interrupt registers (`SIC_IWRx`, `SIC_ISRx`, `SIC_IMASKx`, `SIC_IARx`).

If multiple interrupt sources share a single core interrupt, then the ISR must identify the peripheral that generated the interrupt. The ISR may then need to interrogate the peripheral to determine the appropriate action to take.
The processor system has numerous peripherals, which therefore require many supporting interrupts. Table 4-7 lists:

- Peripheral interrupt source
- Peripheral interrupt ID used in the system interrupt assignment registers (SIC_IARx). See “System Interrupt Assignment (SIC_IARx) Registers” on page 4-34.
- General-purpose interrupt of the core to which the interrupt maps at reset
The core interrupt ID used in the system interrupt assignment registers (SIC_IARx). See “System Interrupt Assignment (SIC_IARx) Registers” on page 4-34.

Table 4-7. Peripheral Interrupt Source Reset State

<table>
<thead>
<tr>
<th>Peripheral Interrupt Source</th>
<th>Peripheral Interrupt ID</th>
<th>General-Purpose Interrupt (Assignment at Reset)</th>
<th>Core Interrupt ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLL wake-up interrupt</td>
<td>0</td>
<td>IVG7</td>
<td>0</td>
</tr>
<tr>
<td>DMA controller 0 error (generic)</td>
<td>1</td>
<td>IVG7</td>
<td>0</td>
</tr>
<tr>
<td>PPI error interrupt</td>
<td>2</td>
<td>IVG7</td>
<td>0</td>
</tr>
<tr>
<td>SPORT0 error interrupt</td>
<td>3</td>
<td>IVG7</td>
<td>0</td>
</tr>
<tr>
<td>SPORT1 error interrupt</td>
<td>4</td>
<td>IVG7</td>
<td>0</td>
</tr>
<tr>
<td>SPI0 error interrupt</td>
<td>5</td>
<td>IVG7</td>
<td>0</td>
</tr>
<tr>
<td>UART0 error interrupt</td>
<td>6</td>
<td>IVG7</td>
<td>0</td>
</tr>
<tr>
<td>Real-time clock interrupts</td>
<td>7</td>
<td>IVG8</td>
<td>1</td>
</tr>
<tr>
<td>DMA 0 interrupt (PPI)</td>
<td>8</td>
<td>IVG8</td>
<td>1</td>
</tr>
<tr>
<td>DMA 1 interrupt (SPORT0 receive)</td>
<td>9</td>
<td>IVG9</td>
<td>2</td>
</tr>
<tr>
<td>DMA 2 interrupt (SPORT0 transmit)</td>
<td>10</td>
<td>IVG9</td>
<td>2</td>
</tr>
<tr>
<td>DMA 3 interrupt (SPORT1 receive)</td>
<td>11</td>
<td>IVG9</td>
<td>2</td>
</tr>
<tr>
<td>DMA 4 interrupt (SPORT1 transmit)</td>
<td>12</td>
<td>IVG9</td>
<td>2</td>
</tr>
<tr>
<td>DMA 5 interrupt (SPI0)</td>
<td>13</td>
<td>IVG10</td>
<td>3</td>
</tr>
<tr>
<td>DMA 6 interrupt (UART0 receive)</td>
<td>14</td>
<td>IVG10</td>
<td>3</td>
</tr>
<tr>
<td>DMA 7 interrupt (UART0 transmit)</td>
<td>15</td>
<td>IVG10</td>
<td>3</td>
</tr>
<tr>
<td>Timer0 interrupt</td>
<td>16</td>
<td>IVG11</td>
<td>4</td>
</tr>
<tr>
<td>Timer1 interrupt</td>
<td>17</td>
<td>IVG11</td>
<td>4</td>
</tr>
<tr>
<td>Timer2 interrupt</td>
<td>18</td>
<td>IVG11</td>
<td>4</td>
</tr>
<tr>
<td>GPIO interrupt A</td>
<td>19</td>
<td>IVG12</td>
<td>5</td>
</tr>
<tr>
<td>GPIO interrupt B</td>
<td>20</td>
<td>IVG12</td>
<td>5</td>
</tr>
</tbody>
</table>
Table 4-7. Peripheral Interrupt Source Reset State (Cont’d)

<table>
<thead>
<tr>
<th>Peripheral Interrupt Source</th>
<th>Peripheral Interrupt ID</th>
<th>General-Purpose Interrupt (Assignment at Reset)</th>
<th>Core Interrupt ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA 8/9 interrupt (memory DMA stream 0)</td>
<td>21</td>
<td>IVG13</td>
<td>6</td>
</tr>
<tr>
<td>DMA 10/11 interrupt (memory DMA stream 1)</td>
<td>22</td>
<td>IVG13</td>
<td>6</td>
</tr>
<tr>
<td>Software watchdog timer interrupt</td>
<td>23</td>
<td>IVG13</td>
<td>6</td>
</tr>
<tr>
<td>DMA controller 1 error (generic)</td>
<td>24</td>
<td>IVG7</td>
<td>0</td>
</tr>
<tr>
<td>SPORT2 error interrupt</td>
<td>25</td>
<td>IVG7</td>
<td>0</td>
</tr>
<tr>
<td>SPORT3 error interrupt</td>
<td>26</td>
<td>IVG7</td>
<td>0</td>
</tr>
<tr>
<td>SPI1 error interrupt</td>
<td>28</td>
<td>IVG7</td>
<td>0</td>
</tr>
<tr>
<td>SPI2 error interrupt</td>
<td>29</td>
<td>IVG7</td>
<td>0</td>
</tr>
<tr>
<td>UART1 error interrupt</td>
<td>30</td>
<td>IVG7</td>
<td>0</td>
</tr>
<tr>
<td>UART2 error interrupt</td>
<td>31</td>
<td>IVG7</td>
<td>0</td>
</tr>
<tr>
<td>CAN error interrupt</td>
<td>32</td>
<td>IVG7</td>
<td>0</td>
</tr>
<tr>
<td>DMA 8 interrupt (SPORT2 receive)</td>
<td>33</td>
<td>IVG9</td>
<td>2</td>
</tr>
<tr>
<td>DMA 9 interrupt (SPORT2 transmit)</td>
<td>34</td>
<td>IVG9</td>
<td>2</td>
</tr>
<tr>
<td>DMA 10 interrupt (SPORT3 receive)</td>
<td>35</td>
<td>IVG9</td>
<td>2</td>
</tr>
<tr>
<td>DMA 11 interrupt (SPORT3 transmit)</td>
<td>36</td>
<td>IVG9</td>
<td>2</td>
</tr>
<tr>
<td>DMA 12 interrupt</td>
<td>37</td>
<td>IVG9</td>
<td>2</td>
</tr>
<tr>
<td>DMA 13 interrupt</td>
<td>38</td>
<td>IVG9</td>
<td>2</td>
</tr>
<tr>
<td>DMA 14 interrupt (SPI1)</td>
<td>39</td>
<td>IVG10</td>
<td>3</td>
</tr>
<tr>
<td>DMA 15 interrupt (SPI2)</td>
<td>40</td>
<td>IVG10</td>
<td>3</td>
</tr>
<tr>
<td>DMA 16 interrupt (UART1 RX)</td>
<td>41</td>
<td>IVG10</td>
<td>3</td>
</tr>
<tr>
<td>DMA 17 interrupt (UART1 TX)</td>
<td>42</td>
<td>IVG10</td>
<td>3</td>
</tr>
<tr>
<td>DMA 18 interrupt (UART2 RX)</td>
<td>43</td>
<td>IVG10</td>
<td>3</td>
</tr>
<tr>
<td>DMA 19 interrupt (UART2 TX)</td>
<td>44</td>
<td>IVG10</td>
<td>3</td>
</tr>
<tr>
<td>TWI0 interrupt</td>
<td>45</td>
<td>IVG11</td>
<td>4</td>
</tr>
</tbody>
</table>
The peripheral interrupt structure of the processor is flexible. By default upon reset, multiple peripheral interrupts share a single, general-purpose interrupt in the core, as shown in Table 4-7.

An interrupt service routine that supports multiple interrupt sources must interrogate the appropriate system MMRs to determine which peripheral generated the interrupt.

If the default assignments shown in Table 4-7 are acceptable, then interrupt initialization involves only initialization of the core EVT vector address entries and IMASK register, and unmasking the specific peripheral interrupts in SIC_IMASKx that the system requires.

<table>
<thead>
<tr>
<th>Peripheral Interrupt Source</th>
<th>Peripheral Interrupt ID</th>
<th>General-Purpose Interrupt (Assignment at Reset)</th>
<th>Core Interrupt ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>TWI1 interrupt</td>
<td>46</td>
<td>IVG11</td>
<td>4</td>
</tr>
<tr>
<td>CAN receive interrupt</td>
<td>47</td>
<td>IVG11</td>
<td>4</td>
</tr>
<tr>
<td>CAN transmit interrupt</td>
<td>48</td>
<td>IVG11</td>
<td>4</td>
</tr>
<tr>
<td>MDMA1 stream 0 (memory DMA)</td>
<td>49</td>
<td>IVG13</td>
<td>6</td>
</tr>
<tr>
<td>MDMA1 stream 1 (memory DMA)</td>
<td>50</td>
<td>IVG13</td>
<td>6</td>
</tr>
<tr>
<td>Reserved</td>
<td>1-63</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
System Interrupt Wake-Up Enable (SIC_IWRx) Registers

The SICs provide the mapping between the peripheral interrupt source and the dynamic power management controller (DPMC). Any of the peripherals can be configured to wake up the core from its idled state to process the interrupt, simply by enabling the appropriate bit in the system interrupt wake-up enable register (shown in Figure 4-6 and Figure 4-7). If a peripheral interrupt source is enabled in SIC_IWRx and the core is idled, the interrupt causes the DPMC to initiate the core wake-up sequence in order to process the interrupt. Note this mode of operation may add latency to interrupt processing, depending on the power control state. For more information, see “Dynamic Power Management” on page 8-1.

By default, all interrupts generate a wake-up request to the core. However, for some applications it may be desirable to disable this function for some peripherals, such as for a SPORTx transmit interrupt.

The SIC_IWRx registers have no effect unless the core is idled. The bits in these registers correspond to those of the system interrupt mask (SIC_IMASKx) and interrupt status (SIC_ISRx) registers.

After reset, all valid bits of the SIC_IWRx register are set to 1, enabling the wake-up function for all interrupts that are not masked. Before enabling interrupts, configure this register in the reset initialization sequence. The SIC_IWRx registers can be read from or written to at any time. To prevent spurious or lost interrupt activity, these registers should be written only when all peripheral interrupts are disabled.

Note the wake-up function is independent of the interrupt mask function. If an interrupt source is enabled in SIC_IWRx but masked off in SIC_IMASKx, the core wakes up if it is idled, but it does not generate an interrupt.
Events and Sequencing

System Interrupt Wake-Up Enable Register 0 (SIC_IWR0)
For all bits, 0 - Wake-up function not enabled, 1 - Wake-up function enabled.

![Diagram of System Interrupt Wake-Up Enable Register 0 (SIC_IWR0)](image)

Figure 4-6. System Interrupt Wake-Up Enable Register0 (SIC_IWR0)
System Interrupt Status (SIC_ISRx) Registers

The SICx include a read-only status register, the system interrupt status register, (shown in Figure 4-8 and Figure 4-9). Each valid bit in this register corresponds to one of the peripheral interrupt sources. The bit is set when SICx detects the interrupt is asserted and cleared when SICx detects that the peripheral interrupt input has been deasserted. Note for some peripherals, such as GPIO asynchronous input interrupts, many cycles of latency may pass from the time that an interrupt service routine initiates the clearing of the interrupt (usually by writing a system MMR) to the time that SICx senses that the interrupt has been deasserted.

Depending on how interrupt sources map to the general-purpose interrupt inputs of the core, the interrupt service routine may have to interrogate multiple interrupt status bits to determine the source of the interrupt.
One of the first instructions executed in an interrupt service routine should read `SIC_ISRx` to determine whether more than one of the peripherals sharing the input has asserted its interrupt output. The service routine should fully process all pending, shared interrupts before executing the RTI, which enables further interrupt generation on that interrupt input.

When an interrupt service routine is finished, the RTI instruction clears the appropriate bit in the `IPEND` register. However, the relevant `SIC_ISRx` bit is not cleared unless the service routine clears the mechanism that generated the interrupt.

Many systems need relatively few interrupt-enabled peripherals, allowing each peripheral to map to a unique core priority level. In these designs, `SIC_ISRx` will seldom, if ever, need to be interrogated.

The `SIC_ISRx` registers are not affected by the state of the system interrupt mask register (`SIC_IMASKx`) and can be read at any time. Writes to `SIC_ISRx` have no effect on its contents.
**System Interrupt Status Register 0 (SIC_ISR0)**

For all bits, 0 - Deasserted, 1 - Asserted.

<table>
<thead>
<tr>
<th>Bit 31</th>
<th>Bit 30</th>
<th>Bit 29</th>
<th>Bit 28</th>
<th>Bit 27</th>
<th>Bit 26</th>
<th>Bit 25</th>
<th>Bit 24</th>
<th>Bit 23</th>
<th>Bit 22</th>
<th>Bit 21</th>
<th>Bit 20</th>
<th>Bit 19</th>
<th>Bit 18</th>
<th>Bit 17</th>
<th>Bit 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>UART2 Error interrupt</td>
<td>UART1 Error interrupt</td>
<td>SPI2 Error interrupt</td>
<td>SPI1 Error interrupt</td>
<td>SPORT3 Error interrupt</td>
<td>SPORT2 Error interrupt</td>
<td>DMA Controller 1 Error (generic) interrupt</td>
<td>\</td>
<td>\</td>
<td>\</td>
<td>\</td>
<td>\</td>
<td>\</td>
<td>\</td>
<td>\</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 31</th>
<th>Bit 30</th>
<th>Bit 29</th>
<th>Bit 28</th>
<th>Bit 27</th>
<th>Bit 26</th>
<th>Bit 25</th>
<th>Bit 24</th>
<th>Bit 23</th>
<th>Bit 22</th>
<th>Bit 21</th>
<th>Bit 20</th>
<th>Bit 19</th>
<th>Bit 18</th>
<th>Bit 17</th>
<th>Bit 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA7 interrupt (UART0 TX)</td>
<td>DMA6 interrupt (UART0 RX)</td>
<td>DMA5 interrupt (SPI0)</td>
<td>DMA4 interrupt (SPORT1 TX)</td>
<td>DMA3 interrupt (SPORT1 RX)</td>
<td>DMA2 interrupt (SPORT0 TX)</td>
<td>DMA1 interrupt (SPORT0 RX)</td>
<td>DMA0 interrupt (PPI)</td>
<td>PLL Wakeup interrupt</td>
<td>DMA Controller 0 Error (generic) interrupt</td>
<td>GPIO interrupt A</td>
<td>GPIO interrupt B</td>
<td>Memory DMA0 Stream 0 interrupt</td>
<td>Memory DMA0 Stream 1 interrupt</td>
<td>Software Watchdog Timer interrupt</td>
<td>\</td>
</tr>
</tbody>
</table>

**Figure 4-8. System Interrupt Status Register 0 (SIC_ISR0)**
System Interrupt Status Register 1 (SIC_ISR1)
For all bits, 0 = deasserted, 1 = asserted.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Reset = 0x0000 0000

Memory DMA1 Stream 1 interrupt
CAN Transmit interrupt
Memory DMA1 Stream 0 interrupt
CAN Receive interrupt
TWI1 interrupt
TWI0 interrupt
DMA19 interrupt (UART2 TX)
DMA18 interrupt (UART2 RX)
DMA17 interrupt (UART1 TX)
DMA16 interrupt (UART1 RX)
DMA15 interrupt (SPI2)

Figure 4-9. System Interrupt Status Register 1 (SIC_ISR1)

System Interrupt Mask (SIC_IMASKx) Registers

The system interrupt mask registers (shown in Figure 4-10 and Figure 4-11) allow masking of any peripheral interrupt source in the SICx, independently of whether it is enabled at the peripheral itself.

A reset forces the contents of SIC_IMASKx to all 0s to mask off all peripheral interrupts. Writing a 1 to a bit location turns off the mask and enables the interrupt.

Although this register can be read from or written to at any time (in supervisor mode), it should be configured in the reset initialization sequence before enabling interrupts.
System Interrupt Mask Register 0 (SIC_IMASK0)
For all bits, 0 - interrupt masked, 1 - interrupt enabled.

Reset = 0x00000000

Figure 4-10. System Interrupt Mask Register 0 (SIC_IMASK0)
Events and Sequencing

System Interrupt Mask Register 1 (SIC_IMASK1)
For all bits, 0 - interrupt masked, 1 - interrupt enabled.

![System Interrupt Mask Register 1 (SIC_IMASK1)](image)

Reset = 0x0000 0000

System Interrupt Assignment (SIC_IARx) Registers

The relative priority of peripheral interrupts can be set by mapping the peripheral interrupt to the appropriate general-purpose interrupt level in the core. The mapping is controlled by the system interrupt assignment register settings, as detailed in Figure 4-12 through Figure 4-18. If more than one interrupt source is mapped to the same interrupt, they are logically OR’ed, with no hardware prioritization. Software can prioritize the interrupt processing as required for a particular system application.

For general-purpose interrupts with multiple peripheral interrupts assigned to them, take special care to ensure that software correctly processes all pending interrupts sharing that input. Software is responsible for prioritizing the shared interrupts.
System Interrupt Assignment Register 0 (SIC_IAR0)

Figure 4-12. System Interrupt Assignment Register 0 (SIC_IAR0)

System Interrupt Assignment Register 1 (SIC_IAR1)

Figure 4-13. System Interrupt Assignment Register 1 (SIC_IAR1)
System Interrupt Assignment Register 2 (SIC_IAR2)

![Diagram of SIC_IAR2](image)

Reset = 0x6665 5444

System Interrupt Assignment Register 3 (SIC_IAR3)

![Diagram of SIC_IAR3](image)

Reset = 0x0000 0000

Figure 4-14. System Interrupt Assignment Register 2 (SIC_IAR2)

Figure 4-15. System Interrupt Assignment Register 3 (SIC_IAR3)
Program Sequencer

Figure 4-16. System Interrupt Assignment Register 4 (SIC_IAR4)

Figure 4-17. System Interrupt Assignment Register 5 (SIC_IAR5)
Events and Sequencing

System Interrupt Assignment Register 6 (SIC_IAR6)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Reset = 0x0044 4664

Figure 4-18. System Interrupt Assignment Register 6 (SIC_IAR6)

These registers can be read or written at any time in supervisor mode. It is advisable, however, to configure them in the reset interrupt service routine before enabling interrupts. To prevent spurious or lost interrupt activity, these registers should be written to only when all peripheral interrupts are disabled.

Table 4-8 defines the value to write in SIC_IARx to configure a peripheral for a particular IVG priority.

Table 4-8. IVG-Select Definitions

<table>
<thead>
<tr>
<th>General-Purpose Interrupt</th>
<th>Value in SIC_IARx</th>
</tr>
</thead>
<tbody>
<tr>
<td>IVG7</td>
<td>0</td>
</tr>
<tr>
<td>IVG8</td>
<td>1</td>
</tr>
<tr>
<td>IVG9</td>
<td>2</td>
</tr>
<tr>
<td>IVG10</td>
<td>3</td>
</tr>
<tr>
<td>IVG11</td>
<td>4</td>
</tr>
<tr>
<td>IVG12</td>
<td>5</td>
</tr>
<tr>
<td>IVG13</td>
<td>6</td>
</tr>
<tr>
<td>IVG14</td>
<td>7</td>
</tr>
<tr>
<td>IVG15</td>
<td>8</td>
</tr>
</tbody>
</table>
Core Event Controller Registers

The event controller uses three MMRs to coordinate pending event requests. In each of these MMRs, the 16 lower bits correspond to the 16 event levels (for example, bit 0 corresponds to “emulator mode”). The registers are:

- **IMASK** - interrupt mask
- **ILAT** - interrupt latch
- **IPEND** - interrupts pending

These three registers are accessible in supervisor mode only.

Core Interrupt Mask (IMASK) Register

This register indicates which interrupt levels are allowed to be taken (see Figure 4-19). The **IMASK** register may be read and written in supervisor mode. Bits [15:5] have significance; bits [4:0] are hard-coded to 1 and events of these levels are always enabled. If **IMASK[N]** = 1 and **ILAT[N]** = 1, then interrupt **N** will be taken if a higher priority is not already recognized. If **IMASK[N]** = 0, and **ILAT[N]** gets set by interrupt **N**, the interrupt will not be taken, and **ILAT[N]** will remain set.
Core Event Controller Registers

Core Interrupt Mask Register (IMASK)
For all bits, 0 - interrupt masked, 1 - interrupt enabled.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```
Reset = 0x0000 001F

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1
```

IVG15
IVG14
IVG13
IVG12
IVG11
IVG10
IVHW (Hardware Error)
IVTMR (Core Timer)
IVG7
IVG8
IVG9

Figure 4-19. Core Interrupt Mask Register

Core Interrupt Latch (ILAT) Register

Each bit in ILAT indicates that the corresponding event is latched, but not yet accepted into the processor (see Figure 4-20). The bit is reset before the first instruction in the corresponding ISR is executed. At the point the interrupt is accepted, ILAT[N] is cleared and IPEND[N] is set simultaneously. The ILAT register can be read in supervisor mode. Writes to ILAT are used to clear bits only (in supervisor mode). To clear bit N from ILAT, first make sure that IMASK[N]==0, and then write ILAT[N]=1. This write functionality to ILAT is provided for cases where latched interrupt requests need to be cleared (cancelled) instead of servicing them.

The RAISE instruction can be used to set ILAT[15] through ILAT[5], and also ILAT[2] or ILAT[1].

Only the JTAG TRST pin can clear ILAT[0].
Core Interrupt Latch Register (ILAT)
Reset value for bit 0 is emulator-dependent. For all bits, 0 - interrupt not latched, 1 - interrupt latched.

![Core Interrupt Latch Register Diagram]

Figure 4-20. Core Interrupt Latch Register

Core Interrupts Pending (IPEND) Register

The IPEND register keeps track of all currently nested interrupts (see Figure 4-23). Each bit in IPEND indicates that the corresponding interrupt is currently active or nested at some level. It may be read in supervisor mode, but not written. The IPEND[4] bit is used by the event controller to temporarily disable interrupts on entry and exit to an interrupt service routine.

When an event is processed, the corresponding bit in IPEND is set. The least significant bit in IPEND that is currently set indicates the interrupt that is currently being serviced. At any given time, IPEND holds the current status of all nested events.
General-purpose interrupts can be globally disabled with the \texttt{CLI Dreg} instruction and re-enabled with the \texttt{STI Dreg} instruction, both of which are only available in supervisor mode. Reset, NMI, emulation, and exception events cannot be globally disabled. Globally disabling interrupts clears \texttt{IMASK[15:5]} after saving \texttt{IMASK}’s current state. See “Enable Interrupts” and “Disable Interrupts” in the “External Event Management” chapter in \textit{Blackfin Processor Programming Reference}.

When program code is too time critical to be delayed by an interrupt, disable general-purpose interrupts, but be sure to re-enable them at the conclusion of the code sequence.
Event Vector Table

The event vector table (EVT), shown in Table 4-9, is a hardware table with sixteen entries that are each 32 bits wide. The EVT contains an entry for each possible core event. Entries are accessed as MMRs, and each entry can be programmed at reset with the corresponding vector address for the interrupt service routine. When an event occurs, instruction fetch starts at the address location in the EVT entry for that event.

The processor architecture allows unique addresses to be programmed into each of the interrupt vectors; that is, interrupt vectors are not determined by a fixed offset from an interrupt vector table base address. This approach minimizes latency by not requiring a long jump from the vector table to the actual ISR code.

Table 4-9 lists events by priority. Each event has a corresponding bit in the event state registers ILAT, IMASK, and IPEND.

Table 4-9. Core Event Vector Table

<table>
<thead>
<tr>
<th>Event Number</th>
<th>Event Class</th>
<th>Name</th>
<th>MMR Location</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVT0</td>
<td>Emulation</td>
<td>EMU</td>
<td>0xFFE0 2000</td>
<td>Highest priority. Vector address is provided by JTAG.</td>
</tr>
<tr>
<td>EVT1</td>
<td>Reset</td>
<td>RST</td>
<td>0xFFE0 2004</td>
<td></td>
</tr>
<tr>
<td>EVT2</td>
<td>NMI</td>
<td>NMI</td>
<td>0xFFE0 2008</td>
<td></td>
</tr>
<tr>
<td>EVT3</td>
<td>Exception</td>
<td>EVX</td>
<td>0xFFE0 200C</td>
<td></td>
</tr>
<tr>
<td>EVT4</td>
<td>Reserved</td>
<td>Reserved</td>
<td>0xFFE0 2010</td>
<td>Reserved vector.</td>
</tr>
<tr>
<td>EVT5</td>
<td>Hardware error</td>
<td>IVHW</td>
<td>0xFFE0 2014</td>
<td></td>
</tr>
<tr>
<td>EVT6</td>
<td>Core timer</td>
<td>IVTMR</td>
<td>0xFFE0 2018</td>
<td></td>
</tr>
<tr>
<td>EVT7</td>
<td>Interrupt 7</td>
<td>IVG7</td>
<td>0xFFE0 201C</td>
<td></td>
</tr>
<tr>
<td>EVT8</td>
<td>Interrupt 8</td>
<td>IVG8</td>
<td>0xFFE0 2020</td>
<td></td>
</tr>
<tr>
<td>EVT9</td>
<td>Interrupt 9</td>
<td>IVG9</td>
<td>0xFFE0 2024</td>
<td></td>
</tr>
</tbody>
</table>
**Event Vector Table**

Table 4-9. Core Event Vector Table (Cont’d)

<table>
<thead>
<tr>
<th>Event Number</th>
<th>Event Class</th>
<th>Name</th>
<th>MMR Location</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVT10</td>
<td>Interrupt 10</td>
<td>IVG10</td>
<td>0xFFE0 2028</td>
<td></td>
</tr>
<tr>
<td>EVT11</td>
<td>Interrupt 11</td>
<td>IVG11</td>
<td>0xFFE0 202C</td>
<td></td>
</tr>
<tr>
<td>EVT12</td>
<td>Interrupt 12</td>
<td>IVG12</td>
<td>0xFFE0 2030</td>
<td></td>
</tr>
<tr>
<td>EVT13</td>
<td>Interrupt 13</td>
<td>IVG13</td>
<td>0xFFE0 2034</td>
<td></td>
</tr>
<tr>
<td>EVT14</td>
<td>Interrupt 14</td>
<td>IVG14</td>
<td>0xFFE0 2038</td>
<td></td>
</tr>
<tr>
<td>EVT15</td>
<td>Interrupt 15</td>
<td>IVG15</td>
<td>0xFFE0 203C</td>
<td>Lowest priority.</td>
</tr>
</tbody>
</table>

**Emulation**

An emulation event causes the processor to enter emulation mode, in which instructions are read from the JTAG interface. It is the highest priority interrupt to the core.

For detailed information about emulation, see “Blackfin Processor Debug” on page 22-1.

**Reset**

The reset interrupt (RST) can be initiated via the RESET pin or through expiration of the watchdog timer. This location differs from that of other interrupts in that its content is read-only. Writes to this address change the register but do not change where the processor vectors upon reset. The processor always vectors to the reset vector address upon reset. For more information, see “Reset State” on page 3-11 and “Booting Methods” on page 3-19.

The core has an output that indicates that a double-fault has occurred. This is a non-recoverable state. The system (via the SWRST register) can be programmed to send a reset request if a double-fault condition is detected. Subsequently, the reset request forces a system reset for core and peripherals.
The reset vector is determined by the processor system. It points to the start of the on-chip boot ROM, or to the start of external asynchronous memory, depending on the state of the BMODE[1:0] pins. Refer to Table 2-10 on page 2-41.

Table 4-10. Reset Vector Addresses

<table>
<thead>
<tr>
<th>Boot Source</th>
<th>BMODE[1:0]</th>
<th>Execution Start Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute from 16-bit external memory</td>
<td>00</td>
<td>0x2000 0000</td>
</tr>
<tr>
<td>Boot from 8-bit or 16-bit external flash memory</td>
<td>01</td>
<td>0xEF00 0000</td>
</tr>
<tr>
<td>Boot from an SPI host in SPI slave mode</td>
<td>10</td>
<td>0xEF00 0000</td>
</tr>
<tr>
<td>Boot from an 8-/16-/24-bit addressable SPI in SPI master mode</td>
<td>11</td>
<td>0xEF00 0000</td>
</tr>
</tbody>
</table>

If the BMODE[1:0] pins indicate either booting from flash or serial ROM, the reset vector points to the start of the internal boot ROM, where a small bootstrap kernel resides. The bootstrap code reads the system reset configuration register (SYSCR) to determine the value of the BMODE[1:0] pins, which determine the appropriate boot sequence. For information about the boot ROM, see “Booting Methods” on page 3-19.

If the BMODE[1:0] pins indicate to bypass boot ROM, the reset vector points to the start of the external asynchronous memory region. In this mode, the internal boot ROM is not used. To support reads from this memory region, the external bus interface unit (EBIU) uses the default external memory configuration that results from hardware reset.

**NMI (Non-Maskable Interrupt)**

The NMI entry is reserved for a non-maskable interrupt, which can be generated by the watchdog timer or by the NMI input signal to the processor. An example of an event that requires immediate processor attention, and thus is appropriate as an NMI, is a power-down warning.
Exceptions

Exceptions are synchronous to the instruction stream. In other words, a particular instruction causes an exception when it attempts to finish execution. No instructions after the offending instruction are executed before the exception handler takes effect.

Many of the exceptions are memory related. For example, an exception is given when a misaligned access is attempted, or when a CPLB miss or protection violation occurs. Exceptions are also given when illegal instructions or illegal combinations of registers are executed.

An excepting instruction may or may not commit before the exception event is taken, depending on if it is a “service” type or an “error” type exception.

An instruction causing a service type event will commit, and the address written to the RETX register will be the next instruction after the excepting one. An example of a service type exception is the single-step.

An instruction causing an error type event cannot commit, so the address written to the RETX register will be the address of the offending instruction. An example of an error type event is a CPLB miss.

Usually the RETX register contains the correct address to return to. To skip over an excepting instruction, take care in case the “next” address is not simply the next linear address. This could happen when the excepting instruction is a loop end. In that case the proper “next” address would be the loop top.
The \texttt{EXCAUSE[5:0]} field in the sequencer status register (\texttt{SEQSTAT}) is written whenever an exception is taken, and indicates to the exception handler which type of exception occurred. Refer to Table 4-11 for a list of events that cause exceptions.

Any exception in any event handler of exception level or above (including \texttt{NMI}) triggers a “double-fault” condition, and the address of the excepting instruction will be written to \texttt{RETX}.

Table 4-11. Events That Cause Exceptions

<table>
<thead>
<tr>
<th>Exception</th>
<th>EXCAUSE [5:0]</th>
<th>Type: (E) error (S) service $^1$</th>
<th>Notes/Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Force Exception instruction \texttt{EXCPT} with 4-bit field m</td>
<td>m-field</td>
<td>S</td>
<td>Instruction provides 4 bits of \texttt{EXCAUSE}.</td>
</tr>
<tr>
<td>Single step</td>
<td>0x10</td>
<td>S</td>
<td>When the processor is in single-step mode, every instruction generates an exception. Primarily used for debugging.</td>
</tr>
<tr>
<td>Exception caused by an emulation trace buffer overflow</td>
<td>0x11</td>
<td>S</td>
<td>The processor takes this exception when the trace buffer overflows (only when enabled by the trace unit control register).</td>
</tr>
<tr>
<td>Undefined instruction</td>
<td>0x21</td>
<td>E</td>
<td>May be used to emulate instructions that are not defined for a particular processor implementation.</td>
</tr>
<tr>
<td>Illegal instruction combination</td>
<td>0x22</td>
<td>E</td>
<td>See section for multi-issue rules in the \texttt{Blackfin Processor Programming Reference}.</td>
</tr>
</tbody>
</table>
### Table 4-11. Events That Cause Exceptions (Cont’d)

<table>
<thead>
<tr>
<th>Exception</th>
<th>EXCAUSE [5:0]</th>
<th>Type: (E) error (S) service</th>
<th>Notes/Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data access CPLB protection violation</td>
<td>0x23</td>
<td>E</td>
<td>Attempted read or write to supervisor resource, or illegal data memory access. Supervisor resources are registers and instructions that are reserved for supervisor use: supervisor only registers, all MMRs, and supervisor only instructions. (A simultaneous, dual access to two MMRs using the data address generators generates this type of exception.) In addition, this entry is used to signal a protection violation caused by disallowed memory access, and it is defined by the memory management unit (MMU) cacheability protection lookaside buffer (CPLB).</td>
</tr>
<tr>
<td>Data access misaligned address violation</td>
<td>0x24</td>
<td>E</td>
<td>Attempted misaligned data memory or data cache access.</td>
</tr>
<tr>
<td>Unrecoverable event</td>
<td>0x25</td>
<td>E</td>
<td>For example, an exception generated while processing a previous exception.</td>
</tr>
<tr>
<td>Data access CPLB miss</td>
<td>0x26</td>
<td>E</td>
<td>Used by the MMU to signal a CPLB miss on a data access.</td>
</tr>
<tr>
<td>Data access multiple CPLB hits</td>
<td>0x27</td>
<td>E</td>
<td>More than one CPLB entry matches data fetch address.</td>
</tr>
<tr>
<td>Exception caused by an emulation watchpoint match</td>
<td>0x28</td>
<td>E</td>
<td>There is a watchpoint match, and one of the EMUSW bits in the watchpoint instruction address control register (WPIACTL) is set.</td>
</tr>
</tbody>
</table>
### Table 4-11. Events That Cause Exceptions (Cont’d)

<table>
<thead>
<tr>
<th>Exception</th>
<th>EXCAUSE [5:0]</th>
<th>Type: (E) error (S) service</th>
<th>Notes/Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction fetch misaligned address violation</td>
<td>0x2A</td>
<td>E</td>
<td>Attempted misaligned instruction cache fetch. On a misaligned instruction fetch exception, the return address provided in RETX is the destination address which is misaligned, rather than the address of the offending instruction. For example, if an indirect branch to a misaligned address held in P0 is attempted, the return address in RETX is equal to P0, rather than to the address of the branch instruction. (Note this exception can never be generated from PC-relative branches, only from indirect branches.)</td>
</tr>
<tr>
<td>Instruction fetch CPLB protection violation</td>
<td>0x2B</td>
<td>E</td>
<td>Illegal instruction fetch access (memory protection violation).</td>
</tr>
<tr>
<td>Instruction fetch CPLB miss</td>
<td>0x2C</td>
<td>E</td>
<td>CPLB miss on an instruction fetch.</td>
</tr>
<tr>
<td>Instruction fetch multiple CPLB hits</td>
<td>0x2D</td>
<td>E</td>
<td>More than one CPLB entry matches instruction fetch address.</td>
</tr>
<tr>
<td>Illegal use of supervisor resource</td>
<td>0x2E</td>
<td>E</td>
<td>Attempted to use a supervisor register or instruction from user mode. Supervisor resources are registers and instructions that are reserved for supervisor use: supervisor only registers, all MMRs, and supervisor only instructions.</td>
</tr>
</tbody>
</table>

1 For services (S), the return address is the address of the instruction that follows the exception. For errors (E), the return address is the address of the excepting instruction.
If an instruction causes multiple exceptions, only the exception with the highest priority is taken. Table 4-12 ranks exceptions by descending priority.

Table 4-12. Exceptions by Descending Priority

<table>
<thead>
<tr>
<th>Priority</th>
<th>Exception</th>
<th>EXCAUSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Unrecoverable event</td>
<td>0x25</td>
</tr>
<tr>
<td>2</td>
<td>I-Fetch multiple CPLB hits</td>
<td>0x2D</td>
</tr>
<tr>
<td>3</td>
<td>I-Fetch misaligned access</td>
<td>0x2A</td>
</tr>
<tr>
<td>4</td>
<td>I-Fetch protection violation</td>
<td>0x2B</td>
</tr>
<tr>
<td>5</td>
<td>I-Fetch CPLB miss</td>
<td>0x2C</td>
</tr>
<tr>
<td>6</td>
<td>I-Fetch access exception</td>
<td>0x29</td>
</tr>
<tr>
<td>7</td>
<td>Watchpoint match</td>
<td>0x28</td>
</tr>
<tr>
<td>8</td>
<td>Undefined instruction</td>
<td>0x21</td>
</tr>
<tr>
<td>9</td>
<td>Illegal combination</td>
<td>0x22</td>
</tr>
<tr>
<td>10</td>
<td>Illegal use protected resource</td>
<td>0x2E</td>
</tr>
<tr>
<td>11</td>
<td>DAG0 multiple CPLB hits</td>
<td>0x27</td>
</tr>
<tr>
<td>12</td>
<td>DAG0 misaligned access</td>
<td>0x24</td>
</tr>
<tr>
<td>13</td>
<td>DAG0 protection violation</td>
<td>0x23</td>
</tr>
<tr>
<td>14</td>
<td>DAG0 CPLB miss</td>
<td>0x26</td>
</tr>
<tr>
<td>15</td>
<td>DAG1 multiple CPLB hits</td>
<td>0x27</td>
</tr>
<tr>
<td>16</td>
<td>DAG1 misaligned access</td>
<td>0x24</td>
</tr>
<tr>
<td>17</td>
<td>DAG1 protection violation</td>
<td>0x23</td>
</tr>
<tr>
<td>18</td>
<td>DAG1 CPLB miss</td>
<td>0x26</td>
</tr>
<tr>
<td>19</td>
<td>EXCPT instruction</td>
<td>m-field</td>
</tr>
<tr>
<td>20</td>
<td>Single step</td>
<td>0x10</td>
</tr>
<tr>
<td>21</td>
<td>Trace buffer</td>
<td>0x11</td>
</tr>
</tbody>
</table>
Exceptions While Executing an Exception Handler

While executing the exception handler, avoid issuing an instruction that generates another exception. If an exception is caused while executing code within the exception handler, the NMI handler, the reset vector, or in emulator mode:

- The excepting instruction is not committed. All write backs from the instruction are prevented.
- The generated exception is not taken.
- The `EXCAUSE` field in `SEQSTAT` is updated with an unrecoverable event code.
- The address of the offending instruction is saved in `RETX`. Note if the processor were executing, for example, the NMI handler, the `RETN` register would not have been updated; the excepting instruction address is always stored in `RETX`.

To determine whether an exception occurred while an exception handler was executing, check `SEQSTAT` at the end of the exception handler for the code indicating an “unrecoverable event” (`EXCAUSE = 0x25`). If an unrecoverable event occurred, register `RETX` holds the address of the most recent instruction to cause an exception. This mechanism is not intended for recovery, but rather for detection.
Hardware Error Interrupt

The hardware error interrupt indicates a hardware error or system malfunction. Hardware errors occur when logic external to the core, such as a memory bus controller, is unable to complete a data transfer (read or write) and asserts the core’s error input signal. Such hardware errors invoke the hardware error interrupt (interrupt IVHW in the event vector table (EVT) and ILAT, IMASK, and IPEND registers). The hardware error interrupt service routine can then read the cause of the error from the 5-bit HWERRCAUSE field appearing in the sequencer status register (SEQSTAT) and respond accordingly.

The hardware error interrupt is generated by:

- Bus parity errors
- Internal error conditions within the core, such as performance monitor overflow
- The DMA access bus comparator interrupt (attempted write to an active DMA register)
- Peripheral errors
- Bus timeout errors

The list of supported hardware conditions, with their related HWERRCAUSE codes, appears in Table 4-13. The bit code for the most recent error appears in the HWERRCAUSE field. If multiple hardware errors occur simultaneously, only the last one can be recognized and serviced. The core does not support prioritizing, pipelining, or queuing multiple error codes. The hardware error interrupt remains active as long as any of the error conditions remain active.
The core timer interrupt (IVTMR) is triggered when the core timer value reaches zero. See “Timers” on page 16-1.

### General-Purpose Interrupts (IVG7-IVG15)

General-purpose interrupts are used for any event that requires processor attention. For instance, a DMA controller may use them to signal the end of a data transmission, or a serial communications device may use them to signal transmission errors.

#### Table 4-13. Hardware Conditions Causing Hardware Error Interrupts

<table>
<thead>
<tr>
<th>Hardware Condition</th>
<th>HWERRCAUSE (Binary)</th>
<th>HWERRCAUSE (Hexadecimal)</th>
<th>Notes/Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>System MMR Error</td>
<td>0b00010</td>
<td>0x02</td>
<td>An error can occur if an invalid System MMR location is accessed, if a 32-bit register is accessed with a 16-bit instruction, or if a 16-bit register is accessed with a 32-bit instruction.</td>
</tr>
<tr>
<td>External Memory Addressing Error</td>
<td>0b00011</td>
<td>0x03</td>
<td>An access to reserved or uninitialized memory was attempted.</td>
</tr>
<tr>
<td>Performance Monitor Overflow</td>
<td>0b10010</td>
<td>0x12</td>
<td>Refer to “Performance Monitoring Unit” on page 22-19.</td>
</tr>
<tr>
<td>RAISE 5 instruction</td>
<td>0b11000</td>
<td>0x18</td>
<td>Software issued a RAISE 5 instruction to invoke the Hardware Error Interrupt (IVHW).</td>
</tr>
<tr>
<td>Reserved</td>
<td>All other bit combinations.</td>
<td>All other values.</td>
<td></td>
</tr>
</tbody>
</table>
Servicing Interrupts

Software can also trigger general-purpose interrupts by using the RAISE instruction. The RAISE instruction can force events for interrupts IVG15-IVG7, IVTMR, IVHW, NMI, and RST, but not for exceptions and emulation (EVX and EMU, respectively).

It is recommended to reserve the two lowest priority interrupts (IVG15 and IVG14) for software interrupt handlers.

Servicing Interrupts

The CEC has a single interrupt queueing element per event, as a bit in the ILAT register. The appropriate ILAT bit is set when an interrupt rising edge is detected (which takes 2 core clock cycles) and cleared when the respective IPEND register bit is set. The IPEND bit indicates that the event vector has entered the core pipeline. At this point, the CEC recognizes and queues the next rising edge event on the corresponding interrupt input. The minimum latency from the rising edge transition of the general-purpose interrupt to the IPEND output asserted is three core clock cycles. However, the latency can be much higher, depending on the core’s activity level and state.

To determine when to service an interrupt, the controller logically ANDs the three quantities in ILAT, IMASK, and the current processor priority level.

Servicing the highest priority interrupt involves the following actions:

1. The interrupt vector in the EVT becomes the next fetch address.

2. On an interrupt, most instructions currently in the pipeline are aborted. On a service exception, all instructions after the excepting instruction are aborted. On an error exception, the excepting instruction and all instructions after it are aborted.

3. The return address is saved in the appropriate return register.
4. The return register is **RETI** for interrupts, **RETX** for exceptions, **RETN** for NMIs, and **RETE** for debug emulation. The return address is the address of the instruction after the last-executed instruction from normal program flow.

5. Processor mode is set to the level of the event taken.

   If the event is an NMI, exception, or interrupt, the processor mode is supervisor. If the event is an emulation exception, the processor mode is emulation.

Before the first instruction starts execution, the corresponding interrupt bit in **ILAT** is cleared and the corresponding bit in **IPEND** is set.

Bit **IPEND[4]** is also set to disable all interrupts until the return address in **RETI** is saved.

## Nesting of Interrupts

Interrupts are handled either with or without nesting.

### Non-Nested Interrupts

If interrupts do not require nesting, all interrupts are disabled during the interrupt service routine. Note, however, that emulation, NMI, and exceptions are still accepted by the system.

When the system does not need to support nested interrupts, there is no need to store the return address held in **RETI**. Only the portion of the machine state used in the interrupt service routine must be saved in the supervisor stack. To return from a non-nested interrupt service routine, only the **RTI** instruction must be executed, because the return address is already held in the **RETI** register.
Figure 4-22 shows an example of interrupt handling where interrupts are globally disabled for the entire interrupt service routine.

Figure 4-22. Non-Nested Interrupt Handling
Nested Interrupts

If nested interrupts are desired, the return address to the interrupted point in the original interrupt service routine (ISR) must be explicitly saved and subsequently restored when execution of the nested ISR has completed. Nesting is enabled by pushing the return address currently held in RETI to the Supervisor stack (\([-\text{SP}] = \text{RETI}\)), which is typically done early in the ISR prolog of the lower priority interrupt. This clears the global interrupt disable bit \(\text{IPEND}[4]\), enabling interrupts. Next, all registers that are modified by the interrupt service routine are saved onto the supervisor stack. Processor state is stored in the supervisor stack, not in the user stack. Hence, the instructions to push \(\text{RETI} ([\text{-SP}] = \text{RETI})\) and pop \(\text{RETI} (\text{RETI} = [\text{SP}++] )\) use the supervisor stack.

Figure 4-23 illustrates that by pushing RETI onto the stack, interrupts can be re-enabled during an ISR, resulting in only a short duration where interrupts are globally disabled.
Figure 4-23. Nested Interrupt Handling
Example Prolog Code for Nested Interrupt Service Routine

/* Prolog code for nested interrupt service routine. Push return address in RETI into Supervisor stack, ensuring that interrupts are back on. Until now, interrupts have been suspended. */
ISR :
[-SP] = RETI ; /* Enables interrupts and saves return address to stack. */
[-SP] = ASTAT ;
[-SP] = FP ;
[-SP] = (R7:0, P5:0) ;
/* Body of service routine. Note none of the processor resources (accumulators, DAGs, loop counters and bounds) have been saved. It's assumed that this interrupt service routine does not use them. */

Example Epilog Code for Nested Interrupt Service Routine

/* Epilog code for nested-interrupt service routine. Restore ASTAT, Data and Pointer registers. Popping RETI from Supervisor stack ensures that interrupts are suspended between load of return address and RTI. */
(R7:0, P5:0 ) = [SP++] ;
FP = [SP++] ;
ASTAT = [SP++] ;
RETI = [SP++] ;
/* Execute RTI, which jumps to return address, re-enables interrupts, and switches to User mode if this is the last nested interrupt in service. */
RTI ;
Nesting of Interrupts

The RTI instruction causes the return from an interrupt. The return address is popped into the RETI register from the stack, an action that suspends interrupts from the time that RETI is restored until RTI finishes executing. The suspension of interrupts prevents a subsequent interrupt from corrupting the RETI register.

Next, the RTI instruction clears the highest priority bit that is currently set in IPEND. The processor then jumps to the address pointed to by the value in the RETI register and re-enables the interrupts by clearing IPEND[4].

Logging of Nested Interrupt Requests

The SICs detect level-sensitive interrupt requests from the peripherals. The CEC provides edge-sensitive detection for its general-purpose interrupts (IVG7-IVG15). Consequently, the SICs generate a synchronous interrupt pulse to the CEC and then wait for interrupt acknowledgement from the CEC. When the interrupt has been acknowledged by the core (via assertion of the appropriate IPEND output), the SICs generate another synchronous interrupt pulse to the CEC if the peripheral interrupt is still asserted. This way, the system does not lose peripheral interrupt requests that occur during servicing of another interrupt.

Because multiple interrupt sources can map to a single core processor general-purpose interrupt, multiple pulse assertions from the SICs can occur simultaneously, before, or during interrupt processing for an interrupt event that is already detected on this interrupt input. For a shared interrupt, the IPEND interrupt acknowledge mechanism described above re-enables all shared interrupts. If any of the shared interrupt sources are still asserted, at least one pulse is again generated by the SICs. The interrupt register registers indicate the current state of the shared interrupt sources.
Exception Handling

Interrupts and exceptions treat instructions in the pipeline differently:

- When an interrupt occurs, all instructions in the pipeline are aborted.
- When an exception occurs, all instructions in the pipeline after the excepting instruction are aborted. For error exceptions, the excepting instruction is also aborted.

Because exceptions, NMIs, and emulation events have a dedicated return register, guarding the return address is optional. Consequently, the push and pop instructions for exceptions, NMIs, and emulation events do not affect the interrupt system.

Note, however, the return instructions for exceptions (RTX, RTN, and RTE) do clear the least significant bit currently set in IPEND.

Deferring Exception Processing

Exception handlers are usually long routines, because they must discriminate among several exception causes and take corrective action accordingly. The length of the routines may result in long periods during which the interrupt system is, in effect, suspended.

To avoid lengthy suspension of interrupts, write the exception handler to identify the exception cause, but defer the processing to a low priority interrupt. To set up the low priority interrupt handler, use the Force interrupt/reset instruction (RAISE).

When deferring the processing of an exception to lower priority interrupt IVGx, the system must guarantee that IVGx is entered before returning to the application-level code that issued the exception. If a pending interrupt of higher priority than IVGx occurs, it is acceptable to enter the high priority interrupt before IVGx.
Exception Handling

Example Code for an Exception Handler

Listing 4-1 is for an exception routine handler with deferred processing.

Listing 4-1. Exception Routine Handler With Deferred Processing

/* Determine exception cause by examining EXCAUSE field in SEQSTAT (first save contents of R0, P0, P1 and ASTAT in Supervisor SP) */
[--SP] = R0 ;
[--SP] = P0 ;
[--SP] = P1 ;
[--SP] = ASTAT ;
R0 = SEQSTAT ;
/* Mask the contents of SEQSTAT, and leave only EXCAUSE in R0 */
R0 << = 26 ;
R0 >> = 26 ;
/* Using jump table EVTABLE, jump to the event pointed by R0 */
P0 = R0 ;
P1 = _EVTABLE ;
P0 = P1 + ( P0 << 1 ) ;
R0 = W[ P0 ] (Z) ;
P1 = R0 ;
JUMP ( PC + P1 ) ;
/* The entry point for an event is as follows. Here, processing is deferred to low-priority interrupt IVG15. Also, parameter-passing would typically be done here. */
_EVENT1 :
RAISE 15 ;
JUMP.S_EXIT ;
/* Entry for event at IVG14 */
_EVENT2 :
RAISE 14 ;
JUMP.S_EXIT ;
/* comments for other events. At the end of handler, restore R0, P0, P1 and ASTAT, and return. */

(EXIT):
ASTAT = [SP++];
P1 = [SP++];
P0 = [SP++];
R0 = [SP++];
RTI;

(EVTABLE):
.byte2 addr_event1;
.byte2 addr_event2;
...
.byte2 addr_eventN;

/* The jump table EVTABLE holds 16-bit address offsets for each event. With offsets, this code is position-independent and the table is small.
+-------------+   +-------------+    +-------------+     +-------------+
| addr_event1 |   | EVTABLE     |    | addr_event2 |   | EVTABLE + 2 |     | addr_eventN |   | EVTABLE + 2N|
|-------------|   |--------------|    |-------------|   |-------------|     |-------------|   |-------------|
*/
Example Code for an Exception Routine

Listing 4-2 provides an example framework for an exception routine that would be jumped to from an exception handler such as that described in Listing 4-1.

Listing 4-2. Exception Routine

`[--SP] = RETX ; /* Push return address on stack. No change to ILAT or IPEND. */`

`/* Put body of exception routine here. */`

`RETX = [SP++] ; /* To return, pop return address and jump. No change to ILAT or IPEND. */`

`RTX ; /* Return from exception. Clear IPEND[3] (Exception Pending bit). */`

Example Code for Using Hardware Loops in an ISR

Listing 4-3 provides shows the optimal method of saving and restoring when using hardware loops in an ISR.

Listing 4-3. Exception Routine

`1handler`

`<save other registers here>`

`[--SP] = LCO ; /* save loop 0 */`
`[--SP] = LBO ;`
`[--SP] = LTO ;`
<handler code here>

/* If the handler uses loop 0, it is a good idea to have it leave
LC0 equal to zero at the end. Normally, this will happen natu-
rally as a loop is fully executed. If LC0 == 0, then LTO and LBO
restores will not incur additional cycles. If LC0 != 0 when the
following pops happen, each pop will incur a 10-cycle “replay”
penalty. Popping or writing LC0 always incurs the penalty. */

LTO = [SP++] ;
LBO = [SP++] ;
LC0 = [SP++] ; /* This will cause a “replay.” That is, a
10-cycle refetch. */

<restore other registers here>

RTI ;

Other Usability Issues

The following sections describe other usability issues.

Executing RTX, RTN, or RTE in a Lower Priority Event

Instructions RTX, RTN, and RTE are designed to return from an exception,
NMI, or emulator event, respectively. Do not use them to return from a
lower-priority event. To return from an interrupt, use the RTI instruction.
Failure to use the correct instruction produces the following results.

If a program mistakenly uses RTX, RTN, or RTE to return from an interrupt,
the core branches to the address in the corresponding return register
(RETX, RETN, RETE). IPEND is modified correctly but the return address will
possibly be wrong.
Other Usability Issues

If a program mistakenly uses RTI or RTX to return from an NMI routine, the core branches to the address in the corresponding return register (RETI, RETX), and clears the bit in IPEND that corresponds to the return instruction.

In the case of RTX, bit IPEND[3] is cleared. In the case of RTI, the bit of the highest priority interrupt in IPEND is cleared.

Recommendation for Allocating the System Stack

The software stack model for processing exceptions implies that the supervisor stack must never generate an exception while the exception handler is saving its state. However, if the supervisor stack grows past a CPLB entry or SRAM block, it may, in fact, generate an exception.

To guarantee that the supervisor stack never generates an exception—never overflows past a CPLB entry or SRAM block while executing the exception handler—calculate the maximum space that all interrupt service routines and the exception handler occupy while they are active, and then allocate this amount of SRAM memory.

Latency in Servicing Events

In some processor architectures, if instructions are executed from external memory and an interrupt occurs while the instruction fetch operation is underway, then the interrupt is held off from being serviced until the current fetch operation has completed. Consider a processor operating at 300 MHz and executing code from external memory with 100 ns access times. Depending on when the interrupt occurs in the instruction fetch operation, the interrupt service routine (ISR) may be held off for around 30 instruction clock cycles. When cache line fill operations are taken into account, the ISR could be held off for many hundreds of cycles.
In order for high-priority interrupts to be serviced with the least latency possible, the processor allows any high latency fill operation to be completed at the system level, while an ISR executes from L1 memory. Figure 2-15 on page 2-48 illustrates this concept.

![Diagram of minimizing latency in servicing an ISR]

Figure 4-24. Minimizing Latency in Servicing an ISR

If an instruction load operation misses the L1 instruction cache and generates a high latency line fill operation, then when an interrupt occurs, it is not held off until the fill has completed. Instead, the processor executes the ISR in its new context, and the cache fill operation completes in the background.

Note the ISR must reside in L1 cache or SRAM memory and must not generate a cache miss, an L2 memory access, or a peripheral access, as the processor is already busy completing the original cache line fill operation.
If a load or store operation is executed in the ISR requiring one of these accesses, then the ISR is held off while the original external access is completed, before initiating the new load or store.

If the ISR finishes execution before the load operation has completed, then the processor continues to stall, waiting for the fill to complete.

This same behavior is also exhibited for stalls involving reads of slow data memory or peripherals.

Writes to slow memory generally do not show this behavior, as the writes are deemed to be single cycle, being immediately transferred to the write buffer for subsequent execution.

For detailed information about cache and memory structures, Chapter 6, “Memory”.
The data address generators (DAGs) generate addresses for data moves to and from memory. By generating addresses, the DAGs let programs refer to addresses indirectly, using a DAG register instead of an absolute address.

The DAG architecture, shown in Figure 5-1, supports several functions that minimize overhead in data access routines. These functions include:

- Supply address – Provides an address during a data access
- Supply address and post-modify – Provides an address during a data move and auto-increments/decrements the stored address for the next move
- Supply address with offset – Provides an address from a base with an offset without incrementing the original address pointer
- Modify address – Increments or decrements the stored address without performing a data move
- Bit-reversed carry address – Provides a bit-reversed carry address during a data move without reversing the stored address

The DAG subsystem comprises two DAG Arithmetic units, nine pointer registers, four index registers and four complete sets of related modify, base, and length registers. These registers hold the values that the DAGs use to generate addresses. The types of registers are:
- Index registers, \( \text{I}[3:0] \). Unsigned 32-bit index registers hold an address pointer to memory. For example, the instruction \( \text{R3} = \text{[I0]} \) loads the data value found at the memory location pointed to by the register \( \text{I0} \). Index registers can be used for 16- and 32-bit memory accesses.

- Modify registers, \( \text{M}[3:0] \). Signed 32-bit modify registers provide the increment or step size by which an index register is post-modified during a register move. For example, the \( \text{R0} = \text{[I0 ++ M1]} \) instruction directs the DAG to:
  - Output the address in register \( \text{I0} \)
  - Load the contents of the memory location pointed to by \( \text{I0} \) into \( \text{R0} \)
  - Modify the contents of \( \text{I0} \) by the value contained in the \( \text{M1} \) register

- Base and length registers, \( \text{B}[3:0] \) and \( \text{L}[3:0] \). Unsigned 32-bit base and length registers set up the range of addresses and the starting address of a circular buffer. Each \( \text{B} \), \( \text{L} \) pair is always coupled with a corresponding I-register, for example, \( \text{I3}, \text{B3}, \text{L3} \). For more information on circular buffers, see “Addressing Circular Buffers” on page 5-6.

- Pointer registers, \( \text{P}[5:0], \text{FP}, \text{USP}, \) and \( \text{SP} \). 32-bit pointer registers hold an address pointer to memory. The \( \text{P}[5:0] \) field, \( \text{FP} \) (frame pointer) and \( \text{SP}/\text{USP} \) (stack pointer/user stack pointer) can be manipulated and used in various instructions. For example, the instruction \( \text{R3} = \text{[P0]} \) loads the register \( \text{R3} \) with the data value found at the memory location pointed to by the register \( \text{P0} \). The pointer registers have no effect on circular buffer addressing. They can be used for 8-, 16-, and 32-bit memory accesses. For added mode protection, \( \text{SP} \) is accessible only in supervisor mode, while \( \text{USP} \) is accessible in user mode.
Data Address Generators

Do not assume the L-registers are automatically initialized to zero for linear addressing. The I-, M-, L-, and B-registers contain random values after reset. For each I-register used, programs must initialize the corresponding L-registers to zero for linear addressing or to the buffer length for circular buffer addressing.

Note all DAG registers must be initialized individually. Initializing a B-register does not automatically initialize the I-register.

Data Address Generator Registers (DAGs)

Supervisor only register. Attempted read or write in User mode causes an exception error.

Figure 5-1. Processor DAG Registers
Addressing With DAGs

The DAGs can generate an address that is incremented by a value or by a register. In post-modify addressing, the DAG outputs the I-register value unchanged; then the DAG adds an M-register or immediate value to the I-register.

In indexed addressing, the DAG adds a small offset to the value in the P-register, but does not update the P-register with this new value, thus providing an offset for that particular memory access.

The processor is byte addressed. All data accesses must be aligned to the data size. In other words, a 32-bit fetch must be aligned to 32 bits, but an 8-bit store can be aligned to any byte. Depending on the type of data used, increments and decrements to the DAG registers can be by 1, 2, or 4 to match the 8-, 16-, or 32-bit accesses.

For example, consider the following instruction:

\[ R0 = [ P3++] \];

This instruction fetches a 32-bit word, pointed to by the value in \( P3 \), and places it in \( R0 \). It then post-increments \( P3 \) by four, maintaining alignment with the 32-bit access.

\[ R0.L = W [ I3++] \];

This instruction fetches a 16-bit word, pointed to by the value in \( I3 \), and places it in the low half of the destination register, \( R0.L \). It then post-increments \( I3 \) by two, maintaining alignment with the 16-bit access.

\[ R0 = B [ P3++] (Z) \];

This instruction fetches an 8-bit word, pointed to by the value in \( P3 \), and places it in the destination register, \( R0 \). It then post-increments \( P3 \) by one, maintaining alignment with the 8-bit access. The byte value may be zero-extended (as shown) or sign-extended into the 32-bit data register.
Instructions using index registers use an M-register or a small immediate value (+/− 2 or 4) as the modifier. Instructions using pointer registers use a small immediate value or another P-register as the modifier. For details, see Table 5-3 on page 5-18.

Frame and Stack Pointers

In many respects, the frame and stack pointer registers perform like the other P-registers, \( P[5:0] \). They can act as general pointers in any of the load/store instructions, for example, \( R1 = B[SP\ (Z)] \). However, \( FP \) and \( SP \) have additional functionality.

The stack pointer registers include:

- a user stack pointer (\( USP \) in supervisor mode, \( SP \) in user mode)
- a supervisor stack pointer (\( SP \) in supervisor mode)

The user stack pointer register and the supervisor stack pointer register are accessed using the register alias \( SP \). Depending on the current processor operating mode, only one of these registers is active and accessible as \( SP \):

- In user mode, any reference to \( SP \) (for example, stack pop \( R0 = [SP++] ; \)) implicitly uses the \( USP \) as the effective address.
- In supervisor mode, the same reference to \( SP \) (for example, \( R0 = [SP++] ; \)) implicitly uses the supervisor stack pointer as the effective address.

To manipulate the user stack pointer for code running in supervisor mode, use the register alias \( USP \). When in supervisor mode, a register move from \( USP \) (for example, \( R0 = USP ; \)) moves the current user stack pointer into \( R0 \). The register alias \( USP \) can only be used in supervisor mode.
Addressing With DAGs

Some load/store instructions use $FP$ and $SP$ implicitly:

- $FP$-indexed load/store, which extends the addressing range for 16-bit encoded load/stores
- Stack push/pop instructions, including those for pushing and popping multiple registers
- Link/unlink instructions, which control stack frame space and manage the frame pointer register ($FP$) for that space

Addressing Circular Buffers

The DAGs support addressing circular buffers. Circular buffers are a range of addresses containing data that the DAG steps through repeatedly, wrapping around to repeat stepping through the same range of addresses in a circular pattern.

The DAGs use four types of DAG registers for addressing circular buffers. For circular buffering, the registers operate this way:

- The index (I) register contains the value that the DAG outputs on the address bus.
- The modify (M) register contains the post-modify amount (positive or negative) that the DAG adds to the I-register at the end of each memory access.

Any M-register can be used with any I-register. The modify value can also be an immediate value instead of an M-register. The size of the modify value must be less than or equal to the length (L-register) of the circular buffer.
Data Address Generators

- The Length (L) register sets the size of the circular buffer and the address range through which the DAG circulates the I-register.

  L is positive and cannot have a value greater than $2^{32} - 1$. If an L-register’s value is zero, its circular buffer operation is disabled.

- The base (B) register or the B-register plus the L-register is the value with which the DAG compares the modified I-register value after each access.

To address a circular buffer, the DAG steps the index pointer (I-register) through the buffer values, post-modifying and updating the index on each access with a positive or negative modify value from the M-register.

If the index pointer falls outside the buffer range, the DAG subtracts the length of the buffer (L-register) from the value or adds the length of the buffer to the value, wrapping the Index pointer back to a point inside the buffer.

The starting address that the DAG wraps around is called the buffer’s base address (B-register). There are no restrictions on the value of the base address for circular buffers that contains 8-bit data. Circular buffers that contain 16- and 32-bit data must be 16-bit aligned and 32-bit aligned, respectively. Exceptions can be made for video operations. For more information, see “Memory Address Alignment” on page 5-14. Circular buffering uses post-modify addressing.
As seen in Figure 5-2, on the first post-modify access to the buffer, the DAG outputs the I-register value on the address bus, then modifies the address by adding the modify value.

- If the updated index value is within the buffer length, the DAG writes the value to the I-register.
- If the updated index value exceeds the buffer length, the DAG subtracts (for a positive modify value) or adds (for a negative modify value) the L-register value before writing the updated index value to the I-register.

![Figure 5-2. Circular Data Buffers](image_url)
In equation form, these post-modify and wraparound operations work as follows, shown for “I+M” operations.

- If $M$ is positive:

  \[ I_{\text{new}} = I_{\text{old}} + M \]
  
  if $I_{\text{old}} + M < \text{buffer base} + \text{length}$ (end of buffer)

  \[ I_{\text{new}} = I_{\text{old}} + M - L \]
  
  if $I_{\text{old}} + M \geq \text{buffer base} + \text{length}$ (end of buffer)

- If $M$ is negative:

  \[ I_{\text{new}} = I_{\text{old}} + M \]
  
  if $I_{\text{old}} + M \geq \text{buffer base}$ (start of buffer)

  \[ I_{\text{new}} = I_{\text{old}} + M + L \]
  
  if $I_{\text{old}} + M < \text{buffer base}$ (start of buffer)

### Addressing With Bit-Reversed Addresses

To obtain results in sequential order, programs need bit-reversed carry addressing for some algorithms, particularly Fast Fourier Transform (FFT) calculations. To satisfy the requirements of these algorithms, the DAG’s bit-reversed addressing feature permits repeatedly subdividing data sequences and storing this data in bit-reversed order. For detailed information about bit-reversed addressing, see the Modify-Increment instruction in *Blackfin Processor Programming Reference*. 
Indexed Addressing With Index and Pointer Registers

Indexed addressing uses the value in the index or pointer register as an effective address. This instruction can load or store 16- or 32-bit values. The default is a 32-bit transfer. If a 16-bit transfer is required, then the \( W \) designator is used to preface the load or store.

For example:

\[
R0 = [I2];
\]

loads a 32-bit value from an address pointed to by \( I2 \) and stores it in the destination register \( R0 \).

\[
R0.H = W[I2];
\]

loads a 16-bit value from an address pointed to by \( I2 \) and stores it in the 16-bit destination register \( R0.H \).

\[
[P1] = R0;
\]

is an example of a 32-bit store operation.

Pointer registers can be used for 8-bit loads and stores.

For example:

\[
B[P1++] = R0;
\]

stores the 8-bit value from the \( R0 \) register in the address pointed to by the \( P1 \) register, then increments the \( P1 \) register.
Auto-Increment and Auto-Decrement Addressing

Auto-increment addressing updates the pointer and index registers after the access. The amount of increment depends on the word size. An access of 32-bit words results in an update of the pointer by 4. A 16-bit word access updates the pointer by 2, and an access of an 8-bit word updates the pointer by 1. Both 8- and 16-bit read operations may specify either to sign-extend or zero-extend the contents into the destination register. Pointer registers may be used for 8-, 16-, and 32-bit accesses while index registers may be used only for 16- and 32-bit accesses.

For example:

\[ \text{R0} = W[P1++] \] (Z);

loads a 16-bit word into a 32-bit destination register from an address pointed to by the \( P1 \) pointer register. The pointer is then incremented by 2 and the word is zero-extended to fill the 32-bit destination register.

Auto-decrement works the same way by decrementing the address after the access.

For example:

\[ \text{R0} = [I2--] ; \]

loads a 32-bit value into the destination register and decrements the Index register by 4.

Pre-Modify Stack Pointer Addressing

The only pre-modify instruction in the processor uses the stack pointer register, \( SP \). The address in \( SP \) is decremented by 4 and then used as an effective address for the store. The instruction \[ [ --SP ] = \text{R0} ; \] is used for stack push operations and can support only a 32-bit word transfer.
Indexed Addressing With Immediate Offset

Indexed addressing allows programs to obtain values from data tables, with reference to the base of that table. The pointer register is modified by the immediate field and then used as the effective address. The value of the pointer register is not updated.

Alignment exceptions are triggered when a final address is unaligned.

For example, if \( P1 = 0x13 \), then \( [P1 + 0x11] \) would effectively be equal to \( [0x24] \), which is aligned for all accesses.

Post-Modify Addressing

Post-modify addressing uses the value in the index or pointer registers as the effective address and then modifies it by the contents of another register. Pointer registers are modified by other pointer registers. Index registers are modified by modify registers. Post-modify addressing does not support the pointer registers as destination registers, nor does it support byte-addressing.

For example:

\[
R5 = [ P1++, P2 ];
\]

loads a 32-bit value into the \( R5 \) register, found in the memory location pointed to by the \( P1 \) register.

The value in the \( P2 \) register is then added to the value in the \( P1 \) register.

For example:

\[
R2 = W [ P4++, P5 ]; (Z);
\]

loads a 16-bit word into the low half of the destination register \( R2 \) and zero-extends it to 32 bits. The value of the pointer \( P4 \) is incremented by the value of the pointer \( P5 \).
For example:

```
R2 = [ I2++M1 ];
```

loads a 32-bit word into the destination register R2. The value in the Index register, I2, is updated by the value in the modify register, M1.

**Modifying DAG and Pointer Registers**

The DAGs support operations that modify an address value in an index register without outputting an address. The operation, address-modify, is useful for maintaining pointers.

The address-modify operation modifies addresses in any DAG index and pointer register (I[3:0], P[5:0], FP, SP) without accessing memory. If the index register’s corresponding B- and L-registers are set up for circular buffering, the address-modify operation performs the specified buffer wraparound (if needed).

The syntax is similar to post-modify addressing (index += modifier). For Index registers, an M-register is used as the modifier. For pointer registers, another P-register is used as the modifier.

Consider the example, `I1 += M2`;

This instruction adds M2 to I1 and updates I1 with the new value.
Memory Address Alignment

The processor requires proper memory alignment to be maintained for the data size being accessed. Unless exceptions are disabled, violations of memory alignment cause an alignment exception. Some instructions—for example, many of the video ALU instructions—automatically disable alignment exceptions because the data may not be properly aligned when stored in memory. Alignment exceptions may be disabled by issuing the DISALGNEXPT instruction in parallel with a load/store operation.

Normally, the memory system requires two address alignments:

- 32-bit word load/stores are accessed on four-byte boundaries, meaning the two least significant bits of the address are b#00.
- 16-bit word load/stores are accessed on two-byte boundaries, meaning the least significant bit of the address must be b#0.

Table 5-1 summarizes the types of transfers and transfer sizes supported by the addressing modes.
Be careful when using the `DISALGNEXPT` instruction, because it disables automatic detection of memory alignment errors. The `DISALGNEXPT` instruction only affects misaligned loads that use I-register indirect addressing. Misaligned loads using P-register addressing will still cause an exception.

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Types of Transfers Supported</th>
<th>Transfer Sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto-increment</td>
<td>To and from data registers</td>
<td>LOADS: 32-bit word 16-bit, zero-extended half word 16-bit, sign-extended half word 8-bit, zero-extended byte 8-bit, sign-extended byte STORES: 32-bit word 16-bit half word 8-bit byte</td>
</tr>
<tr>
<td>Auto-decrement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Indirect</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>To and from pointer registers</td>
<td>LOAD: 32-bit word STORE: 32-bit word</td>
</tr>
<tr>
<td>Post-increment</td>
<td>To and from data registers</td>
<td>LOADS: 32-bit word 16-bit half word to data register high half 16-bit half word to data register low half 16-bit, zero-extended half word 16-bit, sign-extended half word STORES: 32-bit word 16-bit half word from data register high half 16-bit half word from data register low half</td>
</tr>
</tbody>
</table>

Table 5-1. Types of Transfers Supported and Transfer Sizes
Table 5-2 summarizes the addressing modes. In the table, an asterisk (*) indicates the processor supports the addressing mode.

Table 5-2. Addressing Modes

<table>
<thead>
<tr>
<th></th>
<th>32-Bit Word</th>
<th>16-Bit Half-Word</th>
<th>8-Bit Byte</th>
<th>Sign-/Zero-Extend</th>
<th>Data Register</th>
<th>Pointer Register</th>
<th>Data Register Half</th>
</tr>
</thead>
<tbody>
<tr>
<td>P Auto-inc</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>[P0++]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P Auto-dec</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>[P0--]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P Indirect</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>[P0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P Indexed</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>[P0+im]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP indexed</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>[FP+im]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P Post-inc</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>[P0++P1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I Auto-inc</td>
<td>*</td>
<td></td>
<td>*</td>
<td></td>
<td>*</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>[I0++]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I Auto-dec</td>
<td>*</td>
<td></td>
<td>*</td>
<td></td>
<td>*</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>[I0--]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I Indirect</td>
<td>*</td>
<td></td>
<td>*</td>
<td></td>
<td>*</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>[I0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I Post-inc</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>[I0++M0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Data Address Generators

DAG Instruction Summary

Table 5-3 lists the DAG instructions. For more information on assembly language syntax, see Blackfin Processor Programming Reference. In Table 5-3, note the meaning of these symbols:

- Dreg denotes any data register file register.
- Dreg_lo denotes the lower 16 bits of any data register file register.
- Dreg_hi denotes the upper 16 bits of any data register file register.
- Preg denotes any pointer register, FP, or SP register.
- Ireg denotes any DAG index register.
- Mreg denotes any DAG modify register.
- W denotes a 16-bit wide value.
- B denotes an 8-bit wide value.
- immA denotes a signed, A-bits wide, immediate value.
- uimmAmB denotes an unsigned, A-bits wide, immediate value that is an even multiple of B.
- Z denotes the zero-extension qualifier.
- X denotes the sign-extension qualifier.
- BREV denotes the bit-reversal qualifier.

The Blackfin Processor Programming Reference more fully describes the options that may be applied to these instructions and the sizes of immediate fields.

DAG instructions do not affect the ASTAT register flags.
## Table 5-3. DAG Instruction Summary

<table>
<thead>
<tr>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Preg = [ Preg ] ;</code></td>
</tr>
<tr>
<td><code>Preg = [ Preg ++ ] ;</code></td>
</tr>
<tr>
<td><code>Preg = [ Preg -- ] ;</code></td>
</tr>
<tr>
<td><code>Preg = [ Preg + uimm6m4 ] ;</code></td>
</tr>
<tr>
<td><code>Preg = [ Preg + uimm17m4 ] ;</code></td>
</tr>
<tr>
<td><code>Preg = [ Preg – uimm17m4 ] ;</code></td>
</tr>
<tr>
<td><code>Preg = [ FP – uimm7m4 ] ;</code></td>
</tr>
<tr>
<td><code>Dreg = [ Preg ] ;</code></td>
</tr>
<tr>
<td><code>Dreg = [ Preg ++ ] ;</code></td>
</tr>
<tr>
<td><code>Dreg = [ Preg -- ] ;</code></td>
</tr>
<tr>
<td><code>Dreg = [ Preg + uimm6m4 ] ;</code></td>
</tr>
<tr>
<td><code>Dreg = [ Preg + uimm17m4 ] ;</code></td>
</tr>
<tr>
<td><code>Dreg = [ Preg – uimm17m4 ] ;</code></td>
</tr>
<tr>
<td><code>Dreg = [ Preg ++ Preg ] ;</code></td>
</tr>
<tr>
<td><code>Dreg = [ FP – uimm7m4 ] ;</code></td>
</tr>
<tr>
<td><code>Dreg = [ Ireg ] ;</code></td>
</tr>
<tr>
<td><code>Dreg = [ Ireg ++ ] ;</code></td>
</tr>
<tr>
<td><code>Dreg = [ Ireg -- ] ;</code></td>
</tr>
<tr>
<td><code>Dreg = [ Ireg + Mreg ] ;</code></td>
</tr>
<tr>
<td><code>Dreg =W [ Preg ] (Z) ;</code></td>
</tr>
<tr>
<td><code>Dreg =W [ Preg ++ ] (Z) ;</code></td>
</tr>
<tr>
<td><code>Dreg =W [ Preg -- ] (Z) ;</code></td>
</tr>
<tr>
<td><code>Dreg =W [ Preg + uimm5m2 ] (Z) ;</code></td>
</tr>
<tr>
<td><code>Dreg =W [ Preg + uimm16m2 ] (Z) ;</code></td>
</tr>
<tr>
<td><code>Dreg =W [ Preg – uimm16m2 ] (Z) ;</code></td>
</tr>
<tr>
<td><code>Dreg =W [ Preg ++ Preg ] (Z) ;</code></td>
</tr>
<tr>
<td>Instruction</td>
</tr>
<tr>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>Dreg = W [ Preg ] (X) ;</td>
</tr>
<tr>
<td>Dreg = W [ Preg ++ ] (X) ;</td>
</tr>
<tr>
<td>Dreg = W [ Preg -- ] (X) ;</td>
</tr>
<tr>
<td>Dreg = W [ Preg + uimm5m2 ] (X) ;</td>
</tr>
<tr>
<td>Dreg = W [ Preg + uimm16m2 ] (X) ;</td>
</tr>
<tr>
<td>Dreg = W [ Preg - uimm16m2 ] (X) ;</td>
</tr>
<tr>
<td>Dreg = W [ Preg ++ Preg ] (X) ;</td>
</tr>
<tr>
<td>Dreg_hi = W [ Ireg ] ;</td>
</tr>
<tr>
<td>Dreg_hi = W [ Ireg ++ ] ;</td>
</tr>
<tr>
<td>Dreg_hi = W [ Ireg -- ] ;</td>
</tr>
<tr>
<td>Dreg_hi = W [ Preg ] ;</td>
</tr>
<tr>
<td>Dreg_hi = W [ Preg ++ Preg ] ;</td>
</tr>
<tr>
<td>Dreg_lo = W [ Ireg ] ;</td>
</tr>
<tr>
<td>Dreg_lo = W [ Ireg ++ ] ;</td>
</tr>
<tr>
<td>Dreg_lo = W [ Ireg -- ] ;</td>
</tr>
<tr>
<td>Dreg_lo = W [ Preg ] ;</td>
</tr>
<tr>
<td>Dreg_lo = W [ Preg ++ Preg ] ;</td>
</tr>
<tr>
<td>Dreg = B [ Preg ] (Z) ;</td>
</tr>
<tr>
<td>Dreg = B [ Preg ++ ] (Z) ;</td>
</tr>
<tr>
<td>Dreg = B [ Preg -- ] (Z) ;</td>
</tr>
<tr>
<td>Dreg = B [ Preg + uimm15 ] (Z) ;</td>
</tr>
<tr>
<td>Dreg = B [ Preg - uimm15 ] (Z) ;</td>
</tr>
<tr>
<td>Dreg = B [ Preg ] (X) ;</td>
</tr>
<tr>
<td>Dreg = B [ Preg ++ ] (X) ;</td>
</tr>
<tr>
<td>Dreg = B [ Preg -- ] (X) ;</td>
</tr>
<tr>
<td>Dreg = B [ Preg + uimm15 ] (X) ;</td>
</tr>
</tbody>
</table>
### Table 5-3. DAG Instruction Summary (Cont’d)

<table>
<thead>
<tr>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dreg = B [ Preg – uimm15 ] (X) ;</td>
</tr>
<tr>
<td>[ Preg ] = Preg ;</td>
</tr>
<tr>
<td>[ Preg ++ ] = Preg ;</td>
</tr>
<tr>
<td>[ Preg -- ] = Preg ;</td>
</tr>
<tr>
<td>[ Preg + uimm6m4 ] = Preg ;</td>
</tr>
<tr>
<td>[ Preg + uimm17m4 ] = Preg ;</td>
</tr>
<tr>
<td>[ Preg – uimm17m4 ] = Preg ;</td>
</tr>
<tr>
<td>[ FP – uimm7m4 ] = Preg ;</td>
</tr>
<tr>
<td>[ Preg ] = Dreg ;</td>
</tr>
<tr>
<td>[ Preg ++ ] = Dreg ;</td>
</tr>
<tr>
<td>[ Preg -- ] = Dreg ;</td>
</tr>
<tr>
<td>[ Preg + uimm6m4 ] = Dreg ;</td>
</tr>
<tr>
<td>[ Preg + uimm17m4 ] = Dreg ;</td>
</tr>
<tr>
<td>[ Preg – uimm17m4 ] = Dreg ;</td>
</tr>
<tr>
<td>[ Preg ++ Preg ] = Dreg ;</td>
</tr>
<tr>
<td>[FP – uimm7m4 ] = Dreg ;</td>
</tr>
<tr>
<td>[ Ireg ] = Dreg ;</td>
</tr>
<tr>
<td>[ Ireg ++ ] = Dreg ;</td>
</tr>
<tr>
<td>[ Ireg -- ] = Dreg ;</td>
</tr>
<tr>
<td>[ Ireg ++ Mreg ] = Dreg ;</td>
</tr>
<tr>
<td>W [ Ireg ] = Dreg_hi ;</td>
</tr>
<tr>
<td>W [ Ireg ++ ] = Dreg_hi ;</td>
</tr>
<tr>
<td>W [ Ireg -- ] = Dreg_hi ;</td>
</tr>
<tr>
<td>W [ Preg ] = Dreg_hi ;</td>
</tr>
<tr>
<td>W [ Preg ++ Preg ] = Dreg_hi ;</td>
</tr>
<tr>
<td>W [ Ireg ] = Dreg_lo ;</td>
</tr>
</tbody>
</table>
Table 5-3. DAG Instruction Summary (Cont’d)

<table>
<thead>
<tr>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W [ Ireg ++ ] = Dreg_lo ;</td>
</tr>
<tr>
<td>W [ Ireg -- ] = Dreg_lo ;</td>
</tr>
<tr>
<td>W [ Preg ] = Dreg_lo ;</td>
</tr>
<tr>
<td>W [ Preg ] = Dreg ;</td>
</tr>
<tr>
<td>W [ Preg ++ ] = Dreg ;</td>
</tr>
<tr>
<td>W [ Preg -- ] = Dreg ;</td>
</tr>
<tr>
<td>W [ Preg + uimm5m2 ] = Dreg ;</td>
</tr>
<tr>
<td>W [ Preg + uimm16m2 ] = Dreg ;</td>
</tr>
<tr>
<td>W [ Preg – uimm16m2 ] = Dreg ;</td>
</tr>
<tr>
<td>W [ Preg ++ Preg ] = Dreg_lo ;</td>
</tr>
<tr>
<td>B [ Preg ] = Dreg ;</td>
</tr>
<tr>
<td>B [ Preg ++ ] = Dreg ;</td>
</tr>
<tr>
<td>B [ Preg -- ] = Dreg ;</td>
</tr>
<tr>
<td>Preg = imm7 (X) ;</td>
</tr>
<tr>
<td>Preg = imm16 (X) ;</td>
</tr>
<tr>
<td>Preg += Preg (BREV) ;</td>
</tr>
<tr>
<td>Ireg += Mreg (BREV) ;</td>
</tr>
<tr>
<td>Preg = Preg &lt;&lt; 2 ;</td>
</tr>
<tr>
<td>Preg = Preg &gt;&gt; 2 ;</td>
</tr>
<tr>
<td>Preg = Preg &gt;&gt; 1 ;</td>
</tr>
<tr>
<td>Preg = Preg + Preg &lt;&lt; 1 ;</td>
</tr>
<tr>
<td>Preg = Preg + Preg &lt;&lt; 2 ;</td>
</tr>
<tr>
<td>Preg --= Preg ;</td>
</tr>
<tr>
<td>Ireg --= Mreg ;</td>
</tr>
</tbody>
</table>
6 MEMORY

The processor supports a hierarchical memory model with different performance and size parameters, depending on the memory location within the hierarchy. Level 1 (L1) memories are located on the chip and are faster than the Level 2 (L2) memory systems. The Level 2 (L2) memories are off-chip and have longer access latencies. The faster L1 memories, which are typically small scratchpad memory or cache memories, are found within the core itself.

Memory Architecture

The processor has a unified 4G byte address range that spans a combination of on-chip and off-chip memory and memory-mapped I/O resources. Of this range, some of the address space is dedicated to internal, on-chip resources. The processor populates portions of this internal memory space with:

- L1 static random access memories (SRAM)
- a set of memory-mapped registers (MMRs)
- a boot read-only memory (ROM)

A portion of the internal L1 SRAM can also be configured to run as cache. The processor also provides support for an external memory space that includes asynchronous memory space and synchronous DRAM (SDRAM) space. See Chapter 18, “External Bus Interface Unit” for a detailed discussion of each of these memory regions and the controllers that support them.
**Figure 6-1** provides an overview of the ADSP-BF538/ADSP-BF538F processor system memory map. Note the architecture does not define a separate I/O space. All resources are mapped through the flat 32-bit address space. The memory is byte-addressable.

The memory configuration for the ADSP-BF538/ADSP-BF538F processors is shown in the *ADSP-BF538/ADSP-BF538F Embedded Processor Data Sheet*.

The upper portion of internal memory space is allocated to the core and system MMRs. Accesses to this area are allowed only when the processor is in supervisor or emulation mode (see Chapter 3, “Operating Modes and States”).

The lowest 1K byte of internal memory space is occupied by the boot ROM. Depending on the booting option selected, the appropriate boot program is executed from this memory space when the processor is reset (see “Booting Methods” on page 3-19).

Within the external memory map, four banks of asynchronous memory space and one bank of SDRAM memory are available. Each of the asynchronous banks is 1M byte and the SDRAM bank is up to 128M byte.

**Overview of Internal Memory**

The L1 memory system performance provides high bandwidth and low latency. Because SRAMs provide deterministic access time and very high throughput, DSP systems have traditionally achieved performance improvements by providing fast SRAM on the chip.

The addition of instruction and data caches (SRAMs with cache control hardware) provides both high performance and a simple programming model. Caches eliminate the need to explicitly manage data movement into and out of L1 memories. Code can be ported to or developed for the processor quickly without requiring performance optimization for the memory organization.
Figure 6-1. ADSP-BF538/ADSP-BF538F Memory Map
The L1 memory provides:

- A modified Harvard architecture, allowing up to four core memory accesses per clock cycle (one 64-bit instruction fetch, two 32-bit data loads, and one pipelined 32-bit data store)
- Simultaneous system DMA, cache maintenance, and core accesses
- SRAM access at processor clock rate \((CCLK)\) for critical DSP algorithms and fast context switching
• Instruction and data cache options for microcontroller code, excellent high level language (HLL) support, and ease of programming cache control instructions, such as PREFETCH and FLUSH

• Memory protection

The L1 memories operate at the core clock frequency (CCLK).

Overview of Scratchpad Data SRAM

The processor provides a dedicated 4K byte bank of scratchpad data SRAM. The scratchpad is independent of the configuration of the other L1 memory banks and cannot be configured as cache or targeted by DMA. Typical applications use the scratchpad data memory where speed is critical. For example, the user and supervisor stacks should be mapped to the scratchpad memory for the fastest context switching during interrupt handling.

Scratchpad data SRAM cannot be accessed by the DMA controller.

L1 Instruction Memory

L1 instruction memory consists of a combination of dedicated SRAM and banks which can be configured as SRAM or cache. For the 16K byte bank that can be either cache or SRAM, control bits in the IMEM_CONTROL register can be used to organize all four sub-banks of the L1 instruction memory as a:

• simple SRAM
• 4-way, set associative instruction cache
• cache with as many as four locked ways
L1 Instruction Memory

L1 instruction memory can be used only to store instructions.

**Instruction Memory Control (IMEM_CONTROL) Register**

The instruction memory control register (IMEM_CONTROL) contains control bits for the L1 instruction memory. By default after reset, cache and cacheability protection looksaside buffer (CPLB) address checking is disabled (see “L1 Instruction Cache” on page 6-9).

When the LRUPRIORST bit is set to 1, the cached states of all CPLB_LRUPRIO bits (see “Instruction CPLB Data (ICPLB_DATAx) Registers” on page 6-52) are cleared. This simultaneously forces all cached lines to be of equal (low) importance. Cache replacement policy is based first on line importance indicated by the cached states of the CPLB_LRUPRIO bits, and then on LRU (least recently used). See “Instruction Cache Locking by Line” on page 6-17 for complete details. This bit must be 0 to allow the state of the CPLB_LRUPRIO bits to be stored when new lines are cached.

The ILOC[3:0] bits provide a useful feature only after code has been manually loaded into cache. See “Instruction Cache Locking by Way” on page 6-17. These bits specify which ways to remove from the cache replacement policy. This has the effect of locking code present in nonparticipating ways. Code in nonparticipating ways can still be removed from the cache using an I_FLUSH instruction. If an ILOC[3:0] bit is 0, the corresponding way is not locked and that way participates in cache replacement policy. If an ILOC[3:0] bit is 1, the corresponding way is locked and does not participate in cache replacement policy.

The IMC bit reserves a portion of L1 instruction SRAM to serve as cache. Note reserving memory to serve as cache will not alone enable L2 memory accesses to be cached. CPLBs must also be enabled using the EN_ICPLB bit and the CPLB descriptors (ICPLB_DATAx and ICPLB_ADDRx registers) must specify desired memory pages as cache-enabled.
Instruction CPLBs are disabled by default after reset. When disabled, only minimal address checking is performed by the L1 memory interface. This minimal checking generates an exception to the processor whenever it attempts to fetch an instruction from:

- Reserved (non populated) L1 instruction memory space
- L1 data memory space
- MMR space

CPLBs must be disabled using this bit prior to updating their descriptors (DCPLB_DATAx and DCPLB_ADDRx registers). Note since load store ordering is weak (see “Ordering of Loads and Stores” on page 6-64), disabling of CPLBs should be proceeded by a CSYNC.

When enabling or disabling cache or CPLBs, immediately follow the write to IMEM_CONTROL with a CSYNC to ensure proper behavior.

To ensure proper behavior and future compatibility, all reserved bits in this register must be set to 0 whenever this register is written.

**L1 Instruction SRAM**

The processor core reads the instruction memory through the 64-bit wide instruction fetch bus. All addresses from this bus are 64-bit aligned. Each instruction fetch can return any combination of 16-, 32- or 64-bit instructions (for example, four 16-bit instructions, two 16-bit instructions and one 32-bit instruction, or one 64-bit instruction).

The DAGs, which are described in Chapter 5, “Data Address Generators”, cannot access L1 instruction memory directly. A DAG reference to instruction memory SRAM space generates an exception (see “Exceptions” on page 4-46).
Write access to the L1 instruction SRAM memory must be made through the 64-bit wide system DMA port. Because the SRAM is implemented as a collection of single ported sub-banks, the instruction memory is effectively dual ported.

Table 6-1 lists the memory start locations of the L1 instruction memory sub-banks.
Figure 6-4 describes the bank architecture of the L1 instruction memory. As the figure shows, each 16K byte bank is made up of four 4K byte sub-banks.

### L1 Instruction Cache

For information about cache terminology, see “Terminology” on page 6-71.
Figure 6-4. L1 Instruction Memory Bank Architecture
The L1 instruction memory may also be configured to contain a 4-way set associative instruction 16K byte cache. To improve the average access latency for critical code sections, each 2ay or line of the cache can be locked independently. When the memory is configured as cache, it cannot be accessed directly.

When cache is enabled, only memory pages further specified as cacheable by the CPLBs will be cached. When CPLBs are enabled, any memory location that is accessed must have an associated page definition available, or a CPLB exception is generated. CPLBs are described in “Memory Protection and Properties” on page 6-43.

Figure 6-5 shows the overall Blackfin processor instruction cache organization.

**Cache Lines**

As shown in Figure 6-5, the cache consists of a collection of cache lines. Each cache line is made up of a *tag* component and a *data* component.

- The tag component incorporates a 20-bit address tag, least recently used (LRU) bits, a valid bit, and a line lock bit.
- The data component is made up of four 64-bit words of instruction data.

The tag and data components of cache lines are stored in the tag and data memory arrays, respectively.

The address tag consists of the upper 18 bits plus bits 11 and 10 of the physical address. Bits 12 and 13 of the physical address are not part of the address tag. Instead, these bits are used to identify the 4K byte memory sub-bank targeted for the access.

The LRU bits are part of an LRU algorithm used to determine which cache line should be replaced if a cache miss occurs.
L1 Instruction Memory

The valid bit indicates the state of a cache line. A cache line is always valid or invalid.

- Invalid cache lines have their valid bit cleared, indicating the line will be ignored during an address-tag compare operation.
- Valid cache lines have their valid bit set, indicating the line contains valid instruction/data that is consistent with the source memory.
Figure 6-5. Blackfin Processor Instruction Cache Organization
L1 Instruction Memory

The tag and data components of a cache line are illustrated in Figure 6-6.

```plaintext
+----------------+------------------+
| TAG            | LRUPRIO          |
+----------------+------------------+
| TAG - 20-BIT ADDRESS TAG |
| LRUPRIO - LRU PRIORITY BIT FOR LINE LOCKING |
| LRU - LRU STATE |
| V - VALID BIT   |
+----------------+------------------+
| WD 3 | WD 2 | WD 1 | WD 0 |
+----------------+------------------+
| WD - 64-BIT DATA WORD |
```

Figure 6-6. Cache Line – Tag and Data Portions

Cache Hits and Misses

A cache hit occurs when the address for an instruction fetch request from the core matches a valid entry in the cache. Specifically, a cache hit is determined by comparing the upper 18 bits and bits 11 and 10 of the instruction fetch address to the address tags of valid lines currently stored in a cache set. The cache set is selected, using bits 9 through 5 of the instruction fetch address. If the address-tag compare operation results in a match, a cache hit occurs. If the address-tag compare operation does not result in a match, a cache miss occurs.

When a cache miss occurs, the instruction memory unit generates a cache line fill access to retrieve the missing cache line from memory that is external to the core. The address for the external memory access is the address of the target instruction word. When a cache miss occurs, the core halts until the target instruction word is returned from external memory.

Cache Line Fills

A cache line fill consists of fetching 32 bytes of data from memory. The operation starts when the instruction memory unit requests a line-read data transfer (a burst of four 64-bit words of data) on its external
read-data port. The address for the read transfer is the address of the target instruction word. When responding to a line-read request from the instruction memory unit, the external memory returns the target instruction word first. After it has returned the target instruction word, the next three words are fetched in sequential address order. This fetch wraps around if necessary, as shown in Table 6-2.

Table 6-2. Cache Line Word Fetching Order

<table>
<thead>
<tr>
<th>Target Word</th>
<th>Fetching Order for Next Three Words</th>
</tr>
</thead>
<tbody>
<tr>
<td>WD0</td>
<td>WD0, WD1, WD2, WD3</td>
</tr>
<tr>
<td>WD1</td>
<td>WD1, WD2, WD3, WD0</td>
</tr>
<tr>
<td>WD2</td>
<td>WD2, WD3, WD0, WD1</td>
</tr>
<tr>
<td>WD3</td>
<td>WD3, WD0, WD1, WD2</td>
</tr>
</tbody>
</table>

Line Fill Buffer

As the new cache line is retrieved from external memory, each 64-bit word is buffered in a four-entry line fill buffer before it is written to a 4K byte memory bank within L1 memory. The line fill buffer allows the core to access the data from the new cache line as the line is being retrieved from external memory, rather than having to wait until the line has been written into the cache.

Cache Line Replacement

When the instruction memory unit is configured as cache, bits 9 through 5 of the instruction fetch address are used as the index to select the cache set for the tag-address compare operation. If the tag-address compare operation results in a cache miss, the valid and LRU bits for the selected set are examined by a cache line replacement unit to determine the entry to use for the new cache line, that is, whether to use way0, way1, way2, or way3. See Figure 6-5.
L1 Instruction Memory

The cache line replacement unit first checks for invalid entries (that is, entries having its valid bit cleared). If only a single invalid entry is found, that entry is selected for the new cache line. If multiple invalid entries are found, the replacement entry for the new cache line is selected based on the following priority:

- way0 first
- way1 next
- way2 next
- way3 last

For example:

- If way3 is invalid and ways0, 1, 2 are valid, way3 is selected for the new cache line.
- If ways0 and 1 are invalid and ways2 and 3 are valid, way0 is selected for the new cache line.
- If ways2 and 3 are invalid and ways0 and 1 are valid, way2 is selected for the new cache line.

When no invalid entries are found, the cache replacement logic uses an LRU algorithm.

Instruction Cache Management

The system DMA controller and the core DAGs cannot access the instruction cache directly. By a combination of instructions and the use of core MMRs, it is possible to initialize the instruction tag and data arrays indirectly and provide a mechanism for instruction cache test, initialization, and debug.
The coherency of instruction cache must be explicitly managed. To accomplish this and ensure that the instruction cache fetches the latest version of any modified instruction space, invalidate instruction cache line entries, as required.

See “Instruction Cache Invalidation” on page 6-18.

**Instruction Cache Locking by Line**

The CPLB_LRUPRIO bits in the ICPLB_DATAx registers (see “Memory Protection and Properties” on page 6-43) are used to enhance control over which code remains resident in the instruction cache. When a cache line is filled, the state of this bit is stored along with the line’s tag. It is then used in conjunction with the LRU (least recently used) policy to determine which Way is victimized when all cache Ways are occupied when a new cacheable line is fetched. This bit indicates that a line is of either “low” or “high” importance. In a modified LRU policy, a high can replace a low, but a low cannot replace a high. If all Ways are occupied by highs, an otherwise cacheable low will still be fetched for the core, but will not be cached. Fetched highs seek to replace unoccupied ways first, then least recently used lows next, and finally other highs using the LRU policy. Lows can only replace unoccupied ways or other lows, and do so using the LRU policy. If all previously cached highs ever become less important, they may be simultaneously transformed into lows by writing to the LRU_PRIRST bit in the IMEM_CONTROL register (see “Instruction Memory Control (IMEM_CONTROL) Register” on page 6-6).

**Instruction Cache Locking by Way**

The instruction cache has four independent lock bits (ILOC[3:0]) that control each of the four ways of the instruction cache. When the cache is enabled, L1 instruction memory has four ways available. Setting the lock bit for a specific way prevents that way from participating in the LRU replacement policy. Thus, a cached instruction with its way locked can only be removed using an I_FLUSH instruction, or a “back door” MMR assisted manipulation of the tag array.
An example sequence is provided below to demonstrate how to lock down way0:

- If the code of interest may already reside in the instruction cache, invalidate the entire cache first (for an example, see “Instruction Cache Management” on page 6-16).
- Disable interrupts, if required, to prevent interrupt service routines (ISRs) from potentially corrupting the locked cache.
- Set the locks for the other ways of the cache by setting ILOC[3:1]. Only way0 of the instruction cache can now be replaced by new code.
- Execute the code of interest. Any cacheable exceptions, such as exit code, traversed by this code execution are also locked into the instruction cache.
- Upon exit of the critical code, clear ILOC[3:1] and set ILOC[0]. The critical code (and the instructions which set ILOC[0]) is now locked into way0.
- Re-enable interrupts, if required.

If all four ways of the cache are locked, then further allocation into the cache is prevented.

**Instruction Cache Invalidation**

The instruction cache can be invalidated by address, cache line, or complete cache. The IFLUSH instruction can explicitly invalidate cache lines based on their line addresses. The target address of the instruction is generated from the P-registers. Because the instruction cache should not contain modified (dirty) data, the cache line is simply invalidated.

In the following example, the P2 register contains the address of a valid memory location. If this address has been brought into cache, the corresponding cache line is invalidated after the execution of this instruction.
Example of ICACHE instruction:
iflush [ p2 ] ; /* Invalidate cache line containing address
that P2 points to */

Because the IFLUSH instruction is used to invalidate a specific address in
the memory map, it is impractical to use this instruction to invalidate an
entire way or bank of cache. A second technique can be used to invalidate
larger portions of the cache directly. This second technique directly inval-
idates valid bits by setting the invalid bit of each cache line to the invalid
state. To implement this technique, additional MMRs (ITEST_COMMAND
and ITEST_DATA[1:0]) are available to allow arbitrary read/write of all the
cache entries directly. This method is explained in the next section.

For invalidating the complete instruction cache, a third method is avail-
able. By clearing the IMC bit in the IMEM_CONTROL register (see Figure 6-3),
all valid bits in the instruction cache are set to the invalid state. A second
write to the IMEM_CONTROL register to set the IMC bit configures the instruc-
tion memory as cache again. An SSYNC instruction should be run before
invalidating the cache and a CSYNC instruction should be inserted after
each of these operations.
Instruction Test Registers

The instruction test registers allow arbitrary read/write of all L1 cache entries directly. They make it possible to initialize the instruction tag and data arrays and to provide a mechanism for instruction cache test, initialization, and debug.

When the instruction test command register (ITEST_COMMAND) is used, the L1 cache data or tag arrays are accessed, and data is transferred through the instruction test data registers (ITEST_DATA[1:0]). The ITEST_DATAx registers contain either the 64-bit data that the access is to write to or the 64-bit data that was read during the access. The lower 32 bits are stored in the ITEST_DATA[0] register, and the upper 32 bits are stored in the ITEST_DATA[1] register. When the tag arrays are accessed, ITEST_DATA[0] is used. Graphical representations of the ITEST registers begin with Figure 6-7.

The following figures describe the ITEST registers:

- Figure 6-7
- Figure 6-8
- Figure 6-9

Access to these registers is possible only in supervisor or emulation mode. When writing to ITEST registers, always write to the ITEST_DATAx registers first, then the ITEST_COMMAND register. When reading from ITEST registers, reverse the sequence—read the ITEST_COMMAND register first, then the ITEST_DATAx registers.
Instruction Test Command (ITEST_COMMAND) Register

When the instruction test command register (ITEST_COMMAND) is written to, the L1 cache data or tag arrays are accessed, and the data is transferred through the instruction test data registers (ITEST_DATA[1:0]).

Instruction Test Command Register (ITEST_COMMAND)

0xFFE0 1300

<table>
<thead>
<tr>
<th>Bit 31</th>
<th>Bit 30</th>
<th>Bit 29</th>
<th>Bit 28</th>
<th>Bit 27</th>
<th>Bit 26</th>
<th>Bit 25</th>
<th>Bit 24</th>
<th>Bit 23</th>
<th>Bit 22</th>
<th>Bit 21</th>
<th>Bit 20</th>
<th>Bit 19</th>
<th>Bit 18</th>
<th>Bit 17</th>
<th>Bit 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Reset = 0x0000 0000</td>
</tr>
</tbody>
</table>

WAYSEL[1:0] (Access Way)
- 00 - Access Way0
- 01 - Access Way1
- 10 - Access Way2
- 11 - Access Way3
(Address bits [11:10] in SRAM)

SBNK[1:0] (sub-bank Access)
- 00 - Access sub-bank 0
- 01 - Access sub-bank 1
- 10 - Access sub-bank 2
- 11 - Access sub-bank 3
(Address bits [13:12] in SRAM)

SET[4:0] (Set Index)
- Selects one of 32 sets
(Address bits [9:5] in SRAM)

RW (Read/Write Access)
- 0 - Read access
- 1 - Write access

TAGSELB (Array Access)
- 0 - Access tag array
- 1 - Access data array

Figure 6-7. Instruction Test Command Register
Instruction Test Registers

Instruction Test Data (ITEST_DATA1) Register

Instruction test data registers (ITEST_DATA[1:0]) are used to access L1 cache data arrays. They contain either the 64-bit data that the access is to write to or the 64-bit data that the access is to read from. The instruction test data 1 register (ITEST_DATA1) stores the upper 32 bits.

Instruction Test Data 1 Register (ITEST_DATA1)

Used to access L1 cache data arrays and tag arrays. When accessing a data array, stores the upper 32 bits of 64-bit words of instruction data to be written to or read from by the access. See “Cache Lines” on page 6-11.

0xFFE 1404

When accessing tag arrays, all bits are reserved.

Figure 6-8. Instruction Test Data 1 Register
Instruction Test Data 0 (ITEST_DATA0) Register

The instruction test data 0 register (ITEST_DATA0) stores the lower 32 bits of the 64-bit data to be written to or read from by the access. The ITEST_DATA0 register is also used to access tag arrays. This register also contains the valid and dirty bits, which indicate the state of the cache line.

Instruction Test Data 0 Register (ITEST_DATA0)

Used to access L1 cache data arrays and tag arrays. When accessing a data array, stores the lower 32 bits of 64-bit words of instruction data to be written to or read from by the access. See “Cache Lines” on page 6-11.

Used to access the L1 cache tag arrays. The address tag consists of the upper 18 bits and bits 11 and 10 of the physical address. See “Cache Lines” on page 6-11.

Figure 6-9. Instruction Test Data 0 Register
L1 Data Memory

The L1 data SRAM/cache is constructed from single-ported subsections, but organized to reduce the likelihood of access collisions. This organization results in apparent multi-ported behavior. When there are no collisions, this L1 data traffic could occur in a single core clock cycle:

- Two 32-bit DAG loads
- One pipelined 32-bit DAG store
- One 64-bit DMA IO
- One 64-bit cache fill/victim access

L1 data memory can be used only to store data.

Data Memory Control (DMEM_CONTROL) Register

The data memory control register (DMEM_CONTROL) contains control bits for the L1 data memory.

The PORT_PREF1 bit selects the data port used to process DAG1 non-cacheable L2 fetches. Cacheable fetches are always processed by the data port physically associated with the targeted cache memory. Steering DAG0, DAG1, and cache traffic to different ports optimizes performance by keeping the queue to L2 memory full.

The PORT_PREF0 bit selects the data port used to process DAG0 non-cacheable L2 fetches. Cacheable fetches are always processed by the data port physically associated with the targeted cache memory. Steering DAG0, DAG1, and cache traffic to different ports optimizes performance by keeping the queue to L2 memory full.
For optimal performance with dual DAG reads, DAG0 and DAG1 should be configured for different ports. For example, if PORT_PREF0 is configured as 1, then PORT_PREF1 should be programmed to 0.

The DCBS bit provides some control over which addresses alias into the same set. This bit can be used to affect which addresses tend to remain resident in cache by avoiding victimization of repetitively used sets. It has no affect unless both data bank A and data bank B are serving as cache (bits DMC[1:0] in this register are set to 11).
The ENDCPLB bit is used to enable/disable the 16 cacheability protection lookaside buffers (CPLBs) used for data (see “L1 Data Cache” on page 6-30). Data CPLBs are disabled by default after reset. When disabled, only minimal address checking is performed by the L1 memory interface. This minimal checking generates an exception when the processor:

- Addresses nonexistent (reserved) L1 memory space
- Attempts to perform a nonaligned memory access
- Attempts to access MMR space either using DAG1 or when in User mode

CPLBs must be disabled using this bit prior to updating their descriptors (registers DCPLB_DATAx and DCPLB_ADDRx). Note that since load store ordering is weak (see “Ordering of Loads and Stores” on page 6-64), disabling CPLBs should be preceded by a CSYNC instruction.

When enabling or disabling cache or CPLBs, immediately follow the write to DMEM_CONTROL with a SSYNC to ensure proper behavior.

By default after reset, all L1 data memory serves as SRAM. The DMC[1:0] bits can be used to reserve portions of this memory to serve as cache instead. Reserving memory to serve as cache does not enable L2 memory accesses to be cached. To do this, CPLBs must also be enabled (using the ENDCPLB bit) and CPLB descriptors (registers DCPLB_DATAx and DCPLB_ADDRx) must specify chosen memory pages as cache-enabled.

By default after reset, cache and CPLB address checking is disabled.

To ensure proper behavior and future compatibility, all reserved bits in this register must be set to 0 whenever this register is written.
L1 Data SRAM

Accesses to SRAM do not collide unless all of the following are true: the accesses are to the same 32-bit word polarity (address bits 2 match), the same 4K byte sub-bank (address bits 13 and 12 match), the same 16K byte half bank (address bits 16 match), and the same bank (address bits 21 and 20 match). When an address collision is detected, access is nominally granted first to the DAGs, then to the store buffer, and finally to the DMA and cache fill/victim traffic. To ensure adequate DMA bandwidth, DMA is given highest priority if it has been blocked for more than 16 sequential core clock cycles, or if a second DMA I/O is queued before the first DMA I/O is processed.

Table 6-3 shows how the sub-bank organization is mapped into memory.
Table 6-3. L1 Data Memory SRAM Sub-Bank Start Addresses

<table>
<thead>
<tr>
<th>Memory Bank and Sub-Bank</th>
<th>Start Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data bank A, sub-bank 0</td>
<td>0xFF80 0000</td>
</tr>
<tr>
<td>Data bank A, sub-bank 1</td>
<td>0xFF80 1000</td>
</tr>
<tr>
<td>Data bank A, sub-bank 2</td>
<td>0xFF80 2000</td>
</tr>
<tr>
<td>Data bank A, sub-bank 3</td>
<td>0xFF80 3000</td>
</tr>
<tr>
<td>Data bank A, sub-bank 4</td>
<td>0xFF80 4000</td>
</tr>
<tr>
<td>Data bank A, sub-bank 5</td>
<td>0xFF80 5000</td>
</tr>
<tr>
<td>Data bank A, sub-bank 6</td>
<td>0xFF80 6000</td>
</tr>
<tr>
<td>Data bank A, sub-bank 7</td>
<td>0xFF80 7000</td>
</tr>
<tr>
<td>Data bank B, sub-bank 0</td>
<td>0xFF90 0000</td>
</tr>
<tr>
<td>Data bank B, sub-bank 1</td>
<td>0xFF90 1000</td>
</tr>
<tr>
<td>Data bank B, sub-bank 2</td>
<td>0xFF90 2000</td>
</tr>
<tr>
<td>Data bank B, sub-bank 3</td>
<td>0xFF90 3000</td>
</tr>
<tr>
<td>Data bank B, sub-bank 4</td>
<td>0xFF90 4000</td>
</tr>
<tr>
<td>Data bank B, sub-bank 5</td>
<td>0xFF90 5000</td>
</tr>
<tr>
<td>Data bank B, sub-bank 6</td>
<td>0xFF90 6000</td>
</tr>
<tr>
<td>Data bank B, sub-bank 7</td>
<td>0xFF90 7000</td>
</tr>
</tbody>
</table>

Figure 6-11 shows the L1 data memory architecture.
Figure 6-11. L1 Data Memory Architecture
L1 Data Memory

L1 Data Cache

For definitions of cache terminology, see “Terminology” on page 6-71.

When data cache is enabled (controlled by bits $DMC[1:0]$ in the $DMEM\_CONTROL$ register), either 16K byte of data bank A or 16K byte of both data bank A and data bank B can be set to serve as cache. When configured as cache memory for the processor, the upper 16K bytes of the bank are used. Unlike instruction cache, which is 4-way set associative, data cache is 2-way set associative. When two banks are available and enabled as cache, additional sets rather than ways are created. When both data bank A and data bank B have memory serving as cache, the $DCBS$ bit in the $DMEM\_CONTROL$ register may be used to control which half of all address space is handled by which bank of cache memory. The $DCBS$ bit selects either address bit 14 or 23 to steer traffic between the cache banks. This provides some control over which addresses alias into the same set. It may therefore be used to affect which addresses tend to remain resident in cache by avoiding victimization of repetitively used sets.

Accesses to cache do not collide unless they are to the same 4K byte sub-bank, the same half bank, and to the same bank. Cache has less apparent multi-ported behavior than SRAM due to the overhead in maintaining tags. When cache addresses collide, access is granted first to the $DTEST$ register accesses, then to the store buffer, and finally to cache fill/victim traffic.

Three different cache modes are available.

- Write-through with cache line allocation only on reads
- Write-through with cache line allocation on both reads and writes
- Write-back which allocates cache lines on both reads and writes
Cache mode is selected by the DCPLB descriptors (see “Memory Protection and Properties” on page 6-43). Any combination of these cache modes can be used simultaneously since cache mode is selectable for each memory page independently.

If cache is enabled (controlled by bits DMC[1:0] in the DMEM_CONTROL register), data CPLBs should also be enabled (controlled by ENDCPLB bit in the DMEM_CONTROL register). Only memory pages specified as cacheable by data CPLBs will be cached. The default behavior when data CPLBs are disabled is for nothing to be cached.

Erroneous behavior can result when MMR space is configured as cacheable by data CPLBs, or when data banks serving as L1 SRAM are configured as cacheable by data CPLBs.

Example of Mapping Cacheable Address Space

An example of how the cacheable address space maps into two data banks follows.

When both banks are configured as cache they operate as two independent, 16K byte, 2-way set associative caches that can be independently mapped into the Blackfin processor address space.

If both data banks are configured as cache, the DCBS bit in the DMEM_CONTROL register designates address bit A[14] or A[23] as the cache selector. Address bit A[14] or A[23] selects the cache implemented by data bank A or the cache implemented by data bank B.

- If DCBS = 0, then A[14] is part of the address index, and all addresses in which A[14] = 0 use data bank B. All addresses in which A[14] = 1 use data bank A.

In this case, A[23] is treated as merely another bit in the address that is stored with the tag in the cache and compared for hit/miss processing by the cache.
If \( \text{DCBS} = 1 \), then \( A[23] \) is part of the address index, and all addresses where \( A[23] = 0 \) use data bank B. All addresses where \( A[23] = 1 \) use data bank A.

In this case, \( A[14] \) is treated as merely another bit in the address that is stored with the tag in the cache and compared for hit/miss processing by the cache.

The result of choosing \( \text{DCBS} = 0 \) or \( \text{DCBS} = 1 \) is:

- If \( \text{DCBS} = 0 \), \( A[14] \) selects data bank A instead of data bank B.

Alternating 16K byte pages of memory map into each of the two 16K byte caches implemented by the two data banks.

As a result, the cache operates as a single, contiguous, 2-way set associative 32K byte cache. Each way is 16K byte long, and all data elements with the same first 14 bits of address index to a unique set in which up to two elements can be stored (one in each way).

Any data in the first 16K byte of memory could be stored only in data bank B.

Any data in the next range (16K byte through 32K byte) – 1 could be stored only in data bank A.

Any data in the next range (32K byte through 48K byte) – 1 would be stored in data bank B. Alternate mapping would continue.

- If \( \text{DCBS} = 1 \), \( A[23] \) selects data bank A instead of data bank B.

With \( \text{DCBS} = 1 \), the system functions more like two independent caches, each a 2-way set associative 16K byte cache. Each bank serves an alternating set of 8M byte blocks of memory.
For example, data bank B caches all data accesses for the first 8M byte of memory address range. That is, every 8M byte of range vie for the two line entries (rather than every 16K byte repeat). Likewise, data bank A caches data located above 8M byte and below 16M byte.

For example, if the application is working from a data set that is 1M byte long and located entirely in the first 8M byte of memory, it is effectively served by only half the cache, that is, by data bank B (a 2-way set associative 16K byte cache). In this instance, the application never derives any benefit from data bank A.

- For most applications, it is best to operate with $DCBS = 0$.

However, if the application is working from two data sets, located in two memory spaces at least 8M byte apart, closer control over how the cache maps to the data is possible. For example, if the program is doing a series of dual MAC operations in which both DAGs are accessing data on every cycle, by placing DAG0’s data set in one block of memory and DAG1’s data set in the other, the system can ensure that:

- DAG0 gets its data from data bank A for all of its accesses
- DAG1 gets its data from data bank B

This arrangement causes the core to use both data buses for cache line transfer and achieves the maximum data bandwidth between the cache and the core.

Figure 6-12 shows an example of how mapping is performed when $DCBS = 1$.

The $DCBS$ selection can be changed dynamically; however, to ensure that no data is lost, first flush and invalidate the entire cache.
Data Cache Access

The cache controller tests the address from the DAGs against the tag bits. If the logical address is present in L1 cache, a cache hit occurs, and the data is accessed in L1. If the logical address is not present, a cache miss occurs, and the memory transaction is passed to the next level of memory via the system interface. The line index and replacement policy for the cache controller determines the cache tag and data space that are allocated for the data coming back from external memory.

A data cache line is in one of three states: invalid, exclusive (valid and clean), and modified (valid and dirty). If valid data already occupies the allocated line and the cache is configured for write-back storage, the controller checks the state of the cache line and treats it accordingly:

Figure 6-12. Data Cache Mapping When DCBS = 1
• If the state of the line is exclusive (clean), the new tag and data write over the old line.

• If the state of the line is modified (dirty), then the cache contains the only valid copy of the data.

If the line is dirty, the current contents of the cache are copied back to external memory before the new data is written to the cache.

The processor provides victim buffers and line fill buffers. These buffers are used if a cache load miss generates a victim cache line that should be replaced. The line fill operation goes to external memory. The data cache performs the line fill request to the system as critical (or requested) word first, and forwards that data to the waiting DAG as it updates the cache line. In other words, the cache performs critical word forwarding.

The data cache supports hit-under-a-store miss, and hit-under-a-prefetch miss. In other words, on a write-miss or execution of a \texttt{PREFETCH} instruction that misses the cache (and is to a cacheable region), the instruction pipeline incurs a minimum of a 4-cycle stall. Furthermore, a subsequent load or store instruction can hit in the L1 cache while the line fill completes.

Interrupts of sufficient priority (relative to the current context) cancel a stalled load instruction. Consequently, if the load operation misses the L1 data memory cache and generates a high latency line fill operation on the system interface, it is possible to interrupt the core, causing it to begin processing a different context. The system access to fill the cache line is not cancelled, and the data cache is updated with the new data before any further cache miss operations to the respective data bank are serviced. For more information see “Exceptions” on page 4-46.
L1 Data Memory

Cache Write Method

Cache write memory operations can be implemented by using either a write-through method or a write-back method:

- For each store operation, write-through caches initiate a write to external memory immediately upon the write to cache. If the cache line is replaced or explicitly flushed by software, the contents of the cache line are invalidated rather than written back to external memory.

- A write-back cache does not write to external memory until the line is replaced by a load operation that needs the line.

The L1 data memory employs a full cache line width copyback buffer on each data bank. In addition, a two-entry write buffer in the L1 data memory accepts all stores with cache inhibited or store-through protection. An SSYNC instruction flushes the write buffer.

Interrupt Priority Register and Write Buffer Depth

The interrupt priority register (IPRIO) can be used to control the size of the write buffer on port A (see “L1 Data Memory Architecture” on page 6-29).

The IPRIO[3:0] bits can be programmed to reflect the low priority interrupt watermark. When an interrupt occurs, causing the processor to vector from a low priority interrupt service routine to a high priority interrupt service routine, the size of the write buffer increases from two to eight 32-bit words deep. This allows the interrupt service routine to run and post writes without an initial stall, in the case where the write buffer was already filled in the low priority interrupt routine. This is most useful when posted writes are to a slow external memory device. When returning from a high priority interrupt service routine to a low priority interrupt
service routine or user mode, the core stalls until the write buffer has completed the necessary writes to return to a two-deep state. By default, the write buffer is a fixed two-deep FIFO.

**Data Cache Control Instructions**

The processor defines three data cache control instructions that are accessible in user and supervisor modes. The instructions are `PREFETCH`, `FLUSH`, and `FLUSHINV`.

- `PREFETCH` (data cache prefetch) attempts to allocate a line into the L1 cache. If the prefetch hits in the cache, generates an exception, or addresses a cache inhibited region, `PREFETCH` functions like a `NOP`.

![Interrupt Priority Register (IPRIO)](image)

Figure 6-13. Interrupt Priority Register
L1 Data Memory

- **FLUSH** (data cache flush) causes the data cache to synchronize the specified cache line with external memory. If the cached data line is dirty, the instruction writes the line out and marks the line clean in the data cache. If the specified data cache line is already clean or does not exist, **FLUSH** functions like a **NOP**.

- **FLUSHINV** (data cache line flush and invalidate) causes the data cache to perform the same function as the **FLUSH** instruction and then invalidate the specified line in the cache. If the line is in the cache and dirty, the cache line is written out to external memory. The valid bit in the cache line is then cleared. If the line is not in the cache, **FLUSHINV** functions like a **NOP**.

If software requires synchronization with system hardware, place an **SSYNC** instruction after the **FLUSH** instruction to ensure that the flush operation has completed. If ordering is desired to ensure that previous stores have been pushed through all the queues, place an **SSYNC** instruction before the **FLUSH**.

**Data Cache Invalidation**

Besides the **FLUSHINV** instruction, explained in the previous section, two additional methods are available to invalidate the data cache when flushing is not required. The first technique directly invalidates valid bits by setting the invalid bit of each cache line to the invalid state. To implement this technique, additional MMRs (DTEST_COMMAND and DTEST_DATA[1:0]) are available to allow arbitrary reads/writes of all the cache entries directly. This method is explained in the next section.

For invalidating the complete data cache, a second method is available. By clearing the **DMC[1:0]** bits in the **DMEM_CONTROL** register (see Figure 6-10), all valid bits in the data cache are set to the invalid state. A second write to the **DMEM_CONTROL** register to set the **DMC[1:0]** bits to their previous state then configures the data memory back to its previous cache/SRAM configuration. An **SSYNC** instruction should be run before invalidating the cache and a **CSYNC** instruction should be inserted after each of these operations.
Data Test Registers

Like L1 instruction memory, L1 data memory contains additional MMRs to allow arbitrary reads/writes of all cache entries directly. The registers provide a mechanism for data cache test, initialization, and debug.

When the data test command register (DTEST_COMMAND) is written to, the L1 cache data or tag arrays are accessed and data is transferred through the data test data registers (DTEST_DATA[1:0]). The DTEST_DATA[1:0] registers contain the 64-bit data to be written, or they contain the destination for the 64-bit data read. The lower 32 bits are stored in the DTEST_DATA[0] register and the upper 32 bits are stored in the DTEST_DATA[1] register.
When the tag arrays are being accessed, then the DTEST_DATA[0] register is used.

A CSYNC instruction is required after writing the DTEST_COMMAND MMR.

Figure Figure 6-14 through Figure 6-16 describe the DTEST registers. Access to these registers is possible only in supervisor or emulation mode. When writing to DTEST registers, always write to the DTEST_DATA registers first, then the DTEST_COMMAND register.
Data Test Registers

Data Test Command (DTEST_COMMAND) Register

When the data test command register (DTEST_COMMAND) is written to, the L1 cache data or tag arrays are accessed, and the data is transferred through the data test data registers (DTEST_DATA[1:0]).

The data/instruction access bit allows direct access via the DTEST_COMMAND MMR to L1 instruction SRAM.

Figure 6-14. Data Test Command Register
Data Test Data (DTEST_DATA1) Register

Data test data registers (DTEST_DATA[1:0]) contain the 64-bit data to be written, or they contain the destination for the 64-bit data read. The data test data 1 register (DTEST_DATA1) stores the upper 32 bits.

Data Test Data 1 Register (DTEST_DATA1)

When accessing tag arrays, all bits are reserved.

Figure 6-15. Data Test Data 1 Register
Data Test Data (DTEST_DATA0) Register

The data test data 0 register (DTEST_DATA0) stores the lower 32 bits of the 64-bit data to be written, or it contains the lower 32 bits of the destination for the 64-bit data read. The DTEST_DATA0 register is also used to access the tag arrays and contains the valid and dirty bits, which indicate the state of the cache line.

![Data Test Data 0 Register (DTEST_DATA0)](image)

Used to access the L1 cache tag arrays. The address tag consists of the upper 18 bits and bit 11 of the physical address. See “Cache Lines” on page 6-11.

Figure 6-16. Data Test Data 0 Register
External Memory

The external memory space is shown in Figure 6-1. One of the memory regions is dedicated to SDRAM support. The size of the SDRAM bank is programmable and can range in size from 16M byte to 128M byte. The start address of the bank is 0x0000 0000.

Each of the next four banks contains 1M byte and is dedicated to support asynchronous memories. The start address of the asynchronous memory bank is 0x2000 0000. For the ADSP-BF538/ADSP-BF538F processors, the on-chip flash memory can be mapped to any of these four banks of asynchronous memory.

Memory Protection and Properties

This section describes the memory management unit (MMU), memory pages, CPLB management, MMU management, and CPLB registers.

Memory Management Unit

The Blackfin processor contains a page based memory management unit (MMU). This mechanism provides control over cacheability of memory ranges, as well as management of protection attributes at a page level. The MMU provides great flexibility in allocating memory and I/O resources between tasks, with complete control over access rights and cache behavior.

The MMU is implemented as two 16-entry content addressable memory (CAM) blocks. Each entry is referred to as a cacheability protection lookaside buffer (CPLB) descriptor. When enabled, every valid entry in the MMU is examined on any fetch, load, or store operation to determine whether there is a match between the address being requested and the page described by the CPLB entry. If a match occurs, the cacheability and
protection attributes contained in the descriptor are used for the memory transaction with no additional cycles added to the execution of the instruction.

Because the L1 memories are separated into instruction and data memories, the CPLB entries are also divided between instruction and data CPLBs. Sixteen CPLB entries are used for instruction fetch requests; these are called *ICPLBs*. Another sixteen CPLB entries are used for data transactions; these are called *DCPLBs*. The ICPLBs and DCPLBs are enabled by setting the appropriate bits in the L1 instruction memory control (*IMEM_CONTROL*) and L1 data memory control (*DMEM_CONTROL*) registers, respectively. These registers are shown in Figure 6-3 and Figure 6-10, respectively.

Each CPLB entry consists of a pair of 32-bit values. For instruction fetches:

- *ICPLB_ADDR[n]* defines the start address of the page described by the CPLB descriptor.
- *ICPLB_DATA[n]* defines the properties of the page described by the CPLB descriptor.

For data operations:

- *DCPLB_ADDR[m]* defines the start address of the page described by the CPLB descriptor.
- *DCPLB_DATA[m]* defines the properties of the page described by the CPLB descriptor.

There are two default CPLB descriptors for data accesses to the scratchpad data memory and to the system and core MMR space. These default descriptors define the above space as non-cacheable, so that additional CPLBs do not need to be set up for these regions of memory.

⚠️ If valid CPLBs are set up for this space, the default CPLBs are ignored.
Memory Pages

The 4G byte address space of the processor can be divided into smaller ranges of memory or I/O referred to as memory pages. Every address within a page shares the attributes defined for that page. The architecture supports four different page sizes:

- 1K byte
- 4K byte
- 1M byte
- 4M byte

Different page sizes provide a flexible mechanism for matching the mapping of attributes to different kinds of memory and I/O.

Memory Page Attributes

Each page is defined by a two-word descriptor, consisting of an address descriptor word xCPLB_ADDR[n] and a properties descriptor word xCPLB_DATA[n]. The address descriptor word provides the base address of the page in memory. Pages must be aligned on page boundaries that are an integer multiple of their size. For example, a 4M byte page must start on an address divisible by 4M byte; whereas a 1K byte page can start on any 1K byte boundary. The second word in the descriptor specifies the other properties or attributes of the page. These properties include:

- Page size
  1K byte, 4K byte, 1M byte, 4M byte
- Cacheable/non-cacheable
  Accesses to this page use the L1 cache or bypass the cache.
Memory Protection and Properties

- If cacheable: write-through/write-back
  Data writes propagate directly to memory or are deferred until the cache line is reallocated. If write-through, allocate on read only, or read and write.

- Dirty/modified
  The data in this page in memory has changed since the CPLB was last loaded.

- Supervisor write access permission
  – Enables or disables writes to this page when in Supervisor mode.
  – Data pages only.

- User write access permission
  – Enables or disables writes to this page when in User mode.
  – Data pages only.

- User read access permission
  Enables or disables reads from this page when in User mode.

- Valid
  Check this bit to determine whether this is valid CPLB data.

- Lock
  Keep this entry in MMR; do not participate in CPLB replacement policy.
Page Descriptor Table

For memory accesses to utilize the cache when CPLBs are enabled for instruction access, data access, or both, a valid CPLB entry must be available in an MMR pair. The MMR storage locations for CPLB entries are limited to 16 descriptors for instruction fetches and 16 descriptors for data load and store operations.

For small and/or simple memory models, it may be possible to define a set of CPLB descriptors that fit into these 32 entries, cover the entire addressable space, and never need to be replaced. This type of definition is referred to as a static memory management model.

However, operating environments commonly define more CPLB descriptors to cover the addressable memory and I/O spaces than will fit into the available on-chip CPLB MMRs. When this happens, a memory-based data structure, called a page descriptor table, is used; in it can be stored all the potentially required CPLB descriptors. The specific format for the page descriptor table is not defined as part of the Blackfin processor architecture. Different operating systems, which have different memory management models, can implement page descriptor table structures that are consistent with the OS requirements. This allows adjustments to be made between the level of protection afforded versus the performance attributes of the memory-management support routines.

CPLB Management

When the Blackfin processor issues a memory operation for which no valid CPLB (cacheability protection lookaside buffer) descriptor exists in an MMR pair, an exception occurs that places the processor into supervisor mode and vectors to the MMU exception handler (see “Exceptions” on page 4-46 for more information). The handler is typically part of the operating system (OS) kernel that implements the CPLB replacement policy.
Before CPLBs are enabled, valid CPLB descriptors must be in place for both the page descriptor table and the MMU exception handler. The \texttt{LOCK} bits of these CPLB descriptors are commonly set so they are not inadvertently replaced in software.

The handler uses the faulting address to index into the page descriptor table structure to find the correct CPLB descriptor data to load into one of the on-chip CPLB register pairs. If all on-chip registers contain valid CPLB entries, the handler selects one of the descriptors to be replaced, and the new descriptor information is loaded. Before loading new descriptor data into any CPLBs, the corresponding group of sixteen CPLBs must be disabled using:

- The enable DCPLB (\texttt{ENDCPLB}) bit in the \texttt{DMEM\_CONTROL} register for data descriptors, or
- The enable ICPLB (\texttt{ENICPLB}) bit in the \texttt{IMEM\_CONTROL} register for instruction descriptors

The CPLB replacement policy and algorithm to be used are the responsibility of the system MMU exception handler. This policy, which is dictated by the characteristics of the operating system, usually implements a modified LRU (least recently used) policy, a round robin scheduling method, or pseudo random replacement.

After the new CPLB descriptor is loaded, the exception handler returns, and the faulting memory operation is restarted. This operation should now find a valid CPLB descriptor for the requested address, and it should proceed normally.
A single instruction may generate an instruction fetch as well as one or two data accesses. It is possible that more than one of these memory operations references data for which there is no valid CPLB descriptor in an MMR pair. In this case, the exceptions are prioritized and serviced in this order:

- Instruction page miss
- A page miss on DAG0
- A page miss on DAG1

**MMU Application**

Memory management is an optional feature in the Blackfin processor architecture. Its use is predicated on the system requirements of a given application. Upon reset, all CPLBs are disabled, and the memory management unit (MMU) is not used.

If all L1 memory is configured as SRAM, then the data and instruction MMU functions are optional, depending on the application’s need for protection of memory spaces either between tasks or between User and Supervisor modes. To protect memory between tasks, the operating system can maintain separate tables of instruction and/or data memory pages available for each task and make those pages visible only when the relevant task is running. When a task switch occurs, the operating system can ensure the invalidation of any CPLB descriptors on chip that should not be available to the new task. It can also preload descriptors appropriate to the new task.

For many operating systems, the application program is run in user mode while the operating system and its services run in supervisor mode. It is desirable to protect code and data structures used by the operating system from inadvertent modification by a running user mode application. This protection can be achieved by defining CPLB descriptors for protected memory ranges that allow write access only when in supervisor mode. If a
write to a protected memory region is attempted while in user mode, an exception is generated before the memory is modified. Optionally, the user mode application may be granted read access for data structures that are useful to the application. Even supervisor mode functions can be blocked from writing some memory pages that contain code that is not expected to be modified. Because CPLB entries are MMRs that can be written only while in supervisor mode, user programs cannot gain access to resources protected in this way.

If either the L1 instruction memory or the L1 data memory is configured partially or entirely as cache, the corresponding CPLBs must be enabled. When an instruction generates a memory request and the cache is enabled, the processor first checks the ICPLBs to determine whether the address requested is in a cacheable address range. If no valid ICPLB entry in an MMR pair corresponds to the requested address, an MMU exception is generated to obtain a valid ICPLB descriptor to determine whether the memory is cacheable or not. As a result, if the L1 instruction memory is enabled as cache, then any memory region that contains instructions must have a valid ICPLB descriptor defined for it. These descriptors must either reside in MMRs at all times or be resident in a memory-based page descriptor table that is managed by the MMU exception handler. Likewise, if either or both L1 data banks are configured as cache, all potential data memory ranges must be supported by DCPLB descriptors.

Before caches are enabled, the MMU and its supporting data structures must be set up and enabled.
Examples of Protected Memory Regions

In Figure 6-17, a starting point is provided for basic CPLB allocation for Instruction and Data CPLBs. Note some ICPLBs and DCPLBs have common descriptors for the same address space.

Figure 6-17. Examples of Protected Memory Regions
Instruction CPLB Data (ICPLB_DATAx) Registers

Figure 6-18 describes the ICPLB data registers (ICPLB_DATAx).

To ensure proper behavior and future compatibility, all reserved bits in this register must be set to 0 whenever this register is written.

Table 6-4. ICPLB Data Register Memory-Mapped Addresses

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICPLB_DATA0</td>
<td>0xFFE0 1200</td>
</tr>
<tr>
<td>ICPLB_DATA1</td>
<td>0xFFE0 1204</td>
</tr>
<tr>
<td>ICPLB_DATA2</td>
<td>0xFFE0 1208</td>
</tr>
<tr>
<td>ICPLB_DATA3</td>
<td>0xFFE0 120C</td>
</tr>
<tr>
<td>ICPLB_DATA4</td>
<td>0xFFE0 1210</td>
</tr>
<tr>
<td>ICPLB_DATA5</td>
<td>0xFFE0 1214</td>
</tr>
<tr>
<td>ICPLB_DATA6</td>
<td>0xFFE0 1218</td>
</tr>
<tr>
<td>ICPLB_DATA7</td>
<td>0xFFE0 121C</td>
</tr>
<tr>
<td>ICPLB_DATA8</td>
<td>0xFFE0 1220</td>
</tr>
<tr>
<td>ICPLB_DATA9</td>
<td>0xFFE0 1224</td>
</tr>
<tr>
<td>ICPLB_DATA10</td>
<td>0xFFE0 1228</td>
</tr>
<tr>
<td>ICPLB_DATA11</td>
<td>0xFFE0 122C</td>
</tr>
<tr>
<td>ICPLB_DATA12</td>
<td>0xFFE0 1230</td>
</tr>
<tr>
<td>ICPLB_DATA13</td>
<td>0xFFE0 1234</td>
</tr>
<tr>
<td>ICPLB_DATA14</td>
<td>0xFFE0 1238</td>
</tr>
<tr>
<td>ICPLB_DATA15</td>
<td>0xFFE0 123C</td>
</tr>
</tbody>
</table>
Memory

ICPLB Data Registers (ICPLB_DATAx)

For memory-mapped addresses, see Table 6-4.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Reset = 0x0000 0000

PAGE_SIZE[1:0]
00 - 1K byte page size
01 - 4K byte page size
10 - 1M byte page size
11 - 4M byte page size

CPLB_L1_CHBL
Clear this bit whenever L1 memory is configured as SRAM
0 - Non-cacheable in L1
1 - Cacheable in L1

CPLB_LRPRI
See “Instruction Cache Locking by Line” on page 6-17.
0 - Low importance
1 - High importance

CPLB_USER_RD
0 - User mode read access generates protection violation exception
1 - User mode read access permitted

CPLB_VALID
0 - Invalid (disabled) CPLB entry
1 - Valid (enabled) CPLB entry

CPLB_LOCK
Can be used by software in CPLB replacement algorithms
0 - Unlocked, CPLB entry can be replaced
1 - Locked, CPLB entry should not be replaced

Figure 6-18. ICPLB Data Registers

Data CPLB Data (DCPLB_DATAx) Registers

Figure 6-19 shows the DCPLB data registers (DCPLB_DATAx).

To ensure proper behavior and future compatibility, all reserved bits in this register must be set to 0 whenever this register is written.
### Table 6-5. DCPLB Data Register Memory-Mapped Addresses

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCPLB_DATA0</td>
<td>0xFFE0 0200</td>
</tr>
<tr>
<td>DCPLB_DATA1</td>
<td>0xFFE0 0204</td>
</tr>
<tr>
<td>DCPLB_DATA2</td>
<td>0xFFE0 0208</td>
</tr>
<tr>
<td>DCPLB_DATA3</td>
<td>0xFFE0 020C</td>
</tr>
<tr>
<td>DCPLB_DATA4</td>
<td>0xFFE0 0210</td>
</tr>
<tr>
<td>DCPLB_DATA5</td>
<td>0xFFE0 0214</td>
</tr>
<tr>
<td>DCPLB_DATA6</td>
<td>0xFFE0 0218</td>
</tr>
<tr>
<td>DCPLB_DATA7</td>
<td>0xFFE0 021C</td>
</tr>
<tr>
<td>DCPLB_DATA8</td>
<td>0xFFE0 0220</td>
</tr>
<tr>
<td>DCPLB_DATA9</td>
<td>0xFFE0 0224</td>
</tr>
<tr>
<td>DCPLB_DATA10</td>
<td>0xFFE0 0228</td>
</tr>
<tr>
<td>DCPLB_DATA11</td>
<td>0xFFE0 022C</td>
</tr>
<tr>
<td>DCPLB_DATA12</td>
<td>0xFFE0 0230</td>
</tr>
<tr>
<td>DCPLB_DATA13</td>
<td>0xFFE0 0234</td>
</tr>
<tr>
<td>DCPLB_DATA14</td>
<td>0xFFE0 0238</td>
</tr>
<tr>
<td>DCPLB_DATA15</td>
<td>0xFFE0 023C</td>
</tr>
</tbody>
</table>
DCPLB Data Registers (DCPLB_DATAx)

For memory-mapped addresses, see Table 6-5.

- **PAGE_SIZE[1:0]**
  - 00 - 1K byte page size
  - 01 - 4K byte page size
  - 10 - 1M byte page size
  - 11 - 4M byte page size

- **CPLB_DIRTY**
  - Operates only in cache mode
  - 0 - Write back
  - 1 - Write through

- **CPLB_WT**
  - Operates only in cache mode
  - 0 - Write back
  - 1 - Write through

- **CPLB_L1_AOW**
  - Valid only if write through cacheable
  - 0 - Allocate cache lines on reads only
  - 1 - Allocate cache lines on reads and writes

- **CPLB_VALID**
  - 0 - Invalid (disabled) CPLB entry
  - 1 - Valid (enabled) CPLB entry

- **CPLB_LOCK**
  - Can be used by software in CPLB replacement algorithms
  - 0 - Unlocked, CPLB entry can be replaced
  - 1 - Locked, CPLB entry should not be replaced

- **CPLB_USER_RD**
  - 0 - User mode read access generates protection violation exception
  - 1 - User mode read access permitted

- **CPLB_USER_WR**
  - 0 - User mode write access generates protection violation exception
  - 1 - User mode write access permitted

- **CPLB_SUPV_WR**
  - 0 - Supervisor mode write access generates protection violation exception
  - 1 - Supervisor mode write access permitted

Figure 6-19. DCPLB Data Registers
Data CPLB Address (DCPLB_ADDRx) Registers

Figure 6-20 shows the DCPLB address registers (DCPLB_ADDRx).

Table 6-6. DCPLB Address Register Memory-Mapped Addresses

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCPLB_ADDR0</td>
<td>0xFFE0 0100</td>
</tr>
<tr>
<td>DCPLB_ADDR1</td>
<td>0xFFE0 0104</td>
</tr>
<tr>
<td>DCPLB_ADDR2</td>
<td>0xFFE0 0108</td>
</tr>
<tr>
<td>DCPLB_ADDR3</td>
<td>0xFFE0 010C</td>
</tr>
<tr>
<td>DCPLB_ADDR4</td>
<td>0xFFE0 0110</td>
</tr>
<tr>
<td>DCPLB_ADDR5</td>
<td>0xFFE0 0114</td>
</tr>
<tr>
<td>DCPLB_ADDR6</td>
<td>0xFFE0 0118</td>
</tr>
<tr>
<td>DCPLB_ADDR7</td>
<td>0xFFE0 011C</td>
</tr>
<tr>
<td>DCPLB_ADDR8</td>
<td>0xFFE0 0120</td>
</tr>
<tr>
<td>DCPLB_ADDR9</td>
<td>0xFFE0 0124</td>
</tr>
<tr>
<td>DCPLB_ADDR10</td>
<td>0xFFE0 0128</td>
</tr>
<tr>
<td>DCPLB_ADDR11</td>
<td>0xFFE0 012C</td>
</tr>
<tr>
<td>DCPLB_ADDR12</td>
<td>0xFFE0 0130</td>
</tr>
<tr>
<td>DCPLB_ADDR13</td>
<td>0xFFE0 0134</td>
</tr>
<tr>
<td>DCPLB_ADDR14</td>
<td>0xFFE0 0138</td>
</tr>
<tr>
<td>DCPLB_ADDR15</td>
<td>0xFFE0 013C</td>
</tr>
</tbody>
</table>
Instruction CPLB Address (ICPLB_ADDRx) Registers

Figure 6-21 shows the ICPLB Address registers (ICPLB_ADDRx).

Table 6-7. ICPLB Address Register Memory-Mapped Addresses

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICPLB_ADDR0</td>
<td>0xFFFFE0 1100</td>
</tr>
<tr>
<td>ICPLB_ADDR1</td>
<td>0xFFFFE0 1104</td>
</tr>
<tr>
<td>ICPLB_ADDR2</td>
<td>0xFFFFE0 1108</td>
</tr>
<tr>
<td>ICPLB_ADDR3</td>
<td>0xFFFFE0 1110C</td>
</tr>
<tr>
<td>ICPLB_ADDR4</td>
<td>0xFFFFE0 1110</td>
</tr>
<tr>
<td>ICPLB_ADDR5</td>
<td>0xFFFFE0 1114</td>
</tr>
<tr>
<td>ICPLB_ADDR6</td>
<td>0xFFFFE0 1118</td>
</tr>
<tr>
<td>ICPLB_ADDR7</td>
<td>0xFFFFE0 111C</td>
</tr>
<tr>
<td>ICPLB_ADDR8</td>
<td>0xFFFFE0 1120</td>
</tr>
<tr>
<td>ICPLB_ADDR9</td>
<td>0xFFFFE0 1124</td>
</tr>
<tr>
<td>ICPLB_ADDR10</td>
<td>0xFFFFE0 1128</td>
</tr>
</tbody>
</table>
Table 6-7. ICPLB Address Register Memory-Mapped Addresses (Cont’d)

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICPLB_ADDR11</td>
<td>0xFFE0 112C</td>
</tr>
<tr>
<td>ICPLB_ADDR12</td>
<td>0xFFE0 1130</td>
</tr>
<tr>
<td>ICPLB_ADDR13</td>
<td>0xFFE0 1134</td>
</tr>
<tr>
<td>ICPLB_ADDR14</td>
<td>0xFFE0 1138</td>
</tr>
<tr>
<td>ICPLB_ADDR15</td>
<td>0xFFE0 113C</td>
</tr>
</tbody>
</table>

ICPLB Address Registers (ICPLB_ADDRx)

For memory-mapped addresses, see Table 6-7.

Upper Bits of Address for Match[21:6]

Upper Bits of Address for Match[5:0]

Figure 6-21. ICPLB Address Registers
Instruction and Data CPLB Status (ICPLB_STATUS, DCPLB_STATUS) Registers

Bits in the DCPLB status register (DCPLB_STATUS) and ICPLB status register (ICPLB_STATUS) identify the CPLB entry that has triggered CPLB-related exceptions. The exception service routine can infer the cause of the fault by examining the CPLB entries.

The DCPLB_STATUS and ICPLB_STATUS registers are valid only while in the faulting exception service routine.

Bits FAULT_DAG, FAULT_USERSUPV and FAULT_RW in the DCPLB status register (DCPLB_STATUS) are used to identify the CPLB entry that has triggered the CPLB-related exception (see Figure 6-22).

**DCPLB Status Register (DCPLB_STATUS)**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-22</td>
<td>FAULT_IALLADDR</td>
</tr>
<tr>
<td>21-16</td>
<td>FAULT_DAG</td>
</tr>
<tr>
<td>15-8</td>
<td>FAULT_USERSUPV</td>
</tr>
<tr>
<td>7-0</td>
<td>FAULT_RW</td>
</tr>
</tbody>
</table>

**Figure 6-22. DCPLB Status Register**

Each bit indicates the hit/miss status of the associated CPLB entry.
Bit **FAULT_USERSUPV** in the ICPLB status register (**ICPLB_STATUS**) is used to identify the CPLB entry that has triggered the CPLB-related exception (see **Figure 6-23**).

**ICPLB Status Register (ICPLB_STATUS)**

The **DCPLB_FAULT_ADDR** and **ICPLB_FAULT_ADDR** registers hold the address that has caused a fault in the L1 data memory or L1 instruction memory, respectively. See **Figure 6-24** and **Figure 6-25**.

**Instruction and Data CPLB Fault Address (ICPLB_FAULT_ADDR, DCPLB_FAULT_ADDR) Registers**

The **DCPLB_FAULT_ADDR** and **ICPLB_FAULT_ADDR** registers are valid only while in the faulting exception service routine.

---

**Figure 6-23. ICPLB Status Register**

**Figure 6-24. DCPLB Fault Address**

**Figure 6-25. ICPLB Fault Address**
Memory

DCPLB Address Register (DCPLB_FAULT_ADDR)

![DCPLB Address Register Diagram]

**Reset = Undefined**

- **FAULT_ADDR[31:16]**
  Data address that has caused a fault in the L1 Data memory

- **FAULT_ADDR[15:0]**
  Data address that has caused a fault in the L1 Data memory

![DCPLB Address Register Diagram]

**Figure 6-24. DCPLB Address Register**

ICPLB Fault Address Register (ICPLB_FAULT_ADDR)

![ICPLB Fault Address Register Diagram]

**Reset = Undefined**

- **FAULT_ADDR[31:16]**
  Instruction address that has caused a fault in the L1 Instruction memory

- **FAULT_ADDR[15:0]**
  Instruction address that has caused a fault in the L1 Instruction memory

![ICPLB Fault Address Register Diagram]

**Figure 6-25. ICPLB Fault Address Register**
Both internal and external memory locations are accessed in little endian byte order. Figure 6-26 shows a data word stored in register \( R0 \) and in memory at address location \( addr \). B0 refers to the least significant byte of the 32-bit word.

Figure 6-26. Data Stored in Little Endian Order

Figure 6-27 shows 16- and 32-bit instructions stored in memory. The diagram on the left shows 16-bit instructions stored in memory with the most significant byte of the instruction stored in the high address (byte B1 in \( addr+1 \)) and the least significant byte in the low address (byte B0 in \( addr \)).

The diagram on the right shows 32-bit instructions stored in memory. Note the most significant 16-bit half word of the instruction (bytes B3 and B2) is stored in the low addresses (\( addr+1 \) and \( addr \)), and the least significant half word (bytes B1 and B0) is stored in the high addresses (\( addr+3 \) and \( addr+2 \)).

Figure 6-27. Instructions Stored in Little Endian Order
Load/Store Operation

The Blackfin processor architecture supports the RISC concept of a load/store machine. This machine is the characteristic in RISC architectures whereby memory operations (loads and stores) are intentionally separated from the arithmetic functions that use the targets of the memory operations. The separation is made because memory operations, particularly instructions that access off-chip memory or I/O devices, often take multiple cycles to complete and would normally halt the processor, preventing an instruction execution rate of one instruction per cycle.

Separating load operations from their associated arithmetic functions allows compilers or assembly language programmers to place unrelated instructions between the load and its dependent instructions. If the value is returned before the dependent operation reaches the execution stage of the pipeline, the operation completes in one cycle.

In write operations, the store instruction is considered complete as soon as it executes, even though many cycles may execute before the data is actually written to an external memory or I/O location. This arrangement allows the processor to execute one instruction per clock cycle, and it implies that the synchronization between when writes complete and when subsequent instructions execute is not guaranteed. Moreover, this synchronization is considered unimportant in the context of most memory operations.
Interlocked Pipeline

In the execution of instructions, the Blackfin processor architecture implements an interlocked pipeline. When a load instruction executes, the target register of the read operation is marked as busy until the value is returned from the memory system. If a subsequent instruction tries to access this register before the new value is present, the pipeline will stall until the memory operation completes. This stall guarantees that instructions that require the use of data resulting from the load do not use the previous or invalid data in the register, even though instructions are allowed to start execution before the memory read completes.

This mechanism allows the execution of independent instructions between the load and the instructions that use the read target without requiring the programmer or compiler to know how many cycles are actually needed for the memory-read operation to complete. If the instruction immediately following the load uses the same register, it simply stalls until the value is returned. Consequently, it operates as the programmer expects. However, if four other instructions are placed after the load but before the instruction that uses the same register, all of them execute, and the overall throughput of the processor is improved.

Ordering of Loads and Stores

The relaxation of synchronization between memory access instructions and their surrounding instructions is referred to as weak ordering of loads and stores. Weak ordering implies that the timing of the actual completion of the memory operations—even the order in which these events occur—may not align with how they appear in the sequence of the program source code. All that is guaranteed is:
- Load operations complete before the returned data is used by a subsequent instruction.
- Load operations using data previously written will use the updated values.
- Store operations eventually propagate to their ultimate destination.

Because of weak ordering, the memory system is allowed to prioritize reads over writes. In this case, a write that is queued anywhere in the pipeline, but not completed, may be deferred by a subsequent read operation, and the read is allowed to be completed before the write. Reads are prioritized over writes because the read operation has a dependent operation waiting on its completion, whereas the processor considers the write operation complete, and the write does not stall the pipeline if it takes more cycles to propagate the value out to memory. This behavior could cause a read that occurs in the program source code after a write in the program flow to actually return its value before the write has been completed. This ordering provides significant performance advantages in the operation of most memory instructions. However, it can cause side effects that the programmer must be aware of to avoid improper system operation.

When writing to or reading from non-memory locations such as I/O device registers, the order of how read and write operations complete is often significant. For example, a read of a status register may depend on a write to a control register. If the address is the same, the read would return a value from the write buffer rather than from the actual I/O device register, and the order of the read and write at the register may be reversed. Both these effects could cause undesirable side effects in the intended operation of the program and peripheral. To ensure that these effects do not occur in code that requires precise (strong) ordering of load and store operations, synchronization instructions (CSYNC or SSYNC) should be used.
Synchronizing Instructions

When strong ordering of loads and stores is required, as may be the case for sequential writes to an I/O device for setup and control, use the core or system synchronization instructions, \texttt{CSYNC} or \texttt{SSYNC}, respectively.

The \texttt{CSYNC} instruction ensures all pending core operations have completed and the core buffer (between the processor core and the L1 memories) has been flushed before proceeding to the next instruction. Pending core operations may include any pending interrupts, speculative states (such as branch predictions), or exceptions.

Consider the following example code sequence:

\begin{verbatim}
IF CC JUMP away_from_here
  csync;
  r0 = [p0];
away_from_here:
\end{verbatim}

In the preceding example code, the \texttt{CSYNC} instruction ensures:

- The conditional branch (\texttt{IF CC JUMP away_from_here}) is resolved, forcing stalls into the execution pipeline until the condition is resolved and any entries in the processor store buffer have been flushed.

- All pending interrupts or exceptions have been processed before \texttt{CSYNC} completes.

- The load is not fetched from memory speculatively.

The \texttt{SSYNC} instruction ensures that all side effects of previous operations are propagated out through the interface between the L1 memories and the rest of the chip. In addition to performing the core synchronization functions of \texttt{CSYNC}, the \texttt{SSYNC} instruction flushes any write buffers.
between the L1 memory and the system domain and generates a sync request to the system that requires acknowledgement before SSYNC completes.

**Speculative Load Execution**

Load operations from memory do not change the state of the memory value. Consequently, issuing a speculative memory-read operation for a subsequent load instruction usually has no undesirable side effect. In some code sequences, such as a conditional branch instruction followed by a load, performance may be improved by speculatively issuing the read request to the memory system before the conditional branch is resolved.

For example,

```plaintext
IF CC JUMP away_from_here
RO = [P2]:
...
away_from_here:
```

If the branch is taken, then the load is flushed from the pipeline, and any results that are in the process of being returned can be ignored. Conversely, if the branch is not taken, the memory will have returned the correct value earlier than if the operation were stalled until the branch condition was resolved.

However, in the case of an I/O device, this could cause an undesirable side effect for a peripheral that returns sequential data from a FIFO or from a register that changes value based on the number of reads that are requested. To avoid this effect, use synchronizing instructions (CSYNC or SSYNC) to guarantee the correct behavior between read operations.

Store operations never access memory speculatively, because this could cause modification of a memory value before it is determined whether the instruction should have executed.
Conditional Load Behavior

The synchronization instructions force all speculative states to be resolved before a load instruction initiates a memory reference. However, the load instruction itself may generate more than one memory-read operation, because it is interruptible. If an interrupt of sufficient priority occurs between the completion of the synchronization instruction and the completion of the load instruction, the sequencer cancels the load instruction. After execution of the interrupt, the interrupted load is executed again. This approach minimizes interrupt latency. However, it is possible that a memory-read cycle was initiated before the load was canceled, and this would be followed by a second read operation after the load is executed again. For most memory accesses, multiple reads of the same memory address have no side effects. However, for some memory-mapped devices, such as peripheral data FIFOs, reads are destructive. Each time the device is read, the FIFO advances, and the data cannot be recovered and re-read.

When accessing memory-mapped devices that have state dependencies on the number of read or write operations on a given address location, disable interrupts before performing the load or store operation.

Working With Memory

This section contains information about alignment of data in memory and memory operations that support semaphores between tasks. It also contains a brief discussion of MMR registers and a core MMR programming example.

Alignment

Nonaligned memory operations are not directly supported. A nonaligned memory reference generates a misaligned access exception event (see “Exceptions” on page 4-46). However, because some data streams (such as
8-bit video data) can properly be nonaligned in memory, alignment exceptions may be disabled by using the `DISALGNEXCPT` instruction. Moreover, some instructions in the quad 8-bit group automatically disable alignment exceptions.

## Cache Coherency

For shared data, software must provide cache coherency support as required. To accomplish this, use the `FLUSH` instruction (see “Data Cache Control Instructions” on page 6-37), and/or explicit line invalidation through the core MMRs (see “Data Test Registers” on page 6-39).

## Atomic Operations

The processor provides a single atomic operation: `TESTSET`. Atomic operations are used to provide non interruptible memory operations in support of semaphores between tasks. The `TESTSET` instruction loads an indirectly addressed memory half word, tests whether the low byte is zero, and then sets the most significant bit (MSB) of the low memory byte without affecting any other bits. If the byte is originally zero, the instruction sets the CC bit. If the byte is originally nonzero, the instruction clears the CC bit. The sequence of this memory transaction is atomic—hardware bus locking insures that no other memory operation can occur between the test and set portions of this instruction. The `TESTSET` instruction can be interrupted by the core. If this happens, the `TESTSET` instruction is executed again upon return from the interrupt.

The `TESTSET` instruction can address the entire 4G byte memory space, but should not target on-core memory (L1 or MMR space) since atomic access to this memory is not supported.

The memory architecture always treats atomic operations as cache inhibited accesses even if the CPLB descriptor for the address indicates cache enabled access. However, executing `TESTSET` operations on cacheable
regions of memory is not recommended since the architecture cannot guarantee a cacheable location of memory is coherent when the TESTSET instruction is executed.

Memory-Mapped Registers

The MMR reserved space is located at the top of the memory space (0xFFC0 0000). This region is defined as non-cacheable and is divided between the system MMRs (0xFFC0 0000–0xFFE0 0000) and core MMRs (0xFFE0 0000–0xFFFF FFFF).

If strong ordering is required, place a synchronization instruction after stores to MMRs. For more information, see “Load/Store Operation” on page 6-63.

All MMRs are accessible only in supervisor mode. Access to MMRs in user mode generates a protection violation exception. Attempts to access MMR space using DAG1 will also generate a protection violation exception.

All core MMRs are read and written using 32-bit aligned accesses. However, some MMRs have fewer than 32 bits defined. In this case, the unused bits are reserved. System MMRs may be 16 bits.

Accesses to nonexistent MMRs generate an illegal access exception. The system ignores writes to read-only MMRs.

Appendix A, “Blackfin Processor Core MMR Assignments” provides a summary of all core MMRs. Appendix B, “System MMR Assignments” provides a summary of all system MMRs.

Core MMR Programming Code Example

Core MMRs may be accessed only as aligned 32-bit words. Nonaligned access to MMRs generates an exception event. Listing 6-1 shows the instructions required to manipulate a generic core MMR.
Listing 6-1. Core MMR Programming

CLI R0;          /* stop interrupts and save IMASK */
P0 = MMR_BASE;   /* 32-bit instruction to load base of MMRs */
R1 = [P0 + TIMER_CONTROL_REG];   /* get value of control reg */
BITSET R1, #N;                   /* set bit N */
[P0 + TIMER_CONTROL_REG] = R1;   /* restore control reg */
CSYNC;            /* assures that the control reg is written */
STI R0;                           /* enable interrupts */

The CLI instruction saves the contents of the IMASK register and disables interrupts by clearing IMASK. The STI instruction restores the contents of the IMASK register, thus enabling interrupts. The instructions between CLI and STI are not interruptible.

Terminology

The following terminology is used to describe memory.

cache block. The smallest unit of memory that is transferred to/from the next level of memory from/to a cache as a result of a cache miss.

cache hit. A memory access that is satisfied by a valid, present entry in the cache.

cache line. Same as cache block. In this chapter, cache line is used for cache block.

cache miss. A memory access that does not match any valid entry in the cache.

direct-mapped. Cache architecture in which each line has only one place in which it can appear in the cache. Also described as 1-way associative.
dirty or modified. A state bit, stored along with the tag, indicating whether the data in the data cache line has been changed since it was copied from the source memory and, therefore, needs to be updated in that source memory.

exclusive, clean. The state of a data cache line, indicating that the line is valid and that the data contained in the line matches that in the source memory. The data in a clean cache line does not need to be written to source memory before it is replaced.

fully associative. Cache architecture in which each line can be placed anywhere in the cache.

index. Address portion that is used to select an array element (for example, a line index).

invalid. Describes the state of a cache line. When a cache line is invalid, a cache line match cannot occur.

least recently used (LRU) algorithm. Replacement algorithm, used by cache, that first replaces lines that have been unused for the longest time.

Level 1 (L1) memory. Memory that is directly accessed by the core with no intervening memory subsystems between it and the core.

little endian. The native data store format of the Blackfin processor. Words and half words are stored in memory (and registers) with the least significant byte at the lowest byte address and the most significant byte in the highest byte address of the data storage location.

replacement policy. The function used by the processor to determine which line to replace on a cache miss. Often, an LRU algorithm is employed.

set. A group of N-line storage locations in the ways of an N-Way cache, selected by the INDEX field of the address (see Figure 6-5).
set associative. Cache architecture that limits line placement to a number of sets (or ways).

tag. Upper address bits, stored along with the cached data line, to identify the specific address source in memory that the cached line represents.

valid. A state bit, stored with the tag, indicating that the corresponding tag and data are current and correct and can be used to satisfy memory access requests.

victim. A dirty cache line that must be written to memory before it can be replaced to free space for a cache line allocation.

Way. An array of line storage elements in an $N$-Way cache (see Figure 6-5).

write back. A cache write policy, also known as copyback. The write data is written only to the cache line. The modified cache line is written to source memory only when it is replaced. Cache lines are allocated on both reads and writes.

write through. A cache write policy (also known as store through). The write data is written to both the cache line and to the source memory. The modified cache line is not written to the source memory when it is replaced. Cache lines must be allocated on reads, and may be allocated on writes (depending on mode).
Terminology
7 CHIP BUS HIERARCHY

This chapter discusses the on-chip buses, including how data moves through the system and factors that determine the system organization. The chapter describes the system internal chip interfaces and discusses the system interconnects and the associated system buses.

Internal Interfaces

Figure 7-1 shows the core processor and system boundaries and the interfaces between them.

Internal Clocks

The core processor clock (CCLK) rate is highly programmable with respect to CLKIN. The CCLK rate is divided down from the PLL output rate. This divider ratio is set using the CSEL parameter of the PLL divide register.

The peripheral access bus (PAB), the DMA access buses (DAB0/DAB1), the external access bus (EAB), the DMA core buses (DCB0–2), the DMA external bus (DEB), the external port bus (EPB), and the external bus interface unit (EBIU) run at the system clock frequency (SCLK domain). This divider ratio is set using the SSEL parameter of the PLL divide register and must be set so that these buses run as specified in ADSP-BF538/ADSP-BF538F Embedded Processor Data Sheet, and slower than or equal to the core clock frequency.
These buses can also be cycled at a programmable frequency to reduce power consumption, or to allow the core processor to run at an optimal frequency. Note all synchronous peripherals derive their timing from the SCLK. For example, the UART clock rates are determined by further dividing this clock frequency.
Core Overview

For the purposes of this discussion, Level 1 memories (L1) are included in the description of the core; they have full bandwidth access from the processor core with a 64-bit instruction bus and two 32-bit data buses.

Figure 7-2 is a block diagram that shows the core processor and its interfaces to the peripherals and external memory resources.
The core can generate up to three simultaneous off-core accesses per cycle.

The core bus structure between the processor and L1 memory runs at the full core frequency and has data paths up to 64 bits.

When the instruction request is filled, the 64-bit read can contain a single 64-bit instruction or any combination of 16-, 32-, or partial 64-bit instructions.

When cache is enabled, four 64-bit read requests are issued to support 32-byte line fill burst operations. These requests are pipelined so that each transfer after the first is filled in a single, consecutive cycle.

**System Overview**

The system includes the controllers for system interrupts, test/emulation, and clock and power management. Synchronous clock-domain conversion is provided to support clock domain transactions between the core and the system.

**System Interfaces**

The processor system includes the peripheral set—timers, real time clock, general-purpose I/O, UARTs, SPORTs, PPI, watchdog timer, SPIs, TWIs, and CAN. The processor system also includes the external memory controller (EBIU), the DMA controllers, and the interfaces between these units, the system, and the optional external (off-chip) resources. See Figure 7-2.
The following sections describe the six on-chip interfaces between the system and the peripherals:

- “Peripheral Access Bus (PAB)” on page 7-5
- “DMA Access (DAB0/DAB1), Core (DCB0/DCB1), and External Buses (DEB0/DEB1)” on page 7-7
- “External Access Bus (EAB)” on page 7-10

The external bus interface unit (EBIU) is the primary chip pin bus. For more information, see “External Bus Interface Unit” on page 18-1.

**Peripheral Access Bus (PAB)**

The processor has a dedicated peripheral bus. A low latency peripheral bus keeps core stalls to a minimum and allows for manageable interrupt latencies to time-critical peripherals. All peripheral resources accessed through the PAB are mapped into the system MMR space of the processor memory map. The core can access system MMR space through the PAB bus.

The core processor has byte addressability, but the programming model is restricted to only 32-bit (aligned) access to the system MMRs. Byte access to this region is not supported. Also, the `TESTSET` instruction to system memory mapped register space is not supported, since `TESTSET` produces byte-size read-modify-write.

**PAB Arbitration**

The core is the only master on this bus. No arbitration is necessary.

**PAB Performance**

For the PAB, the primary performance criteria is latency, not throughput. Transfer latencies for both read and write transfers on the PAB are 2 SCLK cycles.
For example, the core can transfer up to 32 bits per access to the PAB slaves. With the core clock running at 2 times the frequency of the system clock, the first and subsequent system MMR read or write accesses take 4 core clocks (CCLK) of latency.

The PAB has a maximum frequency of SCLK.

**PAB Agents (Masters, Slaves)**

The processor core can master bus operations on the PAB. All peripherals have a peripheral bus slave interface which allows the core to access control and status state. These registers are mapped into the system MMR space of the memory map. Appendix B, “System MMR Assignments”.

The slaves on the PAB bus are as follows:

- Event controller
- Clock and power management controller
- Watchdog timer
- Real time clock
- Timer0, 1, and 2
- SPORT0–3
- SPI0–2
- General-purpose I/O
- UART0–2
- PPI
- TWI0–1
- CAN
Chip Bus Hierarchy

- Asynchronous memory controller (AMC)
- SDRAM controller (SDC)
- DMA controller 0
- DMA controller 1

DMA Access (DAB0/DAB1), Core (DCB0/DCB1), and External Buses (DEB0/DEB1)

The DABx, DCBx, and DEBx buses provide a means for DMA capable peripherals to gain access to on-chip and off-chip memory with little or no degradation in core bandwidth to memory.

DABx, DCBx, and DEBx Arbitration

There are 13 DMA capable peripherals in the processor system, including both memory DMA controllers. 28 DMA channels and bus masters support these devices. The peripheral DMA controllers can transfer data between peripherals and internal or external memory.

The DAB buses are implemented as two separate bus systems each interfacing to a DMA controller and a fixed set of peripheral DMA bus masters. Each of the two DMA controllers access L1 memory through the DCB buses. In the event of simultaneous requests to L1 memory, access is granted to DMA controller 0, with DMA controller 1 having the lower priority. Each of the two DMA controllers access external memory through the DEB buses. In the event of simultaneous requests to external memory, access is granted to DMA controller 0 first. This fixed priority arrangement should be considered as each of the application specific interfaces are assigned to each peripheral.
Table 7-1 and Table 7-2 show the default arbitration priority of each DAB bus.

Table 7-1. Controller 0 (DAB0) Arbitration Priority

<table>
<thead>
<tr>
<th>DAB, DCB, DEB Master</th>
<th>Default Arbitration Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPI</td>
<td>0 - highest</td>
</tr>
<tr>
<td>SPORT0 RCV DMA controller</td>
<td>1</td>
</tr>
<tr>
<td>SPORT1 RCV DMA controller</td>
<td>3</td>
</tr>
<tr>
<td>SPORT0 XMT DMA controller</td>
<td>2</td>
</tr>
<tr>
<td>SPORT1 XMT DMA controller</td>
<td>4</td>
</tr>
<tr>
<td>SPI0 DMA controller</td>
<td>5</td>
</tr>
<tr>
<td>UART0 RCV controller</td>
<td>6</td>
</tr>
<tr>
<td>UART0 XMT controller</td>
<td>7</td>
</tr>
<tr>
<td>Memory DMA0 (dest) controller</td>
<td>8</td>
</tr>
<tr>
<td>Memory DMA0 (source) controller</td>
<td>9</td>
</tr>
<tr>
<td>Memory DMA1 (dest) controller</td>
<td>10</td>
</tr>
<tr>
<td>Memory DMA1 (source) controller</td>
<td>11 - lowest</td>
</tr>
</tbody>
</table>

Table 7-2. Controller 1 (DAB1) Arbitration Priority

<table>
<thead>
<tr>
<th>DAB, DCB, DEB Master</th>
<th>Default Arbitration Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPORT2 RCV DMA controller</td>
<td>0 - highest</td>
</tr>
<tr>
<td>SPORT2 XMT DMA controller</td>
<td>1</td>
</tr>
<tr>
<td>SPORT3 RCV DMA controller</td>
<td>2</td>
</tr>
<tr>
<td>SPORT3 XMT DMA controller</td>
<td>3</td>
</tr>
<tr>
<td>Unassigned</td>
<td>4</td>
</tr>
<tr>
<td>Unassigned</td>
<td>5</td>
</tr>
<tr>
<td>SPI1 DMA controller</td>
<td>6</td>
</tr>
<tr>
<td>SPI2 DMA controller</td>
<td>7</td>
</tr>
</tbody>
</table>
The DCB has priority over the core processor on arbitration into L1 configured as data SRAM, whereas the core processor has priority over the DCB on arbitration into L1 instruction SRAM. For off-chip memory, the core has priority over the DEB buses on the EAB bus.

### DAB, DCB, and DEB Performance

The processor DAB supports data transfers of 16 bits or 32 bits. The data bus has a 16-bit width with a maximum frequency as specified in *ADSP-BF538/ADSP-BF538F Embedded Processor Data Sheet*.

The DCB has a dedicated port into L1 memory. No stalls occur as long as the core access and the DMA access are not to the same memory bank (4 KB size for L1). If there is a conflict when accessing data memory, DMA is the highest priority requester, followed by the core. If the conflict occurs when accessing instruction memory, the core is the highest priority requester, followed by DMA.

Note that a locked transfer by the core processor (for example, execution of a `TESTSET` instruction) effectively disables arbitration for the addressed memory bank or resource until the memory lock is deasserted. DMA controllers cannot perform locked transfers.
DMA access to L1 memory can only be stalled by an access already in progress from another DMA channel. These latencies caused by these stalls are in addition to any arbitration latencies.

The core processor and the DMA controllers must arbitrate for access to external memory through the EBIU. This additional arbitration latency added to the latency required to read off-chip memory devices can significantly degrade DEB throughput, potentially causing peripheral data buffers to underflow or overflow. If you use DMA peripherals other than the memory DMA controller, and you target external memory for DMA accesses, you need to carefully analyze your specific traffic patterns to ensure that those isochronous peripherals targeting internal memory have enough allocated bandwidth and the appropriate maximum arbitration latencies.

**DAB Bus Agents (Masters)**

All peripherals capable of sourcing a DMA access are masters on this bus, as shown in Table 7-1. A single arbiter supports a programmable priority arbitration policy for access to the DAB.

When two or more DMA master channels are actively requesting the DAB, bus utilization is considerably higher due to the DAB’s pipelined design. Bus arbitration cycles are concurrent with the previous DMA access data cycles.

**External Access Bus (EAB)**

The EAB provides a way for the processor core and the memory DMA controller to directly access off-chip memory and high throughput memory-to-memory DMA transfers.
EAB Arbitration

Arbitration for use of the external port bus interface resources is required because of possible contention between the potential masters of this bus. A fixed-priority arbitration scheme is used.

EAB Performance

The EAB supports single word accesses of either 8-bit or 16-bit data types. The EAB operates at the same frequency as the PAB and the DAB, up to the maximum $\text{SCLK}$ frequency specified in *ADSP-BF538/ADSP-BF538F Embedded Processor Data Sheet*.

Memory DMA transfers typically result in repeated accesses to the same memory location. Because the memory DMA controller has the potential of simultaneously accessing on-chip and off-chip memory, considerable throughput can be achieved. The throughput rate for an on-chip/off-chip memory access is limited by the slower of the two accesses. An additional 1 to 2 cycles per burst access is inherent in the design.

In the case where the transfer is from on-chip to on-chip memory or from off-chip to off-chip memory, the burst accesses cannot occur simultaneously. The transfer rate is then determined by adding each transfer plus and additional cycle between each transfer.

Table 7-3 shows many types of 16-bit memory DMA transfers. In the table, it is assumed that no other DMA activity is conflicting with ongoing operations.
Table 7-3. EAB Performance

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>Approximate SCLKs For n Words (from start of DMA to interrupt at end)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-bit SDRAM</td>
<td>L1 data memory</td>
<td>n + 14</td>
</tr>
<tr>
<td>L1 data memory</td>
<td>16-bit SDRAM</td>
<td>n + 11</td>
</tr>
<tr>
<td>16-bit async memory</td>
<td>L1 data memory</td>
<td>n + 12</td>
</tr>
<tr>
<td>L1 data memory</td>
<td>16-bit async memory</td>
<td>n + 9</td>
</tr>
<tr>
<td>16-bit SDRAM</td>
<td>16-bit SDRAM</td>
<td>10 + (17n/7)</td>
</tr>
<tr>
<td>16-bit async memory</td>
<td>16-bit async memory</td>
<td>10 + 2xn, where x is the number of wait states + setup/hold SCLK cycles (minimum x=2)</td>
</tr>
<tr>
<td>L1 data memory</td>
<td>L1 data memory</td>
<td>2n + 12</td>
</tr>
</tbody>
</table>
This chapter describes the dynamic power management functionality of the processor. This functionality includes:

- Clocking
- Phase-locked loop (PLL)
- Dynamic power management controller
- Operating modes
- Voltage control

**Clocking**

The input clock into the processor, \texttt{CLKIN}, provides the necessary clock frequency, duty cycle, and stability to allow accurate internal clock multiplication by means of an on-chip phase-locked loop (PLL) module. During normal operation, the user programs the PLL with a multiplication factor for \texttt{CLKIN}. The resulting, multiplied signal is the voltage controlled oscillator (\texttt{VCO}) clock. A user-programmable value then divides the \texttt{VCO} clock signal to generate the core clock (\texttt{CCLK}).

A user-programmable value divides the \texttt{VCO} signal to generate the system clock (\texttt{SCLK}). The \texttt{SCLK} signal clocks the peripheral access bus (PAB), DMA bus (DAB), external address bus (EAB), and the external bus interface Unit (EBIU).
These buses run at the PLL frequency divided by 1–15 (SCLK domain). Using the SSEL parameter of the PLL divide register, select a divider value that allows these buses to run at or below the maximum SCLK rate specified in ADSP-BF538/ADSP-BF538F Embedded Processor Data Sheet.

To optimize performance and power dissipation, the processor allows the core and system clock frequencies to be changed dynamically in a coarse adjustment. For a fine adjustment, the PLL clock frequency can also be varied.

**Phase-Locked Loop and Clock Control**

To provide the clock generation for the core and system, the processor uses an analog PLL with programmable state machine control.

The PLL design serves a wide range of applications. It emphasizes embedded and portable applications and low cost, general-purpose processors, in which performance, flexibility, and control of power dissipation are key features. This broad range of applications requires a wide range of frequencies for the clock generation circuitry. The input clock may be a crystal, a crystal oscillator, or a buffered, shaped clock derived from an external system clock oscillator.

The PLL interacts with the dynamic power management controller (DPMC) block to provide power management functions for the processor. For information about the DPMC, see “Dynamic Power Management Controller” on page 8-11.

**PLL Overview**

Subject to the maximum VCO frequency, the PLL supports a wide range of multiplier ratios and achieves multiplication of the input clock, CLkin. To achieve this wide multiplication range, the processor uses a combination of programmable dividers in the PLL feedback circuit and output configuration blocks.
Figure 8-1 illustrates a conceptual model of the PLL circuitry, configuration inputs, and resulting outputs. In the figure, the VCO is an intermediate clock from which the core clock (CCLK) and system clock (SCLK) are derived.

![PLL Block Diagram](image)

**Figure 8-1. PLL Block Diagram**

### PLL Clock Multiplier Ratios

The PLL control register (PLL_CTL) governs the operation of the PLL. For details about the PLL_CTL register, see “PLL Control (PLL_CTL) Register” on page 8-7.

The divide frequency (DF) bit and multiplier select (MSEL[5:0]) field configure the various PLL clock dividers:

- **DF** enables the input divider
- **MSEL[5:0]** controls the feedback dividers

The reset value of MSEL is 0xA. This value can be reprogrammed at startup in the boot code.

*Table 8-1* illustrates the VCO multiplication factors for the various MSEL and DF settings.
Clocking

As shown in the table, different combinations of $MSEL[5:0]$ and $DF$ can generate the same $VCO$ frequencies. For a given application, one combination may provide lower power or satisfy the $VCO$ maximum frequency. Under normal conditions, setting $DF$ to 1 typically results in lower power dissipation. See *ADSP-BF538/ADSP-BF538F Embedded Processor Data Sheet* for maximum and minimum frequencies for $CLKIN$, $CCLK$, and $VCO$.

Table 8-1. MSEL Encodings

<table>
<thead>
<tr>
<th>Signal name</th>
<th>VCO Frequency $DF = 0$</th>
<th>VCO Frequency $DF = 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$MSEL[5:0]$</td>
<td>64x</td>
<td>32x</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>0.5x</td>
</tr>
<tr>
<td>1</td>
<td>2x</td>
<td>1x</td>
</tr>
<tr>
<td>$N = 3–62$</td>
<td>Nx</td>
<td>0.5Nx</td>
</tr>
<tr>
<td>63</td>
<td>63x</td>
<td>31.5x</td>
</tr>
</tbody>
</table>

Core Clock/System Clock Ratio Control

Table 8-2 describes the programmable relationship between the $VCO$ frequency and the core clock. Table 8-3 shows the relationship of the $VCO$ frequency to the system clock. Note the divider ratio must be chosen to limit the $SCLK$ to a frequency specified in *ADSP-BF538/ADSP-BF538F Embedded Processor Data Sheet*. The $SCLK$ drives all synchronous, system-level logic.

The divider ratio control bits, $CSEL$ and $SSEL$, are in the PLL divide register ($PLL\_DIV$). For information about this register, see “PLL Divide ($PLL\_DIV$) Register” on page 8-6. Appendix B, “System MMR Assignments”, shows the register addresses.

The reset value of $CSEL[1:0]$ is 0x0 (/1), and the reset value of $SSEL[3:0]$ is 0x5. These values can be reprogrammed at startup by the boot code.
By writing the appropriate value to `PLL_DIV`, you can change the `CSEL` and `SSEL` value dynamically. Note the divider ratio of the core clock can never be greater than the divider ratio of the system clock. If the `PLL_DIV` register is programmed to illegal values, the `SCLK` divider is automatically increased to be greater than or equal to the core clock divider.

The `PLL_DIV` register can be programmed at any time to change the `CCLK` and `SCLK` divide values without entering the Idle state.

Table 8-2. Core Clock Ratio

<table>
<thead>
<tr>
<th>Signal Name CSEL[1:0]</th>
<th>Divider Ratio VCO/CCLK</th>
<th>Example Frequency Ratios (MHz) VCO</th>
<th>CCLK</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>300</td>
<td>300</td>
</tr>
<tr>
<td>01</td>
<td>2</td>
<td>600</td>
<td>300</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>600</td>
<td>150</td>
</tr>
<tr>
<td>11</td>
<td>8</td>
<td>400</td>
<td>50</td>
</tr>
</tbody>
</table>

As long as the `MSEL` and `DF` control bits in the PLL control register (`PLL_CTL`) remain constant, the PLL is locked.

Table 8-3. System Clock Ratio

<table>
<thead>
<tr>
<th>Signal Name SSEL[3:0]</th>
<th>Divider Ratio VCO/SCLK</th>
<th>Example Frequency Ratios (MHz) VCO</th>
<th>SCLK</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>Reserved</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>0001</td>
<td>1:1</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>0010</td>
<td>2:1</td>
<td>200</td>
<td>100</td>
</tr>
<tr>
<td>0011</td>
<td>3:1</td>
<td>400</td>
<td>133</td>
</tr>
<tr>
<td>0100</td>
<td>4:1</td>
<td>500</td>
<td>125</td>
</tr>
<tr>
<td>0101</td>
<td>5:1</td>
<td>600</td>
<td>120</td>
</tr>
<tr>
<td>0110</td>
<td>6:1</td>
<td>600</td>
<td>100</td>
</tr>
<tr>
<td>N = 7–15</td>
<td>N:1</td>
<td>600</td>
<td>600/N</td>
</tr>
</tbody>
</table>
Clocking

If changing the clock ratio via writing a new SSEL value into PLL_DIV, take care that the enabled peripherals do not suffer data loss due to SCLK frequency changes.

PLL Registers

The user interface to the PLL is through four memory-mapped registers (MMRs):

- The PLL divide register (PLL_DIV)
- The PLL control register (PLL_CTL)
- The PLL status register (PLL_STAT)
- The PLL lock count register (PLL_LOCKCNT)

All four registers are 16-bit MMRs and must be accessed with aligned 16-bit reads/writes.

PLL Divide (PLL_DIV) Register

The PLL divide register (PLL_DIV) divides the PLL output clock to create the processor core clock (CCLK) and the system clock (SCLK). These values can be independently changed during processing to reduce power dissipation without changing the PLL state. The only restrictions are the resulting CCLK frequency must be greater than or equal to the SCLK frequency, and SCLK must fall within the allowed range specified in ADSP-BF538/ADSP-BF538F Embedded Processor Data Sheet. If the CCLK and SCLK divide values are programmed otherwise, the SCLK value is automatically adjusted to be slower than or equal to the core clock. Figure 8-2 shows the bits in the PLL_DIV register.
Dynamic Power Management

PLL Divide Register (PLL_DIV)

![PLL Divide Register Diagram](image)

PLL Control (PLL_CTL) Register

The PLL control register (PLL_CTL) controls operation of the PLL (see Figure 8-3). Note changes to the PLL_CTL register do not take effect immediately. In general, the PLL_CTL register is first programmed with new values, and then a specific PLL programming sequence must be executed to implement the changes. See “PLL Programming Sequence” on page 8-19.

![PLL Control Register Diagram](image)

Figure 8-2. PLL Divide Register

Figure 8-3. The PLL Control Register
Clocking

The following fields of the PLL_CTL register are used to control the PLL:

- **MSEL[5:0]** – The multiplier select (MSEL) field defines the input clock to VCO clock (CLKIN to VCO) multiplier.

- **BYPASS** – This bit is used to bypass the PLL. When BYPASS is set, CLKin is passed directly to the core and peripheral clocks.

- **OUTDELAY[1:0]** – These bits are used to adjust when the external memory output signals (address, write data, and control) transition with respect to CLKOUT. The 00 encoding of the OUTDELAY field represents the nominal output delay. The memory output signals can have 200 ps or 400 ps delay added to the nominal output delay or can have 200 ps of delay subtracted from the nominal output delay. The default value for OUTDELAY field is 01 selecting 400 ps of delay to be added to the nominal output delay.

- **INDELAY[1:0]** – These bits are used to adjust when the external memory input signals (read data) are sampled with respect to CLKOUT. The 00 encoding of INDELAY field selects the nominal sample point. The sample point of the memory input signals can be either be delayed from the nominal sample point by 200 ps or can be sampled 200 ps or 400 ps earlier than the nominal sample point. The default value for INDELAY field is 00 selecting the nominal sample point.

- **PDWN** – The power down (PDWN) bit is used to place the processor in the deep sleep operating mode.

For information about operating modes, see “Operating Modes” on page 8-11.

- **STOPCK** – The stop clock (STOPCK) bit is used to enable/disable the core clock, CCLK.
Dynamic Power Management

- **PLL_OFF** – This bit is used to enable/disable power to the PLL.

- **DF** – The divide frequency (DF) bit controls the PLL input clock divider which determines whether the PLL input clock is CLKIN or CLKIN/2.

**PLL Status (PLL_STAT) Register**

The PLL status register (PLL_STAT) indicates the operating mode of the PLL and processor (see Figure 8-4). For more information about operating modes, see “Operating Modes” on page 8-11.

**PLL Status Register (PLL_STAT)**

Read only. Unless otherwise noted, 1 - Processor operating in this mode.

![ PLL Status Register Diagram ]

Figure 8-4. PLL Status Register

The following fields are used in the PLL_STAT register:

- **PLL_LOCKED** – This field is set to 1 when the internal PLL lock counter has incremented to the value set in the PLL lock count register (PLL_LOCKCNT). For more information, see “PLL Lock Count (PLL_LOCKCNT) Register” on page 8-10.

- **ACTIVE_PLLDISABLED** – This field is set to 1 when the processor is in active operating mode with the PLL powered down.

- **FULL_ON** – This field is set to 1 when the processor is in full on operating mode.

- **ACTIVE_Pllenabled** – This field is set to 1 when the processor is in active operating mode with the PLL powered up.
PLL Lock Count (PLL_LOCKCNT) Register

When changing clock frequencies in the PLL, the PLL requires time to stabilize and lock to the new frequency.

The PLL lock count register (PLL_LOCKCNT) defines the number of SCLK cycles that occur before the processor sets the PLL_LOCKED bit in the PLL_STAT register. When executing the PLL programming sequence, the internal PLL lock counter begins incrementing upon execution of the IDLE instruction. The lock counter increments by 1 each SCLK cycle. When the lock counter has incremented to the value defined in the PLL_LOCKCNT register, the PLL_LOCKED bit is set.

See ADSP-BF538/ADSP-BF538F Embedded Processor Data Sheet for more information about PLL stabilization time and programmed values for this register. For more information about operating modes, see “Operating Modes” on page 8-11. For further information about the PLL programming sequence, see “PLL Programming Sequence” on page 8-19.

PLL Lock Count Register (PLL_LOCKCNT)

![Figure 8-5. PLL Lock Count Register](image)

Figure 8-5. PLL Lock Count Register
Dynamic Power Management

Dynamic Power Management Controller

The dynamic power management controller (DPMC) works in conjunction with the PLL, allowing the user to control the processor’s performance characteristics and power dissipation dynamically. The DPMC provides these features that allow the user to control performance and power:

- Multiple operating modes – The processor works in four operating modes, each with different performance characteristics and power dissipation profiles. See “Operating Modes” on page 8-11.

- Peripheral clocks – Clocks to each peripheral are disabled automatically when the peripheral is disabled.

- Voltage control – The processor provides an on-chip switching regulator controller which, with some external components, can generate internal voltage levels from the external Vdd (V_{DDEXT}) supply.

Depending on the needs of the system, the voltage level can be reduced to save power. See “Voltage Regulator Control (VR_CTL) Register” on page 8-25.

Operating Modes

The processor works in four operating modes, each with unique performance and power saving benefits. Table 8-4 summarizes the operational characteristics of each mode.
Dynamic Power Management Controller States

Power management states are synonymous with the PLL control state. The state of the DPMC/PLL can be determined by reading the PLL status register (see “PLL Status (PLL_STAT) Register” on page 8-9). In all modes except sleep and deep sleep, the core can either execute instructions or be in Idle core state. If the core is in the Idle state, it can be awakened.

In all modes except active, the SCLK frequency is determined by the SSEL-specified ratio to VCO. In sleep mode, although the core clock is disabled, SCLK continues to run at the specified SSEL ratio.

The following sections describe the DPMC/PLL states in more detail, as they relate to the power management controller functions.

Full On Mode

Full on mode is the maximum performance mode. In this mode, the PLL is enabled and not bypassed. Full on mode is the normal execution state of the processor, with the processor and all enabled peripherals running at full speed. DMA access is available to L1 memories. From full on mode, the processor can transition directly to active, sleep, or deep sleep modes, as shown in Figure 8-6 on page 8-16.

Table 8-4. Operational Characteristics

<table>
<thead>
<tr>
<th>Operating Mode</th>
<th>Power Savings</th>
<th>PLL Status</th>
<th>PLL Bypassed</th>
<th>CCLK</th>
<th>SCLK</th>
<th>Allowed DMA Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full On</td>
<td>None</td>
<td>Enabled</td>
<td>No</td>
<td>Enabled</td>
<td>Enabled</td>
<td>L1</td>
</tr>
<tr>
<td>Active</td>
<td>Medium</td>
<td>Enabled 1</td>
<td>Yes</td>
<td>Enabled</td>
<td>Enabled</td>
<td>L1</td>
</tr>
<tr>
<td>Sleep</td>
<td>High</td>
<td>Enabled</td>
<td>No</td>
<td>Disabled</td>
<td>Enabled</td>
<td>–</td>
</tr>
<tr>
<td>Deep Sleep</td>
<td>Maximum</td>
<td>Disabled</td>
<td>–</td>
<td>Disabled</td>
<td>Disabled</td>
<td>–</td>
</tr>
</tbody>
</table>

1 PLL can also be disabled in this mode.
Active Mode

In active mode, the PLL is enabled but bypassed. Because the PLL is bypassed, the processor’s core clock (CCLK) and system clock (SCLK) run at the input clock (CLkin) frequency. DMA access is available to appropriately configured L1 memories.

In active mode, it is possible not only to bypass, but also to disable the PLL. If disabled, the PLL must be re-enabled before transitioning to full on or sleep modes.

From active mode, the processor can transition directly to full on, sleep, or deep sleep modes.

Sleep Mode

Sleep mode significantly reduces power dissipation by idling the core processor. The CCLK is disabled in this mode; however, SCLK continues to run at the speed configured by MSEL and SSEL bit settings. As CCLK is disabled, DMA access is available only to external memory in sleep mode. From sleep mode, a wake-up event causes the processor to transition to one of these modes:

- Active mode if the BYPASS bit in the PLL_CTL register is set
- Full on mode if the BYPASS bit is cleared

The processor resumes execution from the program counter value present immediately prior to entering sleep mode.

The STOPCK bit is not a status bit and is therefore unmodified by hardware when the wakeup occurs. Software must explicitly clear STOPCK in the next write to PLL_CTL to avoid going back into sleep mode.
Deep Sleep Mode

Deep sleep mode maximizes power savings by disabling the PLL, CCLK, and SCLK. In this mode, the processor core and all peripherals except the real-time clock (RTC) are disabled. DMA is not supported in this mode.

Deep sleep mode can be exited only by an RTC interrupt or hardware reset event. An RTC interrupt causes the processor to transition to active mode; a hardware reset begins the hardware reset sequence. For more information about hardware reset, see “Hardware Reset” on page 3-14.

Note an RTC interrupt in deep sleep mode automatically resets some fields of the PLL control register (PLL_CTL). See Table 8-5.

When in deep sleep operating mode, clocking to the SDRAM is turned off. Before entering deep sleep mode, software should ensure that important information in SDRAM is saved to a non-volatile memory.

Table 8-5. Control Register Values after RTC Wake-up Interrupt

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLL_OFF</td>
<td>0</td>
</tr>
<tr>
<td>STOPCK</td>
<td>0</td>
</tr>
<tr>
<td>PDWN</td>
<td>0</td>
</tr>
<tr>
<td>BYPASS</td>
<td>1</td>
</tr>
</tbody>
</table>

Hibernate State

For lowest possible power dissipation, this state allows the internal supply (VDDINT) to be powered down, while keeping the I/O supply (VDDEXT) running. Although not strictly an operating mode like the four modes detailed above, it is illustrative to view it as such (see Figure 8-6). Since
Dynamic Power Management

this feature is coupled to the on-chip switching regulator controller, it is discussed in detail in “Powering Down the Core (Hibernate State)” on page 8-29.

Operating Mode Transitions

Figure 8-6 graphically illustrates the operating modes and transitions. In the diagram, ellipses represent operating modes. Arrows between the ellipses show the allowed transitions into and out of each mode.

The text next to each transition arrow shows the fields in the PLL control register (PLL_CTL) that must be changed for the transition to occur. For example, the transition from full on mode to sleep mode indicates that the STOPCK bit must be set to 1 and the PDWN bit must be set to 0. For information about how to effect mode transitions, see “Programming Operating Mode Transitions” on page 8-18.

In addition to the mode transitions shown in Figure 8-6, the PLL can be modified while in active operating mode. Power to the PLL can be applied and removed, and new clock-in to VCO clock (CLKIN to VCO) multiplier ratios can be programmed. Described in detail below, these changes to the PLL do not take effect immediately. As with operating mode transitions, the PLL programming sequence must be executed for these changes to take effect (see “PLL Programming Sequence” on page 8-19).

- PLL disabled: In addition to being bypassed in the active mode, power to the PLL can be removed.

When power is removed from the PLL, additional power savings are achieved although they are relatively small. To remove power to the PLL, set the PLL_OFF bit in the PLL_CTL register, and then execute the PLL programming sequence.

- PLL enabled: When the PLL is powered down, power can be reapplied later when additional performance is required.
Figure 8-6. Operating Mode Transitions
Power to the PLL must be reapplied before transitioning to full on or sleep operating modes. To apply power to the PLL, clear the PLL_OFF bit in the PLL_CTL register, and then execute the PLL programming sequence.

- New multiplier ratio in active mode: New clock-in to VCO clock (CLKIN to VCO) multiplier ratios can be programmed while in active mode.

Although the CLKin to VCO multiplier changes are not realized in active mode, forcing the PLL to lock to the new ratio in active mode before transitioning to full on mode reduces the transition time, because the PLL is already locked to the new ratio. Note the PLL must be powered up to lock to the new ratio. To program a new CLKin to VCO multiplier, write the new MSEL[5:0] and/or DF values to the PLL_CTL register; then execute the PLL programming sequence.

- New multiplier ratio in full on mode: The multiplier ratio can also be changed while in full on mode.

In this case, the PLL state automatically transitions to active mode while the PLL is locking. After locking, the PLL returns to full on state. To program a new CLKin to VCO multiplier, write the new MSEL[5:0] and/or DF values to the PLL_CTL register; then execute the PLL programming sequence (see “PLL Programming Sequence” on page 8-19).
Table 8-6 summarizes the allowed operating mode transitions. Attempting to cause mode transitions other than those shown in Table 8-6 causes unpredictable behavior.

Table 8-6. Allowed Operating Mode Transitions

<table>
<thead>
<tr>
<th>New Mode</th>
<th>Current Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Full On</td>
</tr>
<tr>
<td>Full On</td>
<td>–</td>
</tr>
<tr>
<td>Active</td>
<td>Allowed</td>
</tr>
<tr>
<td>Sleep</td>
<td>Allowed</td>
</tr>
<tr>
<td>Deep Sleep</td>
<td>Allowed</td>
</tr>
</tbody>
</table>

Programming Operating Mode Transitions

The operating mode is defined by the state of the PLL_OFF, BYPASS, STOPCK, and PDWN bits of the PLL control register (PLL_CTL). Merely modifying the bits of the PLL_CTL register does not change the operating mode or the behavior of the PLL. Changes to the PLL_CTL register are realized only after executing a specific code sequence, which is shown in Listing 8-1. This code sequence first brings the processor to a known, idled state. Once in this idled state, the PLL recognizes and implements the changes made to the PLL_CTL register. After the changes take effect, the processor operates with the new settings, including the new operating mode, if one is programmed.
PLL Programming Sequence

If new values are assigned to MSEL or DF in the PLL control register (PLL_CTL), the instruction sequence shown in Listing 8-1 puts those changes into effect. The PLL programming sequence is also executed when transitioning between operating states.

Changes to the divider-ratio bits, CSEL and SSEL, can be made dynamically; they do not require execution of the PLL programming sequence.

Listing 8-1. PLL Programming Sequence

CLI R0 ;    /* disable interrupts */
IDLE ;      /* drain pipeline and send core into IDLE state */
STI R0 ;    /* re-enable interrupts after wakeup */

The first two instructions in the sequence take the core to an idled state with interrupts disabled; the interrupt mask (IMASK) is saved to the R0 register, and the instruction pipeline is halted. The PLL state machine then loads the PLL_CTL register changes into the PLL.

If the PLL_CTL register changes include a new CLKin to VCO multiplier or the changes reapply power to the PLL, the PLL needs to re-lock. To re-lock, the PLL lock counter is first cleared, and then it begins incrementing, once per SCLK cycle. After the PLL lock counter reaches the value programmed into the PLL Lock count register (PLLLOCKCNT), the PLL sets the PLL_LOCKED bit in the PLL status register (PLL_STAT), and the PLL asserts the PLL wake-up interrupt.

Depending on how the PLL_CTL register is programmed, the processor proceeds in one of the following four ways:

- If the PLL_CTL register is programmed to enter either active or full on operating mode, the PLL generates a wake-up signal, and then the processor continues with the STI instruction in the sequence, as
Dynamic Power Management Controller

described in “PLL Programming Sequence Continues” on page 8-21.

When the state change enters full on mode from active mode or active from full on, the PLL itself generates a wake-up signal that can be used to exit the idled core state. The wake-up signal is generated by the PLL itself or another peripheral, watchdog or other timer, RTC, or other source. For more information about events that cause the processor to wake up from being idled, see “System Interrupt Wake-Up Enable (SIC_IWRx) Registers” on page 4-27.

- If the PLL_CTL register is programmed to enter the sleep operating mode, the processor immediately transitions to the sleep mode and waits for a wake-up signal before continuing.

When the wake-up signal has been asserted, the instruction sequence continues with the STI instruction, as described in the section, “PLL Programming Sequence Continues” on page 8-21, causing the processor to transition to:

- Active mode if BYPASS in the PLL_CTL register is set
- Full on mode if the BYPASS bit is cleared

- If the PLL_CTL register is programmed to enter deep sleep operating mode, the processor immediately transitions to deep sleep mode and waits for an RTC interrupt or hardware reset signal:

- An RTC interrupt causes the processor to enter active operating mode and continue with the STI instruction in the sequence, as described below.
—A hardware reset causes the processor to execute the reset sequence, as described in “Reset” on page 4-44.

- If no operating mode transition is programmed, the PLL generates a wake-up signal, and the processor continues with the STI instruction in the sequence, as described in the following section.

**PLL Programming Sequence Continues**

The instruction sequence shown in Listing 8-1 then continues with the STI instruction. Interrupts are re-enabled, IMASK is restored, and normal program flow resumes.

To prevent spurious activity, DMA should be suspended while executing this instruction sequence.

**Examples**

The following code examples illustrate how to effect various operating mode transitions. Some setup code has been removed for clarity, and the following assumptions are made:

- P0 points to the PLL control register (PLL_CTL). P1 points to the PLL Divide register.
- The PLL wake-up interrupt is enabled as a wake-up signal.
- MSEL[5:0] and DF in PLL_CTL are set to (b#011111) and (b#0) respectively, signifying a CLKin to VCO multiplier of 31x.

**Active Mode to Full On Mode**

Listing 8-2 provides code for transitioning from active operating mode to full on mode.
Listing 8-2. Transitioning From Active Mode to Full On Mode

\begin{verbatim}
CLI R2;          /* disable interrupts, copy IMASK to R2 */  
R1.L = 0x3E00;   /* clear BYPASS bit */  
W[P0] = R1;      /* and write to PLL_CTL */  
IDLE;            /* drain pipeline, enter idled state, wait for 
PLL wakeup */  
STI R2;          /* after PLL wakeup occurs, restore interrupts 
and IMASK */  
...              /* processor is now in Full On mode */
\end{verbatim}

**Full On Mode to Active Mode**

Listing 8-3 provides code for transitioning from full on operating mode to active mode.

Listing 8-3. Transitioning From Full On Mode to Active Mode

\begin{verbatim}
CLI R2;          /* disable interrupts, copy IMASK to R2 */  
R1.L = 0x3F00;   /* set BYPASS bit */  
W[P0] = R1;      /* and write to PLL_CTL */  
IDLE;            /* drain pipeline, enter idled state, wait for 
PLL wakeup */  
STI R2;          /* after PLL wakeup occurs, restore interrupts 
and IMASK */  
...              /* processor is now in Active mode */
\end{verbatim}

**In the Full On Mode, Change CLKIN to VCO Multiplier From 31x to 2x**

Listing 8-4 provides code for changing CLKIN to VCO multiplier from 31x to 2x in full on operating mode.
### Dynamic Power Management

Listing 8-4. Changing CLKIN to VCO Multiplier

```assembly
CLI R2;          /* disable interrupts, copy IMASK to R2 */
R1.L = 0x0400;   /* change VCO multiplier to 2x */
W[P0] = R1;      /* by writing to PLL_CTL */

IDLE;            /* drain pipeline, enter idled state, wait for
                 PLL wakeup */
STI R2;          /* after PLL wakeup occurs, restore interrupts
                 and IMASK */
...
/* processor is now in Full On mode, with the
  CLKIN to VCO multiplier set to 2x */
```

### Dynamic Supply Voltage Control

In addition to clock frequency control, the processor provides the capability to run the core processor at different voltage levels. As power dissipation is proportional to the voltage squared, significant power reductions can be accomplished when lower voltages are used.

The processor uses three power domains. These power domains are shown in Table 8-7. Each power domain has a separate $V_{DD}$ supply. Note the internal logic of the processor and much of the processor I/O can be run over a range of voltages. See *ADSP-BF538/ADSP-BF538F Embedded Processor Data Sheet* for details on the allowed voltage ranges for each power domain and power dissipation data.
Table 8-7. Power Domains

<table>
<thead>
<tr>
<th>Power Domain</th>
<th>V_DD Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>All internal logic except RTC</td>
<td>Variable</td>
</tr>
<tr>
<td>Real-Time Clock I/O and internal logic</td>
<td>Variable</td>
</tr>
<tr>
<td>All other I/O</td>
<td>Variable</td>
</tr>
</tbody>
</table>

**Power Supply Management**

The processor provides an on-chip switching regulator controller which, with some external hardware, can generate internal voltage levels from the external V_DDEXT supply with an external power transistor as shown in Figure 8-7. This voltage level can be reduced to save power, depending upon the needs of the system.

When increasing the V_DDINT voltage, the external FET will switch on for a longer period. The V_DDEXT supply should have appropriate capacitive bypassing to enable it to provide sufficient current without drooping the supply voltage.
Voltage Regulator Control (VR_CTL) Register

The on-chip core voltage regulator controller manages the internal logic voltage levels for the $V_{DDINT}$ supply. The voltage regulator control register (VR_CTL) controls the regulator (see Figure 8-8). Writing to VR_CTL initiates a PLL re-lock sequence.

Figure 8-8. Voltage Regulator Control Register

The following fields of the VR_CTL register are used to control internal logic voltage levels:

- **SCKELOW** — The drive SCKE low during reset (SCKELOW) control bit protects against the default reset state behavior of setting the EBIU pins to their inactive state. This bit should be set if the SDRAM has been properly configured and is being placed into self-refresh mode while the processor is in hibernate state. Failure to set this bit results in the SCKE pin going high during reset, which takes the SDRAM out of self-refresh mode, resulting in data decay in the SDRAM due to loss of refresh rate.
The SCKE pin will be three-stated during hibernate. In addition to setting the SCKELOW bit in the VR_CTL register prior to entering the hibernate state, an external pull-down resistor on the SCKE pin is required to also keep the pin low when the Blackfin processor is not driving it.

- CANWE — The wake-up enable (CANWE) control bit allows the voltage regulator to be awakened from hibernate ($FREQ = b#00$) when a falling edge on the CANRX input pin is detected.

- GPWE — The general-purpose wake-up enable (GPWE) control bit allows the voltage regulator to be awakened from Hibernate ($FREQ = b#00$) upon detection of a falling edge on the GPW pin.

The GPW pin is a 5V-tolerant input-only pin.

- WAKE — The wake-up enable (WAKE) control bit allows the voltage regulator to be awakened from hibernate ($FREQ = b#00$) upon an interrupt from the RTC.

- $FREQ[1:0]$ — The frequency ($FREQ$) field controls the switching oscillator frequency for the voltage regulator. A higher frequency setting allows for smaller switching capacitor and inductor values, while potentially generating more EMI (electromagnetic interference).

To bypass on board regulation, program a value of b#00 in the $FREQ$ field and leave the $VROUT$ pins floating.

- $GAIN[1:0]$ — The gain ($GAIN$) field controls the internal loop gain of the switching regulator loop; this bit controls how quickly the voltage output settles on its final value. In general, higher gain allows for quicker settling times but causes more overshoot in the process.
Dynamic Power Management

- \text{VLEV}[3:0] – The voltage level (VLEV) field identifies the nominal internal voltage level. Refer to \textit{ADSP-BF538/ADSP-BF538F Embedded Processor Data Sheet} for the applicable VLEV voltage range and associated voltage tolerances.

Table 8-8 lists the voltage level values for VLEV[3:0].

Table 8-8. VLEV Encodings

<table>
<thead>
<tr>
<th>VLEV</th>
<th>Voltage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000–0101</td>
<td>Reserved</td>
</tr>
<tr>
<td>0110</td>
<td>.85 volts</td>
</tr>
<tr>
<td>0111</td>
<td>.90 volts</td>
</tr>
<tr>
<td>1000</td>
<td>.95 volts</td>
</tr>
<tr>
<td>1001</td>
<td>1.00 volts</td>
</tr>
<tr>
<td>1010</td>
<td>1.05 volts</td>
</tr>
<tr>
<td>1011</td>
<td>1.10 volts</td>
</tr>
<tr>
<td>1100</td>
<td>1.15 volts</td>
</tr>
<tr>
<td>1101</td>
<td>1.20 volts</td>
</tr>
<tr>
<td>1110</td>
<td>1.25 volts</td>
</tr>
<tr>
<td>1111</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Table 8-9 lists the switching frequency values configured by FREQ[1:0].

Table 8-9. FREQ Encodings

<table>
<thead>
<tr>
<th>FREQ</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Power down/bypass on board regulation</td>
</tr>
<tr>
<td>01</td>
<td>333 kHz</td>
</tr>
<tr>
<td>10</td>
<td>667 kHz</td>
</tr>
<tr>
<td>11</td>
<td>1 MHz</td>
</tr>
</tbody>
</table>
Table 8-10 lists the gain levels configured by GAIN[1:0].

Table 8-10. GAIN Encodings

<table>
<thead>
<tr>
<th>GAIN</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>5</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>11</td>
<td>50</td>
</tr>
</tbody>
</table>

Changing Voltage

Minor changes in operating voltage can be accommodated without requiring special consideration or action by the application program. See ADSP-BF538/ADSP-BF538F Embedded Processor Data Sheet for more information about voltage tolerances and allowed rates of change.

Reducing the processor’s operating voltage to greatly conserve power or raising the operating voltage to greatly increase performance will probably require significant changes to the operating voltage level. To ensure predictable behavior when varying the operating voltage, the processor should be brought to a known and stable state before the operating voltage is modified.

The recommended procedure is to follow the PLL programming sequence when varying the voltage. After changing the voltage level in the VR_CTL register, the PLL will automatically enter the active mode when the processor enters the idle state. At that point the voltage level will change and the PLL will re-lock with the new voltage. After the PLL_LOCKCNT has expired, the part will return to the full on state. When changing voltages, a larger PLL_LOCKCNT value may be necessary than when changing just the PLL frequency. See ADSP-BF538/ADSP-BF538F Embedded Processor Data Sheet for details.
Dynamic Power Management

After the voltage has been changed to the new level, the processor can safely return to any operational mode so long as the operating parameters, such as core clock frequency (CCLK), are within the limits specified in ADSP-BF538/ADSP-BF538F Embedded Processor Data Sheet for the new operating voltage level.

Powering Down the Core (Hibernate State)

The internal supply regulator for the processor can be shut off by writing 00 to the FREQ bits of the VR_CTL register. This disables both CCLK and SCLK. Furthermore, it sets the internal power supply voltage (V_DDINT) to 0 V, eliminating any leakage currents from the processor. The internal supply regulator can be woken up by several user-selectable events, all of which are controlled in the VR_CTL register:

- Assertion of the RESET pin will always exit hibernate state and requires no modification to the VR_CTL register.
- RTC event. Set the wake-up enable (WAKE) control bit to enable wake up upon a RTC interrupt. This can be any of the RTC interrupts (alarm, daily alarm, day, hour, minute, second, or stopwatch).
- Activity on the CANRX pin. Set the CAN RX wake-up enable (CANWE) control bit to enable wake up upon detection of CAN bus activity on the CANRX pin. See “CAN Wake-Up From Hibernate State” on page 19-4 for more details.
- Activity on the GPW pin. Set the general-purpose wake-up enable (GPWE) control bit to enable wake up upon detection of a high to low transition on the GPW pin. This allows an external host to take the Blackfin processor out of the hibernate state.
If the on-chip supply controller is bypassed, so that $V_{DDINT}$ is sourced externally, the only way to power down the core is to remove the external $V_{DDINT}$ voltage source.

When the core is powered down, $V_{DDINT}$ is set to 0V, and thus the internal state of the processor is not maintained, with the exception of the $VR_{CTL}$ register. Therefore, any critical information stored internally (memory contents, register contents, and so on) must be written to a non-volatile storage device prior to removing power. Be sure to set the $SCKELOW$ (drive $SCKE$ low during reset) control bit in the $VR_{CTL}$ register to protect against the default reset state behavior of setting the EBIU pins to their inactive state. Failure to set this bit results in the $SCKE$ pin going high during reset, which takes the SDRAM out of self-refresh mode, resulting in data decay in the SDRAM due to loss of refresh rate.

Powering down $V_{DDINT}$ does not affect $V_{DDEXT}$. While $V_{DDEXT}$ is still applied to the processor, external pins are maintained at a three-state level, unless otherwise specified.

The $SCKE$ pin will be three-stated during hibernate. In addition to setting the $SCKELOW$ bit in the $VR_{CTL}$ register prior to entering the hibernate state, an external pull-down resistor on the $SCKE$ pin is required to also keep the pin low when the Blackfin processor is not driving it.
Dynamic Power Management

To power down the internal supply:

1. Write 0 to the \texttt{SIC\_IWRx} registers to prevent peripheral resources from interrupting the hibernate process.

2. Write to \texttt{VR\_CTL}, setting the \texttt{FREQ} bits to b#00, and the appropriate wake-up bit to 1 (\texttt{CANWE, GPWE, WAKE}). Optionally, set the \texttt{SCKELOW} bit to 1 if the SDRAM is being left in self-refresh mode to maintain the SDRAM data while the processor is put into the hibernate state.

   The \texttt{SCKE} pin will be three-stated during hibernate. In addition to setting the \texttt{SCKELOW} bit in the \texttt{VR\_CTL} register prior to entering the hibernate state, an external pull-down resistor on the \texttt{SCKE} pin is required to also keep the pin low when the Blackfin processor is not driving it.

3. Execute this code sequence:

   \begin{verbatim}
   CLI R0 ;
   IDLE ;
   \end{verbatim}

4. When the idle state is reached, \texttt{V\_DDINT} transitions to 0 V.

5. When the processor is woken up, whether by the RTC, by the CAN bus, by the \texttt{GPW} pin, or by the \texttt{RESET} pin, the PLL re-locks and the boot sequence defined by the \texttt{BMODE[1:0]} pin settings takes effect.
Dynamic Power Management Controller
The processor uses direct memory access (DMA) to transfer data within memory spaces or between a memory space and a peripheral. The processor can specify data transfer operations and return to normal processing while the fully integrated DMA controllers carry out the data transfers independent of processor activity.

The processor contains two DMA engines: DMA controller 0 and DMA controller 1. These DMA controllers are two instances of the DMA engine documented in this chapter.

The DMA controllers can perform several types of data transfers:

- Between memory and memory DMA (MDMA) (For more information, see “Memory DMA” on page 9-50.)
- Between memory and a serial peripheral interface (SPI) (For more information, see “SPI Compatible Port Controllers” on page 10-1.)
- Between memory and a serial port (SPORT) (For more information, see “Serial Port Controllers” on page 13-1.)
- Between memory and a UART port (For more information, see “UART Port Controllers” on page 12-1.)
- Between memory and the parallel peripheral interface (PPI) (For more information, see “Parallel Peripheral Interface” on page 11-1.)
The system includes 14 DMA-capable peripherals, including memory DMA controllers (MDMA0–1). The 28 DMA channels found in Table 9-1 support these devices.

Table 9-1. DMA Channels

<table>
<thead>
<tr>
<th>PPI receive/transmit DMA controller</th>
<th>UART0 transmit DMA controller</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPORT0 receive DMA controller</td>
<td>UART1 receive DMA controller</td>
</tr>
<tr>
<td>SPORT0 transmit DMA controller</td>
<td>UART1 transmit DMA controller</td>
</tr>
<tr>
<td>SPORT1 receive DMA controller</td>
<td>UART2 receive DMA controller</td>
</tr>
<tr>
<td>SPORT1 transmit DMA controller</td>
<td>UART2 transmit DMA controller</td>
</tr>
<tr>
<td>SPORT2 receive DMA controller</td>
<td>MDMA0 stream0 transmit (destination)</td>
</tr>
<tr>
<td>SPORT2 transmit DMA controller</td>
<td>MDMA0 stream0 receive (source)</td>
</tr>
<tr>
<td>SPORT3 receive DMA controller</td>
<td>MDMA0 stream1 transmit (destination)</td>
</tr>
<tr>
<td>SPORT3 transmit DMA controller</td>
<td>MDMA0 stream1 receive (source)</td>
</tr>
<tr>
<td>SPI0 receive/transmit DMA controller</td>
<td>MDMA1 stream0 transmit (destination)</td>
</tr>
<tr>
<td>SPI1 receive/transmit DMA controller</td>
<td>MDMA1 stream0 receive (source)</td>
</tr>
<tr>
<td>SPI2 receive/transmit DMA controller</td>
<td>MDMA1 stream1 transmit (destination)</td>
</tr>
<tr>
<td>UART0 receive DMA controller</td>
<td>MDMA1 stream1 receive (source)</td>
</tr>
</tbody>
</table>

This chapter describes the features common to all the DMA channels, as well as how DMA operations are set up. For specific peripheral features, see the appropriate peripheral chapter for additional information. Performance and bus arbitration for DMA operations can be found in “DAB, DCB, and DEB Performance” on page 7-9.

DMA transfers on the processor can be descriptor based or register based. Descriptor based DMA transfers require a set of parameters stored within memory to initiate a DMA sequence. This sort of transfer allows the chaining together of multiple DMA sequences. In descriptor based DMA operations, a DMA channel can be programmed to automatically set up and start another DMA transfer after the current sequence completes.
Register based DMA allows the processor to directly program DMA control registers to initiate a DMA transfer. On completion, the control registers may be automatically updated with their original setup values for continuous transfer, if needed.

**DMA and Memory DMA MMRs**

For convenience, discussions in this chapter use generic (non-peripheral-specific) DMA and memory DMA register names.

Generic DMA register names are listed in Table 9-2.

Generic memory DMA register names are listed in Table 9-4.

DMA registers fall into three categories:

- **Current registers**, such as `DMAx_CURR_ADDR` and `DMAx_CURR_X_COUNT`
- **Parameter registers**, such as `DMAx_CONFIG` and `DMAx_X_COUNT`
- **Control/register registers**, `DMAx_IRQ_STATUS` and `DMAx_PERIPHERAL_MAP`

The letter x in DMAx represents a specific DMA capable peripheral. For example, for DMA with default channel mapping, `DMA6_CONFIG` represents the `DMA_CONFIG` register for the UART0 receive peripheral. For default DMA channel mappings, see Table 9-5.

Only parameter registers can be loaded directly from descriptor elements; descriptor elements are listed in Table 9-3.
Table 9-2 lists the generic names of the DMA registers. For each register, the table also shows the MMR offset, a brief description of the register, the register category, and reset value.

Table 9-2. Generic Name of DMA Memory-Mapped Registers

<table>
<thead>
<tr>
<th>MMR Offset</th>
<th>Generic MMR Name</th>
<th>MMR Description</th>
<th>Register Category</th>
<th>Reset Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>NEXT_DESC_PTR</td>
<td>Link pointer to next descriptor</td>
<td>Parameter</td>
<td>0x00000000</td>
</tr>
<tr>
<td>0x04</td>
<td>START_ADDR</td>
<td>Start address of current buffer</td>
<td>Parameter</td>
<td>0x00000000</td>
</tr>
<tr>
<td>0x08</td>
<td>DMA_CONFIG</td>
<td>DMA configuration register, including enable bit</td>
<td>Parameter</td>
<td>0x00000000</td>
</tr>
<tr>
<td>0x0C</td>
<td>Reserved</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td>X_COUNT</td>
<td>Inner loop count</td>
<td>Parameter</td>
<td>0x00000001</td>
</tr>
<tr>
<td>0x14</td>
<td>X_MODIFY</td>
<td>Inner loop address increment, in bytes</td>
<td>Parameter</td>
<td>0x00000002</td>
</tr>
<tr>
<td>0x18</td>
<td>Y_COUNT</td>
<td>Outer loop count (2D only)</td>
<td>Parameter</td>
<td>0x00000001</td>
</tr>
<tr>
<td>0x1C</td>
<td>Y_MODIFY</td>
<td>Outer loop address increment, in bytes</td>
<td>Parameter</td>
<td>0x00000002</td>
</tr>
<tr>
<td>0x20</td>
<td>CURR_DESC_PTR</td>
<td>Current descriptor pointer</td>
<td>Current</td>
<td></td>
</tr>
<tr>
<td>0x24</td>
<td>CURR_ADDR</td>
<td>Current DMA address</td>
<td>Current</td>
<td></td>
</tr>
<tr>
<td>0x28</td>
<td>IRQ_STATUS</td>
<td>Interrupt status register</td>
<td>control/register</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Contains completion and DMA error interrupt status and channel state (run/fetch/paused)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x2C</td>
<td>PERIPHERAL_MAP</td>
<td>Peripheral to DMA channel mapping</td>
<td>control/register</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Contains a 4-bit value specifying the peripheral to associate with this DMA channel (read-only for MDMA channels)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x30</td>
<td>CURR_X_COUNT</td>
<td>Current count (1D) or intra-row X count (2D), counts down from X_COUNT</td>
<td>Current</td>
<td></td>
</tr>
<tr>
<td>0x34</td>
<td>Reserved</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Direct Memory Access

All DMA registers can be accessed as 16-bit entities. However, the following registers may also be accessed as 32-bit registers: `NEXT_DESC_PTR`, `START_ADDR`, `CURR_DESC_PTR`, `CURR_ADDR`.

When these four registers are accessed as 16-bit entities, only the lower 16 bits can be accessed.

Naming Conventions for DMA MMRs

Because confusion might arise between descriptor element names and generic DMA register names, this chapter uses the naming conventions in Table 9-3, where:

- Note the generic names in the left column are not actually mapped to resources in the processor.
- The middle column lists the specific MMR name. Only specific MMR names are mapped to processor resources.
- In DMAx, the letter x represents the number of the DMA channel. For instance, `DMA3_IRQ_STATUS` is the `IRQ_STATUS` MMR for DMA channel #3.
- The channel number can be assigned by default or can be programmed. For the DMA channel numbers and the default peripheral mapping, see Table 9-5.
- The last column lists the macro assigned to each descriptor element in memory.

<table>
<thead>
<tr>
<th>MMR Offset</th>
<th>Generic MMR Name</th>
<th>MMR Description</th>
<th>Register Category</th>
<th>Reset Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x38</td>
<td>CURR_Y_COUNT</td>
<td>Current row count (2D only), counts down from Y_COUNT</td>
<td>Current</td>
<td></td>
</tr>
<tr>
<td>0x3C</td>
<td>Reserved</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 9-2. Generic Name of DMA Memory-Mapped Registers (Cont’d)
The macro name in the last column serves only to clarify the discussion of how the DMA engine operates.

The left column lists the generic name of the MMR, which is used when discussing the general operation of the DMA engine.

Table 9-3. Naming Conventions: DMA MMRs and Descriptor Elements

<table>
<thead>
<tr>
<th>Generic MMR Name</th>
<th>Specific MMR Name (x = DMA Channel Number)</th>
<th>Name of Corresponding Descriptor Element in Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA_CONFIG</td>
<td>DMAx_CONFIG</td>
<td>DMACFG</td>
</tr>
<tr>
<td>NEXT_DESC_PTR</td>
<td>DMAx_NEXT_DESC_PTR</td>
<td>NDPH (upper 16 bits), NDPL (lower 16 bits)</td>
</tr>
<tr>
<td>START_ADDR</td>
<td>DMAx_START_ADDR</td>
<td>SAH (upper 16 bits), SAL (lower 16 bits)</td>
</tr>
<tr>
<td>X_COUNT</td>
<td>DMAx_X_COUNT</td>
<td>XCNT</td>
</tr>
<tr>
<td>Y_COUNT</td>
<td>DMAx_Y_COUNT</td>
<td>YCNT</td>
</tr>
<tr>
<td>X_MODIFY</td>
<td>DMAx_X_MODIFY</td>
<td>XMOD</td>
</tr>
<tr>
<td>Y_MODIFY</td>
<td>DMAx_Y_MODIFY</td>
<td>YMOD</td>
</tr>
<tr>
<td>CURR_DESC_PTR</td>
<td>DMAx_CURR_DESC_PTR</td>
<td>N/A</td>
</tr>
<tr>
<td>CURR_ADDR</td>
<td>DMAx_CURR_ADDR</td>
<td>N/A</td>
</tr>
<tr>
<td>CURR_X_COUNT</td>
<td>DMAx_CURR_X_COUNT</td>
<td>N/A</td>
</tr>
<tr>
<td>CURR_Y_COUNT</td>
<td>DMAx_CURR_Y_COUNT</td>
<td>N/A</td>
</tr>
<tr>
<td>IRQ_STATUS</td>
<td>DMAx_IRQ_STATUS</td>
<td>N/A</td>
</tr>
<tr>
<td>PERIPHERAL_MAP</td>
<td>DMAx_PERIPHERAL_MAP</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Naming Conventions for Memory DMA Registers

The names of memory DMA registers differ somewhat from the names of other DMA registers. Memory DMA streams cannot be reassigned to different channels, whereas the peripherals associated with DMA can be mapped to any DMA channel between 0 and 21.

The processor has two DMA controllers. Each DMA controller contains two memory DMA streams. The letter 'x' denotes which of the DMA controllers the channel is in, and letters 'yy' have four possible values:

- S0, memory DMA source stream 0
- D0, memory DMA destination stream 0
- S1, memory DMA source stream 1
- D1, memory DMA destination stream 1

Table 9-4 shows the naming conventions for memory DMA registers.

Table 9-4. Naming Conventions for Memory DMA Registers

<table>
<thead>
<tr>
<th>Generic MMR Name</th>
<th>Memory DMA MMR Name (yy = S0, S1, D0, or D1 x = 0 or 1 denoting DMA Controller 0 or 1)</th>
<th>Name of Corresponding Descriptor Element in Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA_CONFIG</td>
<td>MDMAx_yy_CONFIG</td>
<td>DMACFG</td>
</tr>
<tr>
<td>NEXT_DESC_PTR</td>
<td>MDMAx_yy_NEXT_DESC_PTR</td>
<td>NDPH (upper 16 bits), NDPL (lower 16 bits)</td>
</tr>
<tr>
<td>START_ADDR</td>
<td>MDMAx_yy_START_ADDR</td>
<td>SAH (upper 16 bits), SAL (lower 16 bits)</td>
</tr>
<tr>
<td>X_COUNT</td>
<td>MDMAx_yy_X_COUNT</td>
<td>XCNT</td>
</tr>
<tr>
<td>Y_COUNT</td>
<td>MDMAx_yy_Y_COUNT</td>
<td>YCNT</td>
</tr>
<tr>
<td>X_MODIFY</td>
<td>MDMAx_yy_X_MODIFY</td>
<td>XMOD</td>
</tr>
</tbody>
</table>
Naming Conventions for Memory DMA Registers

Table 9-4. Naming Conventions for Memory DMA Registers (Cont’d)

<table>
<thead>
<tr>
<th>Generic MMR Name</th>
<th>Memory DMA MMR Name (yy = S0, S1, D0, or D1 x = 0 or 1 denoting DMA Controller 0 or 1)</th>
<th>Name of Corresponding Descriptor Element in Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y_MODIFY</td>
<td>MDMAx_yy_Y_MODIFY</td>
<td>YMOD</td>
</tr>
<tr>
<td>CURR_DESC_PTR</td>
<td>MDMAx_yy_CURR_DESC_PTR</td>
<td>N/A</td>
</tr>
<tr>
<td>CURR_ADDR</td>
<td>MDMAx_yy_CURR_ADDR</td>
<td>N/A</td>
</tr>
<tr>
<td>CURR_X_COUNT</td>
<td>MDMAx_yy_CURR_X_COUNT</td>
<td>N/A</td>
</tr>
<tr>
<td>CURR_Y_COUNT</td>
<td>MDMAx_yy_CURR_Y_COUNT</td>
<td>N/A</td>
</tr>
<tr>
<td>IRQ_STATUS</td>
<td>MDMAx_yy_IRQ_STATUS</td>
<td>N/A</td>
</tr>
<tr>
<td>PERIPHERAL_MAP</td>
<td>MDMAx_yy_PERIPHERAL_MAP</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Next Descriptor Pointer (DMAx_NEXT_DESC_PTR, MDMAx_yy_NEXT_DESC_PTR) Registers

The NEXT_DESC_PTR register specifies where to look for the start of the next descriptor block when the DMA activity specified by the current descriptor block finishes. This register is used only in small and large descriptor list modes. At the start of a descriptor fetch in either of these modes, the 32-bit NEXT_DESC_PTR is copied into CURR_DESC_PTR. Then, during the descriptor fetch, the CURR_DESC_PTR register increments after each element of the descriptor is read in.

In small and large descriptor list modes, NEXT_DESC_PTR, and not CURR_DESC_PTR, must be programmed directly via MMR access before starting DMA operation.

In descriptor array mode, the next descriptor pointer register is disregarded, and fetching is controlled only by the CURR_DESC_PTR register.
Next Descriptor Pointer Register (DMAx_NEXT_DESC_PTR / MDMAx_yy_NEXT_DESC_PTR)
R/W prior to enabling channel, RO after enabling channel

![Diagram of Next Descriptor Pointer Register](image)

Figure 9-1. Next Descriptor Pointer Register
Start Address (DMAx_START ADDR, MDMAx_yy_START_ADDR) Registers

The START_ADDR register, shown in Figure 9-2, contains the start address of the data buffer currently targeted for DMA.

Start Address Register (DMAx_START_ADDR / MDMAx_yy_START_ADDR)
R/W prior to enabling channel, RO after enabling channel

Figure 9-2. Start Address Register
Direct Memory Access

DMA Configuration (DMAx_CONFIG, MDMAx_yy_CONFIG) Registers

The DMA_CONFIG register, shown in Figure 9-3, is used to set up DMA parameters and operating modes. Note that writing the DMA_CONFIG register while DMA is already running causes a DMA error unless writing with the DMA_EN bit set to 0.

Configuration Register (DMAx_CONFIG / MDMAx_yy_CONFIG)
R/W prior to enabling channel, RO after enabling channel

```
<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>Reset</td>
<td>0x0000</td>
</tr>
<tr>
<td>14</td>
<td>DMA_EN (DMA Channel Enable)</td>
<td>0 - Disabling DMA channel. 1 - Enabling DMA channel</td>
</tr>
<tr>
<td>13</td>
<td>WNR (DMA Direction)</td>
<td>0 - DMA is a memory read (transmit) operation 1 - DMA is a memory write (receive) operation</td>
</tr>
<tr>
<td>12</td>
<td>DMA2D (DMA Mode)</td>
<td>0 - Linear 1 - Two-dimensional (2D)</td>
</tr>
<tr>
<td>11</td>
<td>DMA2D2 (DMA Mode)</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>WDSIZE[1:0] (Transfer Word Size)</td>
<td>00 - 8-bit transfers 01 - 16-bit transfers 10 - 32-bit transfers 11 - Reserved</td>
</tr>
<tr>
<td>9</td>
<td>NDSIZE[3:0] (Flex Descriptor Size)</td>
<td>0000 - Required if in Stop or Autobuffer mode 0001 - 1001 - Descriptor size 1010 - 1111 - Reserved</td>
</tr>
<tr>
<td>8</td>
<td>FLOW[2:0] (Next Operation)</td>
<td>0x0 - Stop 0x1 - Autobuffer mode 0x4 - Descriptor array 0x6 - Descriptor list (small model) 0x7 - Descriptor list (large model)</td>
</tr>
<tr>
<td>7</td>
<td>DI_SEL (Data interrupt Timing Select)</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>RESTART (DMA Buffer Clear)</td>
<td>0 - Retain DMA FIFO data between work units 1 - Discard DMA FIFO before beginning work unit</td>
</tr>
<tr>
<td>5</td>
<td>DI_EN (Data interrupt Enable)</td>
<td>0 - Do not allow completion of work unit to generate an interrupt 1 - Allow completion of work unit to generate a data interrupt</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 9-3. Configuration Register
The fields of the `DMAx_CONFIG` register are used to set up DMA parameters and operating modes.

- **FLOW[2:0]** (next operation). This field specifies the type of DMA transfer to follow the present one. The flow options are:
  - 0 – Stop the DMA activity on this channel.
  - 0x0 – Stop. When the current work unit completes, the DMA channel stops automatically, after signalling an interrupt (if selected). The DMA_RUN status bit in `DMAx_IRQ_STATUS` changes from 1 to 0, while the DMA_EN bit in `DMAx_CONFIG` is unchanged. In this state, the channel is paused. Peripheral interrupts are still filtered out by the DMA unit. The channel may be restarted simply by another write to `DMAx_CONFIG` specifying the next work unit, in which DMA_EN is set to 1.
  - 0x1 – Autobuffer mode. In this mode, no descriptors in memory are used. Instead, DMA is performed in a continuous circular-buffer fashion based on user-programmed DMAx MMR settings. On completion of the work unit, the parameter registers are reloaded into the current registers, and DMA resumes immediately with zero overhead. Autobuffer mode is stopped by a user write of 0 to the DMA_EN bit in `DMAx_CONFIG`.
  - 0x4 – Descriptor Array mode. This mode fetches a descriptor from memory that does not include the NDPH or NDPL elements. Because the descriptor does not contain a next descriptor pointer entry, the DMA engine defaults to using CURR_DESC_PTR to step through descriptors, thus allowing a group of descriptors to follow one another in memory like an array.
  - 0x6 – Descriptor list (small model) mode. This mode fetches a descriptor from memory that includes NDPL, but not NDPH. Therefore, the high 16 bits of the next descriptor pointer field are taken from the upper 16 bits of the NEXT_DESC_PTR register, thus confining all descriptors to a specific 64K page in memory.
Direct Memory Access

- **0x7 – Descriptor list (large model) mode.** This mode fetches a descriptor from memory that includes **NDPH** and **NDPL**, thus allowing maximum flexibility in locating descriptors in memory.

- **NDSIZE[3:0] (flex descriptor size).** This field specifies the number of DMA MMRs to load from descriptor elements in memory. This field must be 0 if in stop or autobuffer mode. If **NDSIZE** and **FLOW** specify a descriptor that extends beyond **YMOD**, a DMA error results.

- **DI_EN (data interrupt enable).** This bit specifies whether to allow completion of a work unit to generate a data interrupt.

- **DI_SEL (data interrupt timing select).** This bit specifies the timing of a data interrupt: after completing the whole buffer or after completing each row of the inner loop. This bit is used only in 2D DMA operation.

- **RESTART (DMA buffer clear).** This bit specifies whether receive data held in the channel’s data FIFO should be preserved (**RESTART=0**) or discarded (**RESTART=1**) before beginning the next work unit. Receive data is automatically discarded when **DMA_EN** changes from 0 to 1, typically when a channel is first enabled. Received FIFO data should usually be retained between work units if the work units make up a continuous data stream. If, however, a new work unit starts a new data stream, the **RESTART** bit should be set to 1 to clear out any previously received data.

- **DMA2D (DMA mode).** This bit specifies whether DMA mode involves only **X_COUNT** and **X_MODIFY** (one-dimensional DMA) or also involves **Y_COUNT** and **Y_MODIFY** (two-dimensional DMA).
Naming Conventions for Memory DMA Registers

- **WDSIZE[1:0]** (Transfer Word Size). The DMA engine supports transfers of 8-, 16-, or 32-bit items. Each request/grant results in a single memory access (although two cycles are required to transfer 32-bit data through a 16-bit memory port), or through the 16-bit DAB bus. The DMA address pointer increment sizes (strides) must be a multiple of the transfer unit size: 1 for 8-bit, 2 for 16-bit, 4 for 32-bit.

For information about how to set up each DMA channel to support different transfer widths, see “Peripheral Map (DMAx_PERIPHERAL_MAP, MDMAx_yy_PERIPHERAL_MAP) Registers” on page 9-21.

- **WNR** (DMA direction). This bit specifies DMA direction: memory read (0) or memory write (1).

- **DMA_EN** (DMA channel enable). This bit specifies whether to enable a given DMA channel.

  When a peripheral DMA channel is enabled, interrupts from the peripheral denote DMA requests. When a channel is disabled, the DMA unit ignores the peripheral interrupt and passes it directly to the interrupt controller. To avoid unexpected results, take care to enable the DMA channel before enabling the peripheral, and to disable the peripheral before disabling the DMA channel.

**Inner Loop Count (DMAx_X_COUNT, MDMAx_yy_X_COUNT) Registers**

For 2D DMA, the X_COUNT registers, shown in Figure 9-4, contain the inner loop count. For 1D DMA, it specifies the number of elements to read in. For details, see “Two-Dimensional DMA” on page 9-30. A value of 0 in X_COUNT corresponds to 65,536 elements.
Direct Memory Access

Inner Loop Address Increment (DMAx_X_MODIFY, MDMAx_yy_X_MODIFY) Registers

The inner loop address increment (X_MODIFY) registers contain a signed, 2’s complement byte-address increment. In 1D DMA, this increment is the stride that is applied after transferring each element.

In 2D DMA, this increment is applied after transferring each element in the inner loop, up to but not including the last element in each inner loop. After the last element in each inner loop, Y_MODIFY is applied instead, except on the very last transfer of each work unit. X_MODIFY is always applied on the last transfer of a work unit.

The X_MODIFY field may be set to 0. In this case, DMA is performed repeatedly to or from the same address. This is useful, for example, in transferring data between a data register and an external memory-mapped peripheral.

---

ADSP-BF538/ADSP-BF538F Blackfin Processor Hardware Reference 9-15
Inner Loop Address Increment Register (DMAx_X_MODIFY / MDMAx_yy_X_MODIFY)
R/W prior to enabling channel, RO after enabling channel

![Figure 9-5. Inner Loop Address Increment Register](image)

Outer Loop Count (DMAx_Y_COUNT, MDMAx_yy_Y_COUNT) Registers

For 2D DMA, the outer loop count (Y_COUNT) registers contain the outer loop count. It is not used in 1D DMA mode. This register contains the number of rows in the outer loop of a 2D DMA sequence. For details, see “Two-Dimensional DMA” on page 9-30.

Outer Loop Count Register (DMAx_Y_COUNT / MDMAx_yy_Y_COUNT)
R/W prior to enabling channel, RO after enabling channel

![Figure 9-6. Outer Loop Count Register](image)
Direct Memory Access

Outer Loop Address Increment (DMAx\_Y\_MODIFY, MDMAx\_yy\_Y\_MODIFY) Registers

The outer loop address increment (Y\_MODIFY) registers contain a signed, 2’s complement value. This byte-address increment is applied after each decrement of CURR\_Y\_COUNT except for the last item in the 2D array on which the CURR\_Y\_COUNT also expires. The value is the offset between the last word of one “row” and the first word of the next “row.” For details, see “Two-Dimensional DMA” on page 9-30.

Current Descriptor Pointer (DMAx\_CURR\_DESC\_PTR, MDMAx\_yy\_CURR\_DESC\_PTR) Registers

The current descriptor pointer (CURR\_DESC\_PTR) registers contain the memory address for the next descriptor element to be loaded. For FLOW mode settings that involve descriptors (FLOW=4, 6, or 7), this register is used to read descriptor elements into appropriate MMRs before a DMA work block begins. For descriptor list modes (FLOW=6 or 7), this register is initialized from NEXT\_DESC\_PTR before loading each descriptor. Then, the address in CURR\_DESC\_PTR increments as each descriptor element is read in. When the entire descriptor has been read, CURR\_DESC\_PTR contains the value of Descriptor Start Address + Descriptor Size (# of elements).
For descriptor array mode (FLOW=4), this register, and not the NEXT_DESC_PTR register, must be programmed by MMR access before starting DMA operation.

Current Descriptor Pointer Register (DMAx_CURR_DESC_PTR / MDMAx_yy_CURR_DESC_PTR)
R/W prior to enabling channel, RO after enabling channel

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Reset = 0000 0000

Current Descriptor Pointer[31:16]
Upper 16 bits of memory address of the next descriptor element

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Current Descriptor Pointer[15:0]
Lower 16 bits of memory address of the next descriptor element

Figure 9-8. Current Descriptor Pointer Register

Current Address (DMAx_CURR_ADDR, MDMAx_yy_CURR_ADDR) Registers

The current address (CURR_ADDR) registers, shown in Figure 9-9, contain the present DMA transfer address for a given DMA session. At the start of a DMA session, it is loaded from the START_ADDR register, and it is incremented as each transfer occurs. The current address register contains 32 bits.
Current Address Register (DMAx_CURR_ADDR / MDMAx_yy_CURR_ADDR)
R/W prior to enabling channel, RO after enabling channel

![Current Address Register Diagram]

Figure 9-9. Current Address Register

Current Inner Loop Count (DMAx_CURR_X_COUNT, MDMAx_yy_CURR_X_COUNT) Registers

The current inner loop count (CURR_X_COUNT) register is loaded by X_COUNT at the beginning of each DMA session (for 1D DMA) and also after the end of DMA for each row (for 2D DMA). Otherwise it is decremented each time an element is transferred. Expiration of the count in this register signifies that DMA is complete. In 2D DMA, CURR_X_COUNT is 0 only when the entire transfer is complete. Between rows it is equal to X_COUNT.
Naming Conventions for Memory DMA Registers

Current Inner Loop Count Register (DMAx_CURR_X_COUNT / MDMAx_yy_CURR_X_COUNT)
R/W prior to enabling channel, RO after enabling channel

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Reset = 0x0000

CURR_X_COUNT[15:0] (Current inner-Loop Count)
Loaded by X_COUNT at the beginning of each DMA session (1D DMA) or at the beginning of each row (2D DMA)

Figure 9-10. Current Inner Loop Count Register

Current Outer Loop Count (DMAx_CURR_Y_COUNT, MDMAx_yy_CURR_Y_COUNT) Registers

The current outer loop count (CURR_Y_COUNT) register is loaded by Y_COUNT at the beginning of each 2D DMA session. It is not used for 1D DMA. This register is decremented each time that the CURR_X_COUNT register expires during 2D DMA operation (1 to X_COUNT or 1 to 0 transition), signifying completion of an entire row transfer. After a 2D DMA session is complete, CURR_Y_COUNT=1 and CURR_X_COUNT=0.

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Reset = 0x0000

CURR_Y_COUNT[15:0] (Current Outer Loop Count)
Loaded by Y_COUNT at the beginning of each 2D DMA session. Not used for 1D DMA.

Figure 9-11. Current Outer Loop Count Register
Peripheral Map (DMAx_PERIPHERAL_MAP, MDMAx_yy_PERIPHERAL_MAP) Registers

Each DMA channel PERIPHERAL_MAP register contains bits that:

- Map the channel to a specific peripheral
- Identify whether the channel is a peripheral DMA channel or a memory DMA channel.

There are two sets of PERIPHERAL_MAP registers. One set is for the channels associated with DMA controller 0 and the other set for the channels associated with DMA controller 1. Peripherals are assigned to a specific controller (0 or 1). This assignment is fixed and is not selectable by the user.

Note that a 1:1 mapping should exist between DMA channels and peripherals. The user is responsible for ensuring that multiple DMA channels are not mapped to the same peripheral and that multiple peripherals are not mapped to the same DMA port. If multiple channels are mapped to the same peripheral, only one channel is connected (the lowest-priority channel). If a nonexistent peripheral (for example, 0xF in the PMAP field) is mapped to a channel, that channel is disabled—DMA requests are ignored, and no DMA grants are issued. The DMA requests are also not forwarded from the peripheral to the interrupt controller.
Follow these steps to swap the DMA channel priorities of two channels. Assume that channels 6 and 7 are involved.

1. Make sure DMA is disabled on channels 6 and 7
2. Write DMA6_PERIPHERAL_MAP with 0x7000 and DMA7_PERIPHERAL_MAP with 0x6000
3. Enable DMA on channels 6 and/or 7

Figure 9-12. DMA Controller 0 Peripheral Map Register

---

DMA Controller 0 Peripheral Map Register (DMAx_PERIPHERAL_MAP / MDMAx_yy_PERIPHERAL_MAP)
R/W prior to enabling channel, RO after enabling channel

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Reset = 0x0000

PMAP[3:0] (Peripheral Mapped to This Channel)
- 0x0 - PPI
- 0x1 - SPORT0 RX
- 0x2 - SPORT0 TX
- 0x3 - SPORT1 RX
- 0x4 - SPORT1 TX
- 0x5 - SPI0
- 0x6 - UART0 RX
- 0x7 - UART0 TX

CTYPE (DMA Channel Type)
- RO
  - 0 - Peripheral DMA
  - 1 - Memory DMA
Table 9-5 lists the binary peripheral map settings for each DMA capable peripheral.

### Table 9-5. Peripheral Mapping of DMA Controller 0

<table>
<thead>
<tr>
<th>DMA Channel</th>
<th>Default Peripheral Mapping</th>
<th>Default PERIPHERAL_MAP Setting (Binary)</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA0 (highest priority)</td>
<td>PPI</td>
<td>0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>DMA1</td>
<td>SPORT0 RX</td>
<td>0001 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>DMA2</td>
<td>SPORT0 TX</td>
<td>0010 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>DMA3</td>
<td>SPORT1 RX</td>
<td>0011 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>DMA4</td>
<td>SPORT1 TX</td>
<td>0100 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>DMA5</td>
<td>SPI0</td>
<td>0101 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>DMA6</td>
<td>UART0 RX</td>
<td>0110 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>DMA7</td>
<td>UART0 TX</td>
<td>0111 0000 0000 0000</td>
<td></td>
</tr>
</tbody>
</table>
### Table 9-5. Peripheral Mapping of DMA Controller 0 (Cont’d)

<table>
<thead>
<tr>
<th>DMA Channel</th>
<th>Default Peripheral Mapping</th>
<th>Default PERIPHERAL_MAP Setting (Binary)</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>MDMA0_D0</td>
<td>Mem DMA Stream 0 TX (destination)</td>
<td>0000 0000 0100 0000</td>
<td>Not re-assignable</td>
</tr>
<tr>
<td>MDMA0_S0</td>
<td>Mem DMA Stream 0 RX (source)</td>
<td>0000 0000 0100 0000</td>
<td>Not re-assignable</td>
</tr>
<tr>
<td>MDMA0_D1</td>
<td>Mem DMA Stream 1 TX (destination)</td>
<td>0000 0000 0100 0000</td>
<td>Not re-assignable</td>
</tr>
<tr>
<td>MDMA0_S1</td>
<td>Mem DMA Stream 1 RX (source)</td>
<td>0000 0000 0100 0000</td>
<td>Not re-assignable</td>
</tr>
</tbody>
</table>

**Table 9-6** assumes the following peripheral ID mappings into the 4-bit **PMAP** field.

### Table 9-6. Peripheral ID Mappings Into PMAP Field for DMA Controller 0

<table>
<thead>
<tr>
<th>Peripheral</th>
<th>PMAP ID#</th>
<th>Peripheral</th>
<th>PMAP ID#</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPI</td>
<td>0000</td>
<td>SPORT1 TX</td>
<td>0100</td>
</tr>
<tr>
<td>SPORT0 RX</td>
<td>0001</td>
<td>SPI</td>
<td>0101</td>
</tr>
<tr>
<td>SPORT0 TX</td>
<td>0010</td>
<td>UART0 RX</td>
<td>0110</td>
</tr>
<tr>
<td>SPORT1 RX</td>
<td>0011</td>
<td>UART0 TX</td>
<td>0111</td>
</tr>
</tbody>
</table>

### Table 9-7. Peripheral Mapping of DMA Controller 1

<table>
<thead>
<tr>
<th>DMA Channel</th>
<th>Default Peripheral Mapping</th>
<th>Default PERIPHERAL_MAP Setting (Binary)</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA8</td>
<td>SPORT2 RX</td>
<td>0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>DMA9</td>
<td>SPORT2 TX</td>
<td>0001 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>DMA10</td>
<td>SPORT3 RX</td>
<td>0010 0000 0000 0000</td>
<td></td>
</tr>
</tbody>
</table>
Table 9-7. Peripheral Mapping of DMA Controller 1 (Cont’d)

<table>
<thead>
<tr>
<th>DMA Channel</th>
<th>Default Peripheral Mapping</th>
<th>Default PERIPHERAL_MAP Setting (Binary)</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA11</td>
<td>SPORT3 TX</td>
<td>0011 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>DMA12</td>
<td>Unassigned</td>
<td>0100 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>DMA13</td>
<td>Unassigned</td>
<td>0101 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>DMA14</td>
<td>SPI1</td>
<td>0110 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>DMA15</td>
<td>SPI2</td>
<td>0111 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>DMA16</td>
<td>UART1 RX</td>
<td>1000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>DMA17</td>
<td>UART1 TX</td>
<td>1001 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>DMA18</td>
<td>UART2 RX</td>
<td>1010 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>DMA19</td>
<td>UART2 TX</td>
<td>1011 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>MDMA1_D0_D1</td>
<td>Mem DMA Stream 2 TX (destination)</td>
<td>0000 0000 0100 0000</td>
<td>Not re-assignable</td>
</tr>
<tr>
<td>MDMA1_S0_S1</td>
<td>Mem DMA Stream 2 RX (source)</td>
<td>0000 0000 0100 0000</td>
<td>Not re-assignable</td>
</tr>
<tr>
<td>MDMA1_D1_D1</td>
<td>Mem DMA Stream 3 TX (destination)</td>
<td>0000 0000 0100 0000</td>
<td>Not re-assignable</td>
</tr>
</tbody>
</table>

Table 9-8. Peripheral ID Mappings Into PMAP Field for DMA Controller 1

<table>
<thead>
<tr>
<th>Peripheral</th>
<th>PMAP ID#</th>
<th>Peripheral</th>
<th>PMAP ID#</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPORT2 RX</td>
<td>0000</td>
<td>SPI1</td>
<td>0110</td>
</tr>
<tr>
<td>SPORT2 TX</td>
<td>0001</td>
<td>SPI2</td>
<td>0111</td>
</tr>
<tr>
<td>SPORT3 RX</td>
<td>0010</td>
<td>UART1 RX</td>
<td>1000</td>
</tr>
<tr>
<td>SPORT3 TX</td>
<td>0011</td>
<td>UART1 TX</td>
<td>1001</td>
</tr>
<tr>
<td>Unassigned</td>
<td>0100</td>
<td>UART2 RX</td>
<td>1010</td>
</tr>
<tr>
<td>Unassigned</td>
<td>0101</td>
<td>UART2 TX</td>
<td>1011</td>
</tr>
</tbody>
</table>
Interrupt Status (DMAx_IRQ_STATUS, MDMAx_yy_IRQ_STATUS) Registers

The interrupt register (IRQ_STATUS) register, shown in Figure 9-14, contains bits that record whether the DMA channel:

- Is enabled and operating, enabled but stopped, or disabled
- Is fetching data or a DMA descriptor
- Has detected that a global DMA interrupt or a channel interrupt is being asserted
- Has logged occurrence of a DMA error

Note the DMA_DONE interrupt is asserted when the last memory access (read or write) has completed. For a transmit (memory read) transfer to a peripheral, there may be up to four data words in the channel’s DMA FIFO when the interrupt occurs. At this point, it is normal to immediately start the next work unit. If, however, the application needs to know when the final data item is actually transferred to the peripheral, the application can test or poll the DMA_RUN bit. As long as there is un-delivered transmit data in the FIFO, DMA_RUN is 1.

The processor supports a flexible interrupt control structure with three interrupt sources. Separate IRQ levels are allocated for data, peripheral errors, and DMA errors:

- data-driven interrupts (see Table 9-7).
- peripheral error interrupts.
- DMA error interrupts (for example, bad descriptor or bus error)

All DMA channels are OR’ed together into one system-level DMA error interrupt. The individual IRQ_STATUS words of each channel can be read to identify the channel that caused the DMA error interrupt.
Direct Memory Access

Interrupt Status Register (DMAx_IRQ_STATUS / MDMA_yy_IRQ_STATUS)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>Reset = 0x0000</td>
</tr>
<tr>
<td>14</td>
<td>DMA_RUN (DMA Channel Running) - RO</td>
</tr>
<tr>
<td>13</td>
<td>This bit is set to 1 automatically when the DMA_CONFIG register is written</td>
</tr>
<tr>
<td>12</td>
<td>0 - This DMA channel is disabled, or it is enabled but paused (FLOW mode 0)</td>
</tr>
<tr>
<td>11</td>
<td>1 - This DMA channel is enabled and operating, either transferring data or fetching a DMA descriptor</td>
</tr>
<tr>
<td>10</td>
<td>DFETCH (DMA Descriptor Fetch) - RO</td>
</tr>
<tr>
<td>9</td>
<td>This bit is set to 1 automatically when the DMA_CONFIG register is written with FLOW modes 4–7</td>
</tr>
<tr>
<td>8</td>
<td>0 - This DMA channel is disabled, or it is enabled but stopped (FLOW mode 0)</td>
</tr>
<tr>
<td>7</td>
<td>1 - This DMA channel is enabled and presently fetching a DMA descriptor</td>
</tr>
<tr>
<td>6</td>
<td>DMA_DONE (DMA Completion interrupt register) - write-1-to-clear</td>
</tr>
<tr>
<td>5</td>
<td>0 - No interrupt is being asserted for this channel</td>
</tr>
<tr>
<td>4</td>
<td>1 - DMA work unit has completed, and this DMA channel’s interrupt is being asserted</td>
</tr>
<tr>
<td>3</td>
<td>DMA_ERR (DMA Error interrupt register) - write-1-to-clear</td>
</tr>
<tr>
<td>2</td>
<td>0 - No DMA error has occurred</td>
</tr>
<tr>
<td>1</td>
<td>1 - A DMA error has occurred, and the global DMA Error interrupt is being asserted. After this error occurs, the contents of the DMA Current registers are unspecified. Control/ register and Parameter registers are unchanged.</td>
</tr>
<tr>
<td>0</td>
<td>DMA_CTRL (DMA Control register) - RO</td>
</tr>
</tbody>
</table>

Figure 9-14. Interrupt Status Register

Table 9-9. Data Driven Interrupts

<table>
<thead>
<tr>
<th>Interrupt Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>No interrupt</td>
<td>Interrupts can be disabled for a given work unit.</td>
</tr>
<tr>
<td>Peripheral interrupt</td>
<td>Peripheral (non-DMA) interrupt.</td>
</tr>
<tr>
<td>Row Completion</td>
<td>DMA interrupts can occur on the completion of a row (CURR_X_COUNT expiration).</td>
</tr>
<tr>
<td>Buffer Completion</td>
<td>DMA interrupts can occur on the completion of an entire buffer (when CURR_X_COUNT and CURR_Y_COUNT expire).</td>
</tr>
</tbody>
</table>
When switching a peripheral from DMA to non-DMA mode, the peripheral interrupts should be disabled during the mode switch (via the appropriate peripheral registers or SIC_IMASKx) so that no unintended interrupt is generated on the shared DMA/interrupt request line.

**Flex Descriptor Structure**

DMA flex descriptors are variable sized data structures whose contents are loaded into DMA parameter registers. The sequence of registers in the descriptor is essentially fixed (among three similar variations), but the length of the descriptor is completely programmable. The DMA channel registers are ordered so that the registers that are most commonly reloaded per work unit are at the lowest MMR addresses. The user may choose whether or not to use descriptors. If not using descriptors, the user can write the DMA MMRs directly to start DMA, and use either Autobuffer mode for continuous operation or Stop mode for single-buffer operation.

To use descriptors, the user programs the NDSIZE field of the DMAx_CONFIG register with the number of DMA registers to load from the descriptor, starting with the lowest MMR address. The user may select a descriptor size from one entry (the lower 16 bits of START_ADDR) to nine entries (all the DMA parameters).

The three variations of the descriptor value sequences depend on whether a next descriptor pointer is included and, if so, what kind.

1. None included (descriptor array mode)
2. The lower 16 bits of the next descriptor pointer (descriptor list, small model)
3. All 32 bits of the next descriptor pointer (descriptor list, large model)
All the other registers not loaded from the descriptor retain their prior values (although the CURR_ADDR, CURR_X_COUNT, CURR_Y_COUNT registers are reloaded between the descriptor fetch and the start of DMA operation.)

There are certain DMA settings that are not allowed to change from one descriptor to the next in a chain (small or large list and array modes). These include DMA direction, word size, and memory space (that is, switching between internal and external memory).

A single descriptor chain cannot control the transfer of a sequence of data buffers which reside in different memory spaces. Instead, group the data buffers into chains of buffers in the same space, but do not link the chains together. Transfer the first chain, wait for its final interrupt, and then start the next chain with an MMR write to DMA_CONFIG.

Note that while the user must locate each chain’s data buffers in the same memory space, the descriptor structures themselves may be placed in any memory space, and they may link from a descriptor in one space to a descriptor in the other space without restriction.

Table 9-10 shows the descriptor offsets for descriptor elements in the three modes described above. Note the names in the table list the descriptor elements in memory, not the actual MMRs into which they are eventually loaded.

Table 9-10. Parameter Registers and Descriptor Offsets

<table>
<thead>
<tr>
<th>Descriptor Offset</th>
<th>Descriptor Array Mode</th>
<th>Small Descriptor List Mode</th>
<th>Large Descriptor List Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>SAL</td>
<td>NDPL</td>
<td>NDPL</td>
</tr>
<tr>
<td>0x2</td>
<td>SAH</td>
<td>SAL</td>
<td>NDPH</td>
</tr>
<tr>
<td>0x4</td>
<td>DMACFG</td>
<td>SAH</td>
<td>SAL</td>
</tr>
<tr>
<td>0x6</td>
<td>XCNT</td>
<td>DMACFG</td>
<td>SAH</td>
</tr>
<tr>
<td>0x8</td>
<td>XMOD</td>
<td>XCNT</td>
<td>DMACFG</td>
</tr>
<tr>
<td>0xA</td>
<td>YCNT</td>
<td>XMOD</td>
<td>XCNT</td>
</tr>
</tbody>
</table>
Two-Dimensional DMA

Two-dimensional DMA supports arbitrary row and column sizes up to 64 K x 64 K elements, as well as arbitrary \texttt{X\_MODIFY} and \texttt{Y\_MODIFY} values up to ±32 K bytes. Furthermore, \texttt{Y\_MODIFY} can be negative, allowing implementation of interleaved data streams. \texttt{X\_COUNT} and \texttt{Y\_COUNT} specify the row and column sizes, where \texttt{X\_COUNT} must be 2 or greater.

The start address and modify values are in bytes, and they must be aligned to a multiple of the DMA transfer word size (\texttt{WDSIZE[1:0]} in \texttt{DMA\_CONFIG}). Misalignment causes a DMA error.

The \texttt{X\_MODIFY} value is the byte-address increment that is applied after each transfer that decrements \texttt{CURR\_X\_COUNT}. The \texttt{X\_MODIFY} value is not applied when the inner loop count is ended by decrementing \texttt{CURR\_X\_COUNT} from 1 to 0, except that it is applied on the final transfer when \texttt{CURR\_Y\_COUNT} is 1 and \texttt{CURR\_X\_COUNT} decrements from 1 to 0.

The \texttt{Y\_MODIFY} value is the byte-address increment that is applied after each decrement of \texttt{CURR\_Y\_COUNT}. However, the \texttt{Y\_MODIFY} value is not applied to the last item in the array on which the outer loop count (\texttt{CURR\_Y\_COUNT}) also expires by decrementing from 1 to 0.

Table 9-10. Parameter Registers and Descriptor Offsets (Cont’d)

<table>
<thead>
<tr>
<th>Descriptor Offset</th>
<th>Descriptor Array Mode</th>
<th>Small Descriptor List Mode</th>
<th>Large Descriptor List Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xC</td>
<td>YMOD</td>
<td>YCNT</td>
<td>XMOD</td>
</tr>
<tr>
<td>0xE</td>
<td></td>
<td>YMOD</td>
<td>YCNT</td>
</tr>
<tr>
<td>0x10</td>
<td></td>
<td></td>
<td>YMOD</td>
</tr>
</tbody>
</table>

**Two-Dimensional DMA**

[Table 9-10. Parameter Registers and Descriptor Offsets (Cont’d)]
After the last transfer completes, CURR_Y_COUNT=1, CURR_X_COUNT=0, and CURR_ADDR is equal to the last item’s address plus X_MODIFY. Note that if the DMA channel is programmed to refresh automatically (auto buffer mode), then these registers are loaded from X_COUNT, Y_COUNT, and START_ADDR upon the first data transfer.

Example 1: Retrieve a 16x8 Block of Bytes From a Video Frame Buffer of Size (N x M) Pixels:

\[ X_{\text{MODIFY}} = 1 \]
\[ X_{\text{COUNT}} = 16 \]
\[ Y_{\text{MODIFY}} = N-15 \text{ (offset from the end of one row to the start of another)} \]
\[ Y_{\text{COUNT}} = 8 \]

This produces the following address offsets from the start address:

\[ 0, 1, 2, \ldots, 15, \]
\[ N, N+1, \ldots, N+15, \]
\[ 2N, 2N+1, \ldots, 2N+15, \ldots \]
\[ 7N, 7N+1, \ldots, 7N+15, \]
\[ 0, \ldots \]

Example 2: Receive a Video Data Stream of Bytes, (R,G,B Pixels) x (N x M Image Size):

\[ X_{\text{MODIFY}} = (N*M) \]
\[ X_{\text{COUNT}} = 3 \]
\[ Y_{\text{MODIFY}} = 1 - 2(N*M) \text{ (negative)} \]
\[ Y_{\text{COUNT}} = (N*M) \]
DMA Operation Flow

This produces the following address offsets from the start address:

0, (N*M), 2(N*M),

1, (N*M)+1, 2(N*M)+1,

2, (N*M)+2, 2(N*M)+2,

...  

(N*M)-1, 2(N*M)-1, 3(N*M)-1.

0, ...

DMA Operation Flow

Figure 9-15 and Figure 9-16 describe the DMA Flow.
Figure 9-15. DMA Flow, From DMA Controller Point of View (1 of 2)
DMA Operation Flow

Figure 9-16. DMA Flow, From DMA Controller Point of View (2 of 2)
DMA Startup

This section discusses starting DMA “from scratch.” This is similar to starting it after it has been paused by FLOW = 0 mode.

Before initiating DMA for the first time on a given channel, be sure to initialize all parameter registers. Be especially careful to initialize the upper 16 bits of NEXT_DESC_PTR and START_ADDR, because they might not otherwise be accessed, depending on the chosen FLOW mode of operation.

To start DMA operation on a given channel, some or all of the DMA parameter registers must first be written directly. At a minimum, the NEXT_DESC_PTR register (or CURR_DESC_PTR register in FLOW = 4 mode) must be written at this stage, but the user may wish to write other DMA registers that might be static throughout the course of DMA activity (for example, X_MODIFY, Y_MODIFY). The contents of NDSIZE and FLOW in DMA_CONFIG indicate which registers, if any, are fetched from descriptor elements in memory. After the descriptor fetch, if any, is completed, DMA operation begins, initiated by writing DMA_CONFIG with DMA_EN = 1.

When DMA_CONFIG is written directly, the DMA controller recognizes this as the special startup condition that occurs when starting DMA for the first time on this channel or after the engine has been stopped (FLOW = 0).

When the descriptor fetch is complete and DMA_EN=1, the DMACFG descriptor element that was read into DMA_CONFIG assumes control. (Before this point, the direct write to DMA_CONFIG had control.)

As Figure 9-15 and Figure 9-16 show, at startup the FLOW and NDSIZE bits in DMA_CONFIG determine the course of the DMA setup process. The FLOW value determines whether to load more current registers from descriptor elements in memory, while the NDSIZE bits detail how many descriptor elements to fetch before starting DMA. DMA registers not included in the descriptor are not modified from their prior values. Moreover, the reset values of DMA MMRs are never restored except after a system reset.
DMA Operation Flow

If the `FLOW` value specifies small or large descriptor list modes, the `NEXT_DESC_PTR` is copied into `CURR_DESC_PTR`. Then, fetches of new descriptor elements from memory are performed, indexed by `CURR_DESC_PTR`, which is incremented after each fetch. If `NDPL` and/or `NDPH` is part of the descriptor, then these values are loaded into `NEXT_DESC_PTR`, but the fetch of the current descriptor continues using `CURR_DESC_PTR`. After completion of the descriptor fetch, `CURR_DESC_PTR` points to the next 16-bit word in memory past the end of the descriptor.

If neither `NDPH` nor `NDPL` are part of the descriptor (that is, in descriptor array mode, `FLOW = 4`), then the transfer from `NDPH/NDPL` into `CURR_DESC_PTR` does not occur. Instead, descriptor fetch indexing begins with the value in `CURR_DESC_PTR`.

If `DMACFG` is not part of the descriptor, the previous `DMA_CONFIG` settings (as written by MMR access at startup) control the work unit operation. If `DMACFG` is part of the descriptor, then the `DMA_CONFIG` value programmed by the MMR access controls only the loading of the first descriptor from memory. The subsequent DMA work operation is controlled by the low byte of the descriptor `DMACFG` and by the parameter registers loaded from the descriptor. The bits `DI_EN`, `DI_SEL`, `DMA2D`, `WDSIZE`, and `WNR` in the value programmed by the MMR access are disregarded.

The `DMA_RUN` and `DFETCH` status bits in the `IRQ_STATUS` register indicate the state of the DMA channel. After a write to `DMA_CONFIG`, the `DMA_RUN` and `DFETCH` bits can be automatically set to 1. No data interrupts are signaled as a result of loading the first descriptor from memory.

After the above steps, the current registers are loaded automatically from the appropriate descriptor elements, overwriting their previous contents, as follows.

- `START_ADDR` is copied to `CURR_ADDR`
- `X_COUNT` is copied to `CURR_X_COUNT`
- `Y_COUNT` is copied to `CURR_Y_COUNT`
Then DMA data transfer operation begins, as shown in Figure 9-16.

**DMA Refresh**

On completion of a work unit, the DMA controller completes the transfer of all data between memory and the DMA unit. If enabled by DI_EN, the DMA controller signals an interrupt to the core and sets the DMA_DONE bit in the channel’s IRQ_STATUS register.

When the FLOW bit is cleared (= 0) the operation is stopped by clearing the DMA_RUN bit in IRQ_STATUS after any transmit data in the channel’s DMA FIFO has been transferred to the peripheral.

During the fetch in FLOW modes 4, 6, and 7, the DMA controller sets the DFETCH bit in IRQ_STATUS to 1. At this point, the DMA operation depends on whether FLOW = 4, 6, or 7, as follows:

**If FLOW = 4 (descriptor array):** Loads a new descriptor from memory into DMA registers via the contents of CURR_DESC_PTR, while incrementing CURR_DESC_PTR. The descriptor size comes from the NDSIZE field of the DMA_CONFIG value prior to the beginning of the fetch.

**If FLOW = 6 (descriptor list small):** Copies the 32-bit NEXT_DESC_PTR into CURR_DESC_PTR. Next, fetches a descriptor from memory into DMA registers via the new contents of CURR_DESC_PTR, while incrementing CURR_DESC_PTR. The first descriptor element loaded is a new 16-bit value for the lower 16 bits of NEXT_DESC_PTR, followed by the rest of the descriptor elements. The high 16 bits of NEXT_DESC_PTR retain their former value. This supports a shorter, more efficient descriptor than the descriptor list large model, suitable whenever the application can place the channel’s descriptors in the same 64KB range of memory.
If FLOW = 7 (descriptor list large): Copies the 32-bit NEXT_DESC_PTR into CURR_DESC_PTR. Next, fetches a descriptor from memory into DMA registers via the new contents of CURR_DESC_PTR, while incrementing CURR_DESC_PTR. The first descriptor elements loaded are a new 32-bit value for the full NEXT_DESC_PTR, followed by the rest of the descriptor elements. The high 16 bits of NEXT_DESC_PTR may differ from their former value. This supports a fully flexible descriptor list which can be located anywhere in internal memory, external memory, or ROM.

Note if it is necessary to link from a descriptor chain whose descriptors are in one 64KB area to another chain whose descriptors are outside that area, only one descriptor needs to use FLOW=7—just the descriptor which contains the link leaving the 64KB range. All the other descriptors located together in the same 64KB areas may use FLOW = 6.

If FLOW = 1, 4, 6, or 7 (auto buffer, descriptor array, descriptor list small, or descriptor list large, respectively):

(Re)loads the current registers:

CURR_ADDR loaded from START_ADDR,
CURR_X_COUNT loaded from X_COUNT,
CURR_Y_COUNT loaded from Y_COUNT

The DFETCH bit in IRQ_STATUS is then cleared, after which the DMA transfer begins again, as shown in Figure 9-16.

To Stop DMA Transfers

In FLOW = 0 mode, DMA stops automatically after the work unit is complete.

If a list or array of descriptors is used to control DMA, and if every descriptor contains a DMACFG element, then the final DMACFG element should have a FLOW = 0 setting to gracefully stop the channel.
In auto buffer ($\text{FLOW} = 1$) mode, or if a list or array of descriptors without DMACFG elements is used, then the DMA transfer process must be terminated by an MMR write to the DMAx_CONFIG register with a value whose DMA_EN bit is 0. A write of 0 to the entire register always terminates DMA gracefully (without DMA abort).

Before enabling the channel again, make sure that any slow memory read operations that may have started are completed (for example, reads from slow external memory). Do not enable the channel again until any such reads are complete.

**Software Management of DMA**

Several synchronization and control methods are available for use in development of software tasks which manage DMA and MDMA (see also “Memory DMA” on page 9-50). Such software needs to be able to accept requests for new DMA transfers from other software tasks, integrate these transfers into existing transfer queues, and reliably notify other tasks when the transfers are complete.

In the processor, it is possible for each DMA peripheral and MDMA stream to be managed by a separate task or to be managed together with any other stream. Each DMA channel has independent, orthogonal control registers, resources, and interrupts, so that the selection of the control scheme for one channel does not affect the choice of control scheme on other channels. For example, one peripheral can use a linked-descriptor-list, interrupt-driven scheme while another peripheral can simultaneously use a demand-driven, buffer-at-a-time scheme synchronized by polling of the IRQ_STATUS register.
Synchronization of Software and DMA

A critical element of software DMA management is synchronization of DMA buffer completion with the software. This can best be done using interrupts, polling of IRQ_STATUS, or a combination of both. Polling for address or count can only provide synchronization within loose tolerances comparable to pipeline lengths.

Interrupt-based synchronization methods must avoid interrupt overrun, or the failure to invoke a DMA channel’s interrupt handler for every interrupt event due to excessive latency in processing of interrupts. Generally, the system design must either ensure that only one interrupt per channel is scheduled (for example, at the end of a descriptor list), or that interrupts are spaced sufficiently far apart in time that system processing budgets can guarantee every interrupt is serviced. Note since every interrupt channel has its own distinct interrupt, interaction among the interrupts of different peripherals is much simpler to manage.

Polling of the CURR_ADDR, CURR_DESC_PTR, or CURR_X/Y_COUNT registers is not recommended as a method of precisely synchronizing DMA with data processing, due to DMA FIFOS and DMA/memory pipelining. The current address, pointer, and count registers change several cycles in advance of the completion of the corresponding memory operation, as measured by the time at which the results of the operation would first be visible to the core by memory read or write instructions. For example, in a DMA receive (memory write) operation to external memory, assume a DMA write by channel A is initiated which causes the SRAM to perform a page open operation which will take many system clock cycles. The DMA engine may then move on to another DMA operation by channel B which does not in itself incur latency, but which will be stalled behind the slow operation by channel A. Software monitoring channel B could not safely conclude whether the memory location pointed to by the channel B CURR_ADDR has or has not been written, based on examination of the CURR_ADDR register contents.
Polling of the current address, pointer, and count registers can permit loose synchronization of DMA with software, however, if allowances are made for the lengths of the DMA/memory pipeline. The length of the DMA FIFO for a peripheral DMA channel is four locations (either four 8- or 16-bit data elements, or two 32-bit data elements) and for an MDMA FIFO is eight locations (or four 32-bit data elements). The DMA does not advance current address/pointer/count registers if these FIFOs are filled with incomplete work (including reads that have been started but not yet finished). Next, the length of the combined DMA and L1 pipelines to internal memory is approximately six 8- or 16-bit data elements, while the length of the DMA and EBIU pipelines is approximately three data elements, measuring from the point where a DMA register update is visible to an MMR read to the point where DMA and core accesses to memory become strictly ordered. If the DMA FIFO length and the DMA/memory pipeline length are added, an estimate can be made of the maximum number of incomplete memory operations in progress at one time. (Note this is a maximum, as the DMA/memory pipeline may include traffic from other DMA channels.) For example, assume a peripheral DMA channel is transferring a work unit of 100 data elements into internal memory and its CURR_X_COUNT register reads a value of 60 remaining elements, so that processing of the first 40 elements has at least been started. The total pipeline length is no greater than the sum of 4 (for the PDMA FIFO) plus 6 (for the DMA/memory pipeline), or 10 data elements, so it is safe to conclude that the DMA transfer of the first 40-10 = 30 data elements is complete.

For precise synchronization, software should either wait for an interrupt or consult the channel’s IRQ_STATUS register to confirm completion of DMA, rather than polling current-address/pointer/count registers. When the DMA system issues an interrupt or changes an IRQ_STATUS status bit, it guarantees that the last memory operation of the work unit has been completed and will definitely be visible to DSP code. For memory read DMA, the final memory read data will have been safely received in the DMA’s FIFO; for memory write DMA, the DMA unit will have received an acknowledge from L1 or the EBIU that the data has been written.
DMA Operation Flow

The following examples show methods of synchronizing software with several different styles of DMA.

Single-Buffer DMA Transfers

Synchronization is simple if peripheral DMA activity consists of isolated transfers of single buffers. DMA activity is initiated by software writes to the channel’s MMR control registers. The user may choose to use a single descriptor in memory, in which case the software only needs to write the DMA_CONFIG and the NEXT_DESC_PTR registers. Alternatively, the user may choose to write all the MMR registers directly from software, ending with the write to the DMA_CONFIG register.

The simplest way to signal completion of DMA is by an interrupt. This is selected by the DI_EN bit in the DMA_CONFIG register, and by the necessary setup of the system interrupt controllers. If it is desirable not to use an interrupt, the software can poll for completion by reading the IRQ_STATUS register and testing the DMA_RUN bit. If this bit is zero, the buffer transfer has completed.

Continuous Transfers Using Auto Buffering

If a peripheral’s DMA data consists of a steady, periodic stream of signal data, DMA auto buffering (FLOW=1) may be an effective option. Here, DMA is transferred from or to a memory buffer using a circular addressing scheme, using either 1- or 2-dimensional indexing with zero processor and DMA overhead for looping. Synchronization options include:

1-D, interrupt driven—software is interrupted at the conclusion of each buffer. The critical design consideration is that the software must deal with the first items in the buffer before the next DMA transfer, which might overwrite or re-read the first buffer location before it is processed by software. This scheme may be workable if the system design guarantees that the data repeat period is longer than the interrupt latency under all circumstances.
Direct Memory Access

2-D, interrupt-driven (double buffering)—the DMA buffer is partitioned into two or more sub-buffers, and interrupts are selected (set \( \text{DI_SEL} = 1 \) in \( \text{DMA_CONFIG} \)) to be signaled at the completion of each DMA inner loop. For example, two 512-word sub-buffers inside a 1 K-word buffer could be used to receive 16-bit peripheral data with these settings:

\[
\begin{align*}
\text{START_ADDR} & = \text{buffer base address} \\
\text{DMA_CONFIG} & = 0x10D7 (\text{FLOW} = 1, \text{DI_EN} = 1, \text{DI_SEL} = 1, \text{DMA2D} = 1,
\text{WDSIZ} = 01, \text{WNR} = 1, \text{DMA_EN} = 1) \\
\text{X_COUNT} & = 512 \\
\text{X_MODIFY} & = 2 \text{ for 16-bit data} \\
\text{Y_COUNT} & = 2 \text{ for two sub-buffers} \\
\text{Y_MODIFY} & = 2, \text{ same as X_MODIFY for contiguous sub-buffers} \\
\end{align*}
\]

In this way, a traditional double-buffer or “ping-pong” scheme could be implemented.

2-D, polled—if interrupt overhead is unacceptable but the loose synchronization of address/count register polling is acceptable, a 2-D multi-buffer synchronization scheme may be used. For example, if receive data needs to be processed in packets of sixteen 32-bit elements. A four-part 2-D DMA buffer can be allocated where each of the four sub-buffers can hold one packet with these settings:

\[
\begin{align*}
\text{START_ADDR} & = \text{buffer base address} \\
\text{DMA_CONFIG} & = 0x101B (\text{FLOW} = 1, \text{DI_EN} = 0, \text{DMA2D} = 1, \text{WDSIZ} = 10, \text{WNR} = 1, \text{DMA_EN} = 1) \\
\text{X_COUNT} & = 16 \\
\text{X_MODIFY} & = 4 \text{ for 32-bit data} \\
\end{align*}
\]
Y_COUNT = 4 for four sub-buffers

Y_MODIFY = 4, same as X_MODIFY for contiguous sub-buffers

The synchronization core might read Y_COUNT to determine which sub-buffer is currently being transferred, and then allow one full sub-buffer to account for pipelining. For example, if a read of Y_COUNT shows a value of 3, then the software should assume that sub-buffer 3 is being transferred, but some portion of sub-buffer 2 may not yet be received. The software could, however, safely proceed with processing sub-buffers 1 or 0.

1-D un synchronized FIFO—If the system design guarantees that the processing of peripheral data and the DMA rate of the data will remain correlated in the steady state, but that short-term latency variations must be tolerated, it may be appropriate to build a simple FIFO. Here, the DMA channel may be programmed using 1-D auto buffer-mode addressing without any interrupts or polling.

Descriptor Structures

DMA descriptors may be used to transfer data to or from memory data structures that are not simple 1-D or 2-D arrays. For example, if a packet of data is to be transmitted from several different locations in memory (a header from one location, a payload from a list of several blocks of memory managed by a memory-pool allocator, and a small trailer containing a checksum), a separate DMA descriptor can be prepared for each memory area, and the descriptors can be grouped in either an array or list as desired by selecting the appropriate FLOW setting in DMA_CONFIG.

The software can synchronize with the progress of the structure’s transfer by selecting interrupt notification for one or more of the descriptors. For example, the software might select interrupt notification for the header’s descriptor and for the trailer’s descriptor, but not for the payload blocks’ descriptors.
Direct Memory Access

It is important to remember the meaning of the various fields in the \texttt{DMA\_CONFIG} descriptor elements when building a list or array of DMA descriptors. In particular:

The lower byte of \texttt{DMA\_CONFIG} specifies the DMA transfer to be performed by the current descriptor (for example, interrupt-enable, 2D-mode).

The upper byte of \texttt{DMA\_CONFIG} specifies the format of the next descriptor in the chain. The \texttt{NDSIZ} and \texttt{FLOW} fields in a given descriptor do not correspond to the format of the descriptor itself; they specify the link to the next descriptor, if any.

On the other hand, when the DMA unit is being restarted, both bytes of the \texttt{DMA\_CONFIG} value written to the DMA channel’s \texttt{DMA\_CONFIG} register should correspond to the current descriptor. At a minimum, the \texttt{FLOW}, \texttt{NDSIZ}, \texttt{WNR}, and \texttt{DMA\_EN} fields must all agree with the current descriptor; the \texttt{WDSIZ}, \texttt{DI\_EN}, \texttt{DI\_SEL}, \texttt{RESTART}, and \texttt{DMA2D} fields will be taken from the \texttt{DMA\_CONFIG} value in the descriptor read from memory (and the field values initially written to the register are ignored).

Descriptor Queue Management

A system designer might want to write a DMA manager facility which accepts DMA requests from other software. The DMA manager software does not know in advance when new work requests will be received or what these requests might contain. The software could manage these transfers using a circular linked list of DMA descriptors, where each descriptor’s \texttt{NDPTR} points to the next, and the last descriptor points to the first.

The code which writes into this descriptor list could use the processor’s circular addressing modes (\texttt{Ix}, \texttt{Lx}, \texttt{Mx}, and \texttt{Bx} registers), so that it does not need to use comparison and conditional instructions to manage the circular structure. In this case, the \texttt{NDPTR} members of each descriptor could even be written once at startup, and skipped over as each descriptor’s new contents are written.
The recommended method for synchronization of a descriptor queue is through the use of an interrupt. The descriptor queue is structured so that at least the final valid descriptor is always programmed to generate an interrupt.

There are two general methods for managing a descriptor queue using interrupts:

1. interrupt on every descriptor
2. interrupt minimally - only on the last descriptor

**Descriptor Queue Using Interrupts on Every Descriptor**

In this system, the DMA manager software synchronizes with the DMA unit by enabling an interrupt on every descriptor. This method should only be used if system design can guarantee that each interrupt event will be serviced separately (no interrupt overrun).

To maintain synchronization of the descriptor queue, the non-interrupt software maintains a count of descriptors added to the queue, while the interrupt handler maintains a count of completed descriptors removed from the queue. The counts are equal only when the DMA channel is paused after having processed all the descriptors.

When each new work request is received, the DMA manager software initializes a new descriptor, taking care to write a `DMA_CONFIG` value with a `FLOW` value of 0. Next, the software compares the descriptor counts to determine if the DMA channel is running or not. If the DMA channel is paused (counts equal), the software increments its count and then starts the DMA unit by writing the new descriptor’s `DMA_CONFIG` value to the DMA channel’s `DMA_CONFIG` register.

If the counts are unequal, the software instead modifies the next-to-last descriptor’s `DMA_CONFIG` value so that its upper half (`FLOW` and `NDSIZ`) now describes the newly enqueued descriptor. This operation does not disrupt the DMA channel, provided the rest of the descriptor data structure is
initialized in advance. It is necessary, however, to synchronize the software to the DMA to correctly determine whether the new or the old DMA_CONFIG value was read by the DMA channel.

This synchronization operation should be performed in the interrupt handler. First, upon interrupt, the handler should read the channel’s IRQ_STATUS register. If the DMA_RUN status bit is set, then the channel has moved on to processing another descriptor, and the interrupt handler may increment its count and exit. If the DMA_RUN status bit is not set, however, then the channel has paused, either because there are no more descriptors to process, or because the last descriptor was enqueued too late (that is, the modification of the next-to-last descriptor’s DMA_CONFIG element occurred after that element was read into the DMA unit.) In this case, the interrupt handler should write the DMA_CONFIG value appropriate for the last descriptor to the DMA channel’s DMA_CONFIG register, increment the completed-descriptor count, and exit.

Again, this system can fail if the system interrupt latencies are large enough to cause any of the channel DMA interrupts to be dropped. An interrupt handler capable of safely synchronizing multiple descriptor interrupts would need to be complex and would need to do several MMR register accesses to ensure robust operation. In such a system environment, a minimal-interrupt synchronization method is preferred.

Descriptor Queue Using Minimal Interrupts

In this system, only one DMA interrupt event is possible in the queue at any time. The DMA interrupt handler for this system can also be extremely short. Here, the descriptor queue is organized into an “active” and a “waiting” portion, where interrupts are enabled only on the last descriptor in each portion.

When each new DMA request is processed, the software’s non-interrupt code fills in a new descriptor’s contents and adds it to the waiting portion of the queue. The descriptor’s DMA_CONFIG word should have a FLOW value of zero. If more than one request is received before the
DMA Operation Flow

DMA-queue-completion interrupt occurs, the non-interrupt code should enqueue later descriptors, forming a waiting portion of queue that is disconnected from the active portion of queue being processed by the DMA unit. In other words, all but the last active descriptors contain FLOW values >= 4 and have no interrupt enable set, while the last active descriptor contains a FLOW of 0 and an interrupt enable bit DI_EN set to 1. Also, all but the last waiting descriptors contain FLOW values >=4 and no interrupt enables set, while the last waiting descriptor contains a FLOW of 0 and an interrupt enable bit set to 1. This ensures that the DMA unit can automatically process the whole active queue and then issue one interrupt. Also, this arrangement makes it easy to start the waiting queue within the interrupt handler by a single DMA_CONFIG register write.

After enqueuing a new waiting descriptor, the non-interrupt software should leave a message for its interrupt handler in a memory mailbox location, containing the desired DMA_CONFIG value to use to start the first waiting descriptor in the waiting queue (or 0 to indicate no descriptors are waiting.)

It is critical that the software not modify the contents of the active descriptor queue directly, once its processing by the DMA unit has been started, unless careful synchronization measures are taken. In the most straightforward implementation of a descriptor queue, the DMA manager software would never modify descriptors on the active queue; instead, the DMA manager waits until the DMA queue completion interrupt indicates the processing of the entire active queue is complete.

When a DMA queue completion interrupt is received, the interrupt handler reads the mailbox from the non-interrupt software and writes the value in it to the DMA channel’s DMA_CONFIG register. This single register write restarts the queue, effectively transforming the waiting queue to an active queue. The interrupt handler should then pass a message back to the non-interrupt software indicating the location of the last descriptor accepted into the active queue. If, on the other hand, the interrupt handler reads its mailbox and finds a DMA_CONFIG value of zero, indicating
there is no more work to perform, then it should pass an appropriate mes-
sage (for example, zero) back to the non-interrupt software indicating that
the queue has stopped. This simple handler should be able to be coded in
a very small number of instructions.

The non-interrupt software which accepts new DMA work requests needs
to synchronize the activation of new work with the interrupt handler. If
the queue has stopped, that is, if the mailbox from the interrupt software
is zero, the non-interrupt software is responsible for starting the queue
(writing the first descriptor’s DMA_CONFIG value to the channel’s
DMA_CONFIG register). If the queue is not stopped, however, the non-inter-
rupt software must not write the DMA_CONFIG register (which would cause a
DMA error), but instead it should enqueue the descriptor onto the wait-
ing queue and update its mailbox directed to the interrupt handler.

More 2D DMA Examples

Examples of DMA styles supported by flex descriptors include:

- Single linear buffer that stops on completion (FLOW = stop mode).
- Linear buffer with stride greater than one (X_MODIFY > 1).
- Circular, auto-refreshing buffer that interrupts on each full buffer.
- Similar buffer that interrupts on fractional buffers
  (for example, 1/2, 1/4) (2D DMA).
- 1D DMA, using a set of identical ping-pong buffers defined by a
  linked ring of 3-word descriptors, each containing
  { link pointer, 32-bit address }.
- 1D DMA, using a linked list of 5-word descriptors containing
  { link pointer, 32-bit address, length, config } (ADSP-2191 style).
Memory DMA

- 2D DMA, using an array of 1-word descriptors, specifying only the base DMA address within a common data page.
- 2D DMA, using a linked list of 9-word descriptors, specifying everything.

Memory DMA

This section describes the memory DMA (MDMA) controllers, which provide memory-to-memory DMA transfers among the various memory spaces. These include L1 memory and external synchronous/asynchronous memories.

Each MDMA controller contains a DMA FIFO, an 8-word by 16-bit FIFO block used to transfer data to and from either L1 or the EAB bus. Typically, it is used to transfer data between external memory and internal memory. It will also support DMA from Boot ROM on the EAB bus. The FIFO can be used to hold DMA data transferred between two L1 memory locations or between two external memory locations.

Each DMA controller provides four MDMA channels:

- two source channels (for reading from memory)
- two destination channels (for writing to memory)
There are two DMA controllers, each containing two streams. The DMA controllers arbitrate between themselves, with DMA controller 0 taking priority over DMA controller 1. Within each DMA controller, the two streams have the priorities shown in Table 9-11.

Table 9-11. DMA Controller Stream Priorities

<table>
<thead>
<tr>
<th>Highest priority</th>
<th>Memory DMA destination stream D0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Memory DMA source stream S0</td>
</tr>
<tr>
<td></td>
<td>Memory DMA destination stream D1</td>
</tr>
<tr>
<td>Lowest priority</td>
<td>Memory DMA source stream S1</td>
</tr>
</tbody>
</table>

Because lower priority values take precedence over higher values, memory DMA stream 0 takes precedence over memory DMA stream 1, unless round-robin scheduling is used. Note it is illegal to program a source stream for memory write and it is illegal to program a destination stream for memory read.

The channels support 8-bit, 16-bit, and 32-bit memory DMA transfers, but both ends of the MDMA transfer must be programmed to the same word size. In other words, the MDMA transfer does not perform packing or unpacking of data; each read results in one write. Both ends of MDMA FIFO for a given stream are granted priority at the same time. Each pair shares an 8-word-deep 16-bit FIFO. The source DMA engine fills the FIFO, while the destination DMA engine empties it. The FIFO depth allows the burst transfers of the EAB and DAB buses to overlap, significantly improving throughput on block transfers between internal and external memory. Two separate descriptor blocks are required to supply the operating parameters for each MDMA pair, one for the source channel and one for the destination channel.

Because the source and destination DMA engines share a single FIFO buffer, the descriptor blocks must be configured to have the same data size. It is possible to have a different mix of descriptors on both ends as long as the total count is the same.
To start an MDMA transfer operation, the MMR registers for the source and destination streams are written, each in a manner similar to peripheral DMA. The only constraint is that the DMA_CONFIG register for the source stream must be written before the DMA_CONFIG register for the destination stream. When the destination DMA_CONFIG register is written, MDMA operation starts, after a latency of 3 SCLK cycles.

First, if either MDMA stream has been selected to use descriptors, the descriptors are fetched from memory. The destination stream descriptors are fetched first. Then, after a latency of 4 SCLK cycles after the last descriptor word is returned from memory (or typically 8 SCLK cycles after the fetch of the last descriptor word, due to memory pipelining), the source MDMA stream begins fetching data from the source buffer. The resulting data is deposited in the MDMA stream 8-location FIFO, and then after a latency of 2 SCLK cycles, the destination MDMA stream begins writing data to the destination memory buffer.

**MDMA Bandwidth**

If source and destination are in different memory spaces (one internal and one external), the internal and external memory transfers are typically simultaneous and continuous, maintaining 100% bus utilization of the internal and external memory interfaces. This performance is affected by core-to-system clock frequency ratios. At ratios below about 2.5:1, synchronization and pipeline latencies result in lower bus utilization in the system clock domain. At a clock ratio of 2:1, for example, DMA typically runs at 2/3 of the system clock rate. At higher clock ratios, full bandwidth is maintained.

If source and destination are in the same memory space (both internal or both external), the MDMA stream typically pre-fetches a burst of source data into the FIFO, and then automatically turns around and delivers all available data from the FIFO to the destination buffer. The burst length is
Dependent on traffic, and is equal to 3 plus the memory latency at the DMA in \(SCLKs\) (typically 7 for internal transfers and 6 for external transfers).

**MDMA Priority and Scheduling**

All MDMA operations have lower precedence than any peripheral DMA operations. MDMA thus makes effective use of any memory bandwidth unused by peripheral DMA traffic.

If two MDMA streams are used (S0-D0 and S1-D1), the user may choose to allocate bandwidth either by fixed stream priority or by a round-robin scheme. This is selected by programming the `MDMA_ROUND_ROBIN_PERIOD` field in the `DMAx_TCPER` register (see “Prioritization and Traffic Control” on page 9-59).

If this field is set to 0, then MDMA is scheduled by fixed priority. MDMA stream 0 takes precedence over MDMA stream 1 whenever stream 0 is ready to perform transfers. Since an MDMA stream is typically capable of transferring data on every available cycle, this could cause MDMA stream 1 traffic to be delayed for an indefinite time until any and all MDMA stream 0 operations are complete. This scheme could be appropriate in systems where low-duration but latency-sensitive data buffers need to be moved immediately, interrupting long-duration, low-priority background transfers.

If the `MDMA_ROUND_ROBIN_PERIOD` field is set to some nonzero value in the range \(1 \leq P \leq 31\), then a round robin scheduling method is used. The two MDMA streams are granted bus access in alternation in bursts of up to \(P\) data transfers. This could be used in systems where two transfer processes need to coexist, each with a guaranteed fraction of the available bandwidth. For example, one stream might be programmed for internal-to-external moves while the other is programmed for external-to-internal moves, and each would be allocated approximately equal data bandwidth.
In round robin operation, the MDMA stream selection at any time is either free or locked. Initially, the selection is free. On any free cycle available to MDMA (when no PDMA accesses take precedence), if either or both MDMA streams request access, the higher-precedence stream is granted (stream 0 in case of conflict), and that stream selection is then locked. The MDMA_ROUND_ROBIN_COUNT counter field in the DMAx_TCCNT register is loaded with the period $P$ from MDMA_ROUND_ROBIN_PERIOD, and MDMA transfers begin. The counter is decremented on every data transfer (as each data word is written to memory). After the transfer corresponding to a count of 1, the MDMA stream selection is passed automatically to the other stream with zero overhead, and the MDMA_ROUND_ROBIN_COUNT counter is reloaded with the period value $P$ from MDMA_ROUND_ROBIN_PERIOD. In this cycle, if the other MDMA stream is ready to perform a transfer, the stream selection is locked on the new MDMA stream. If the other MDMA stream is not ready to perform a transfer, then no transfer is performed, and on the next cycle the stream selection unlocks and becomes free again.

If round robin operation is used when only one MDMA stream is active, one idle cycle will occur for each $P$ MDMA data cycles, slightly lowering bandwidth by a factor of $1/(P+1)$. If both MDMA streams are used, however, memory DMA can operate continuously with zero additional overhead for alternation of streams (other than overhead cycles normally associated with reversal of read/write direction to memory, for example). By selection of various round-robin period values $P$ which limit how often the MDMA streams alternate, maximal transfer efficiency can be maintained.

**DMA Controller Errors (Aborts)**

The two DMA controllers each have the ability to generate a DMA controller error.
DMA controller errors (aborts) are detected by the DMA channel module in the cases listed below. When a DMA error occurs, the channel is immediately stopped (\texttt{DMA\_RUN} goes to 0) and any prefetched data is discarded. In addition, a \texttt{DMA\_ERROR} interrupt is asserted.

There is only one \texttt{DMA\_ERROR} interrupt for the whole DMA controller, which is asserted whenever any of the channels has detected an error condition.

The \texttt{DMA\_ERROR} interrupt handler must perform the following for each channel:

- Read each channel’s \texttt{IRQSTAT} register to look for a channel with the \texttt{DMA\_ERR} bit set (bit 1).
- Clear the problem with that channel, for example, fix register values.
- Clear the IRQ bit (write \texttt{IRQSTAT} with bit 1 = 1).

The following error conditions are detected by the DMA hardware.

- A disallowed register write occurred while the channel was running. Only the \texttt{DMA\_CONFIG} and \texttt{IRQ\_STATUS} registers can be written when \texttt{DMA\_RUN = 1}.
- An address alignment error occurred during any memory access. For example, \texttt{DMA\_CONFIG} register \texttt{WDSIZE = 1} (16 bit) but address LSB is not equal to 0, or \texttt{WDSIZE = 2} (32 bit) but two address LSBs are not equal to 00.
- A memory space transition was attempted (internal to external or vice versa).
- A memory access error occurred. Either an access attempt was made to an internal address not populated or defined as cache, or an external access caused an error (signalled by the external memory interface).
DMA Controller Errors (Aborts)

They result in a DMA abort interrupt and the configuration register contains the following invalid values.

- Incorrect WDSIZE value \((WDSIZE=11)\)
- Bit 15 not set to 0
- Incorrect FLOW value \((FLOW=2, 3, \text{ or } 5)\)
- NDSIZE value does not agree with FLOW. See Table 9-12.

Table 9-12. Legal NDSIZE Values

<table>
<thead>
<tr>
<th>FLOW</th>
<th>NDSIZE</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>(0 &lt; \text{NDSIZE} \leq 7)</td>
<td>Descriptor array, no descriptor pointer fetched</td>
</tr>
<tr>
<td>6</td>
<td>(0 &lt; \text{NDSIZE} \leq 8)</td>
<td>Descriptor list, small descriptor pointer fetched</td>
</tr>
<tr>
<td>7</td>
<td>(0 &lt; \text{NDSIZE} \leq 9)</td>
<td>Descriptor list, large descriptor pointer fetched</td>
</tr>
</tbody>
</table>

Some prohibited situations are not detected by the DMA hardware. No DMA abort is signalled for the following situations.

- DMA_CONFIG direction bit \((WNR)\) does not agree with the direction of the mapped peripheral.
- DMA_CONFIG direction bit does not agree with the direction of the MDMA channel.
- DMA_CONFIG word size \((WDSIZE)\) is not supported by the mapped peripheral.
- DMA_CONFIG word size in source and destination of the MDMA stream are not equal.
- Descriptor chain indicates data buffers that are not in the same internal/external memory space.
- In 2D DMA, \( X_{\text{COUNT}} = 1 \).

**DMA Performance: Prioritization and Optimization**

The DMA system is designed to provide maximum throughput per channel and maximum utilization of the internal buses, while accommodating the inherent latencies of memory accesses.

A key feature of the DMA architecture is the separation of the activity on the peripheral DMA bus (the DAB bus) from the activity on the buses between the DMA and memory (the DCB and DEB buses). Each peripheral DMA channel has its own data FIFO which lies between the DAB bus and the memory buses. These FIFOs automatically prefetch data from memory for transmission and buffer received data for later memory writes. This allows the peripheral to be granted a DMA transfer with very low latency compared to the total latency of a pipelined memory access, permitting the repeat rate (bandwidth) of each DMA channel to be as fast as possible.

This allows the peripheral DMA channels to have a maximum transfer rate of one 16-bit word per two system clocks per channel in either direction. The MDMA channels have a maximum transfer rate of 1 16-bit word per 1 system clock, per channel.
When all DMA channel traffic is taken in the aggregate:

- Transfers between the peripherals and the DMA unit have a maximum rate of 1 16-bit transfer per system clock.
- Transfers between the DMA unit and internal memory (L1) have a maximum rate of 1 16-bit transfer per system clock.
- Transfers between the DMA unit and external memory have a maximum rate of 1 16-bit transfer per system clock.

Some considerations which limit the actual performance are:

- Accesses to internal or external memory which conflict with core accesses to the same memory. This can cause delays, for example, for accessing the same L1 bank, for opening/closing SDRAM pages, or while filling cache lines.
- Each direction change from RX to TX on the DAB bus imposes a 1-clock delay.
- Direction changes on the DCB bus (for example, write followed by read) to the same bank of internal memory can impose delays.
- Direction changes (for example, read followed by write) on the DEB bus to external memory can each impose a several-cycle delay.
- MMR accesses to registers other than \texttt{DMAx_CONFIG}, \texttt{DMAx_IRQ_STATUSF}, or \texttt{DMAx_PERIPHERAL_MAP} will stall all DMA activity for 1 cycle per 16-bit word transferred. In contrast, MMR accesses to the control/register registers do not cause stalls or wait states.
- Reads from registers other than control/register registers use one PAB bus wait state, delaying the core for several core clocks.
Direct Memory Access

- Descriptor fetches consume one DMA memory cycle per 16-bit word read from memory, but do not delay transfers on the DAB bus.

- Initialization of a DMA channel stalls DMA activity for one cycle. This occurs when DMA_EN changes from 0 to 1 or when the RESTART bit is set to 1 in the DMAx_CONFIG register.

Several of these factors may be minimized by proper design of the application software. It is often possible to structure the software to avoid internal and external memory conflicts by careful allocation of data buffers within banks and pages, and by planning for low cache activity during critical DMA operations. Furthermore, unnecessary MMR accesses can be minimized, especially by using descriptors or auto buffering.

Efficiency loss caused by excessive direction changes (thrashing) can be minimized by the processor’s traffic control features, described in the next section.

Prioritization and Traffic Control

DMA channels are ordinarily granted service strictly according to their priority. The priority of a channel is simply its channel number, where lower priority numbers are granted first. Thus, peripherals with high data rates or low latency requirements should be assigned to lower numbered (higher priority) channels using the DMAx_PERIPHERAL_MAP registers. The memory DMA streams are always lower priority than the peripherals, but as they request service continuously, they ensure that any time slots unused by peripheral DMA are applied to MDMA transfers. By default, when more than one MDMA stream is enabled and ready, only the highest-priority MDMA stream is granted. If it is desirable for the MDMA streams to share the available bandwidth, however, the MDMAROUND_ROBIN_PERIOD may be programmed to select each stream in turn for a fixed number of transfers.
In the processor DMA, there are two completely separate but simultaneous prioritization processes: the DAB bus prioritization and the memory bus (DCB and DEB) prioritization. Peripherals that are requesting DMA via the DAB bus, and whose data FIFOs are ready to handle the transfer, compete with each other for DAB bus cycles. Similarly but separately, channels whose FIFOs need memory service (prefetch or post-write) compete together for access to the memory buses. MDMA streams compete for memory access as a unit, and source and destination may be granted together if their memory transfers do not conflict. In this way, internal-to-external or external-to-internal memory transfers may occur at the full system clock rate (SCLK). Examples of memory conflict include simultaneous access to the same memory space and simultaneous attempts to fetch descriptors. Special processing may occur if a peripheral is requesting DMA but its FIFO is not ready (for example, an empty transmit FIFO or full receive FIFO). For more information, see “Urgent DMA Transfers” on page 9-64.

Traffic control is an important consideration in optimizing use of DMA resources. Traffic control is a way to influence how often the transfer direction on the data buses may change, by automatically grouping same-direction transfers together. The DMA block provides a traffic control mechanism controlled by the DMACx_TC_PER and DMACx_TC_CNT registers. This mechanism performs the optimization without real-time processor intervention, and without the need to program transfer bursts into the DMA work unit streams. Traffic can be independently controlled for each of the three buses (DAB, DCB, and DEB) with simple counters. In addition, alternation of transfers among MDMA streams can be controlled with the MDMA_ROUND_ROBIN_COUNT field of the DMACx_TC_CNT register.

Using the traffic control features, the DMA system preferentially grants data transfers on the DAB or memory buses which are going in the same read/write direction as the previous transfer, until either the traffic control counter times out, or until traffic stops or changes direction on its own.
When the traffic counter reaches zero, the preference is changed to the opposite flow direction. These directional preferences work as if the priority of the opposite-direction channels were decreased by 16.

For example, if channels 3 and 5 were requesting DAB access, but lower-priority channel 5 is going “with traffic” and higher-priority channel 3 is going “against traffic,” then channel 3’s effective priority becomes 19, and channel 5 would be granted instead. If on the next cycle the only channels requesting DAB transfers were all going against traffic, (channels 3 and 6), then their effective priorities become 19 and 22. One of the channels (channel 3) is granted, even though its direction is opposite to the current flow. No bus cycles are wasted, other than any necessary delay required by the bus turnaround.

This type of traffic control represents a trade-off of latency to improve utilization (efficiency). Higher traffic timeouts might increase the length of time each request waits for its grant, but it often dramatically improves the maximum attainable bandwidth in congested systems, often to above 90%.

To disable preferential DMA prioritization, program the `DMACx_TC_PER` register to 0x0000.

**DMA Traffic Control Counter Period (DMACx_TC_PER) and Counter (DMACx_TC_CNT) Registers**

The `MDMA_ROUND_ROBIN_COUNT` field shows the current transfer count remaining in the MDMA round robin period. It initializes to `MDMA_ROUND_ROBIN_PERIOD` whenever `DMACx_TC_PER` is written, whenever a different MDMA stream is granted, or whenever every MDMA stream is idle, then counts down to 0 with each MDMA transfer. When this count decrements from 1 to 0, the next available MDMA stream is selected.
Prioritization and Traffic Control

The \texttt{DAB\_TRAFFIC\_COUNT} field shows the current cycle count remaining in the DAB traffic period. It initializes to \texttt{DAB\_TRAFFIC\_PERIOD} whenever \texttt{DMACx\_TC\_PER} is written, or whenever the DAB bus changes direction or becomes idle, then counts down from \texttt{DAB\_TRAFFIC\_PERIOD} to 0 on each system clock (except for DMA stalls). While this count is nonzero, same-direction DAB accesses are preferred. When this count decrements from 1 to 0, the opposite-direction DAB access is preferred, which may result in a direction change. When this count is 0 and a DAB bus access occurs, the count is reloaded from \texttt{DAB\_TRAFFIC\_PERIOD} to begin a new burst.

The \texttt{DEB\_TRAFFIC\_COUNT} field shows the current cycle count remaining in the DEB traffic period. It initializes to \texttt{DEB\_TRAFFIC\_PERIOD} whenever \texttt{DMACx\_TC\_PER} is written, or whenever the DEB bus changes direction or becomes idle, then counts down from \texttt{DEB\_TRAFFIC\_PERIOD} to 0 on each system clock (except for DMA stalls). While this count is nonzero, same-direction DEB accesses are preferred. When this count decrements from 1 to 0, the opposite-direction DEB access is preferred, which may result in a direction change. When this count is 0 and a DEB bus access occurs, the count is reloaded from \texttt{DEB\_TRAFFIC\_PERIOD} to begin a new burst.

The \texttt{DCB\_TRAFFIC\_COUNT} field shows the current cycle count remaining in the DCB traffic period. It initializes to \texttt{DCB\_TRAFFIC\_PERIOD} whenever \texttt{DMACx\_TC\_PER} is written, or whenever the DCB bus changes direction or becomes idle, then counts down from \texttt{DCB\_TRAFFIC\_PERIOD} to 0 on each system clock (except for DMA stalls). While this count is nonzero, same-direction DCB accesses are preferred. When this count decrements from 1 to 0, the opposite-direction DCB access is preferred, which may result in a direction change. When this count is 0 and a DCB bus access occurs, the count is reloaded from \texttt{DCB\_TRAFFIC\_PERIOD} to begin a new burst.
Direct Memory Access

DMA Traffic Control Counter Period Registers (DMACx_TC_PER)

**Figure 9-17. DMA Traffic Control Counter Period Register**

**DMA Traffic Control Counter Registers (DMACx_TC_CNT)**

**Figure 9-18. DMA Traffic Control Counter Register**
Typically, DMA transfers for a given peripheral occur at regular intervals. Generally, the shorter the interval, the higher the priority that should be assigned to the peripheral. If the average bandwidth of all the peripherals is not too large a fraction of the total, then all peripherals’ requests should be granted as required.

Occasionally, instantaneous DMA traffic might exceed the available bandwidth, causing congestion. This may occur if L1 or external memory is temporarily stalled, perhaps for an SDRAM page swap or a cache line fill. Congestion might also occur if one or more DMA channels initiates a flurry of requests, perhaps for descriptor fetches or to fill a FIFO in the DMA or in the peripheral.

If congestion persists, lower priority DMA peripherals may become starved for data. Even though the peripheral’s priority is low, if the necessary data transfer does not take place before the end of the peripheral’s regular interval, system failure may result. To minimize this possibility, the DMA unit detects peripherals whose need for data has become urgent, and preferentially grants them service at the highest priority.

A DMA channel’s request for memory service is defined as urgent if both the channel’s FIFO is not ready for a DAB bus transfer (that is, a transmit FIFO is empty or a receive FIFO is full), and the peripheral is asserting its DMA request line.

Descriptor fetches may be urgent, if they are necessary to initiate or continue a DMA work unit chain for a starving peripheral. DMA requests from an MDMA channel are never urgent.
When one or more DMA channels express an urgent memory request, two events occur:

1. All non-urgent memory requests are decreased in priority by 32, guaranteeing that only an urgent request will be granted. The urgent requests compete with each other, if there is more than one, and directional preference among urgent requests is observed.

2. The resulting memory transfer is marked for expedited processing in the targeted memory system (L1 or external), and so are all prior incomplete memory transfers ahead of it in that memory system. This may cause a series of external memory core accesses to be delayed for a few cycles so that a peripheral’s urgent request may be accommodated.

The preferential handling of urgent DMA transfers is completely automatic. No user controls are required for this function to operate.
The processor has three serial peripheral interface (SPI) ports that provide an I/O interface to a wide variety of SPI compatible peripheral devices.

With a range of configurable options, the SPI ports provide glueless hardware interface to other SPI compatible devices. The SPI is a four-wire interface consisting of two data pins, a device select pin, and a clock pin. The SPI is a full-duplex synchronous serial interface, supporting master modes, slave modes, and multimaster environments. The SPI compatible peripheral implementation also supports programmable baud rate and clock phase/polarities. The SPI features the use of open drain drivers to support the multimaster scenario and to avoid data contention.

Typical SPI compatible peripheral devices that can be used to interface to the SPI compatible interface include:

- Other CPUs or micro controllers
- Codecs
- A/D converters
- D/A converters
- Sample rate converters
- SP/DIF or AES/EBU digital audio transmitters and receivers
- LCD displays
The SPI is an industry-standard synchronous serial link that supports communication with multiple SPI compatible devices. The SPI peripheral is a synchronous, four-wire interface consisting of two data pins (\textsc{Mosi}\textsubscript{x} and \textsc{Miso}\textsubscript{x}), one device select pin (\textsc{Spi}\textsubscript{x}\textsubscript{SS}), and a gated clock pin (\textsc{sck}\textsubscript{x}). With the two data pins, it allows for full-duplex operation to other SPI compatible devices. The SPI also includes programmable baud rates, clock phase, and clock polarity.

The SPI can operate in a multimaster environment by interfacing with several other devices, acting as either a master device or a slave device. In a multimaster environment, the SPI interface uses open drain outputs to avoid data bus contention.

**Figure 10-1** provides a block diagram of the SPI. The interface is essentially a shift register that serially transmits and receives data bits, one bit at a time at the \textsc{sck}\textsubscript{x} rate, to and from other SPI devices. SPI data is transmitted and received at the same time through the use of a shift register. When an SPI transfer occurs, data is simultaneously transmitted (shifted serially out of the shift register) as new data is received (shifted serially into the other end of the same shift register). The \textsc{sck}\textsubscript{x} synchronizes the shifting and sampling of the data on the two serial data pins.

During SPI data transfers, one SPI device acts as the SPI link master, where it controls the data flow by generating the SPI serial clock and asserting the SPI device select signal (\textsc{spi}\textsubscript{x}\textsubscript{SS}). The other SPI device acts as the slave and accepts new data from the master into its shift register, while it transmits requested data out of the shift register through its SPI transmit data pin. Multiple processors can take turns being the master device, as can other micro controllers or microprocessors. One master device can also simultaneously shift data into multiple slaves (known as broadcast mode). However, only one slave may drive its output to write data back to the master at any given time. This must be enforced in broadcast mode on
SPI Compatible Port Controllers

SPI0, where several slaves can be selected to receive data from the master, but only one slave at a time can be enabled to send data back to the master.

In a multimaster or multidevice environment where multiple processors are connected via their SPI ports, all MOSI pins are connected together, all MISO pins are connected together, and all SCK pins are connected together.

For a multislave environment, the processor can make use of seven GPIO port F, PF1–PF7, that are dedicated SPI0 slave select signals for the SPI0 slave devices. For SPI1 and SPI2, the processor uses a single GPIO pin for a single slave-select output each. SPI1 uses the PD4 pin, and SPI2 uses the PD9 pin in this capacity.

At reset, the SPI is disabled and configured as a slave.
Interface Signals

The following section discusses the SPI signals.

Serial Peripheral Interface Clock Signals (SCKx)

The \texttt{SCKx} signal is the SPI clock signal. This control signal is driven by the master and controls the rate at which data is transferred. The master may transmit data at a variety of baud rates. The \texttt{SCKx} signal cycles once for each bit transmitted. It is an output signal if the device is configured as a master, and an input signal if the device is configured as a slave.

The \texttt{SCKx} is a gated clock that is active during data transfers only for the length of the transferred word. The number of active clock edges is equal to the number of bits driven on the data lines. Slave devices ignore the serial clock if the serial peripheral slave select input (\texttt{SPI\textsubscript{x}SS}) is driven inactive (high).

The \texttt{SCKx} is used to shift out and shift in the data driven on the \texttt{MISOx} and \texttt{MOSIx} lines. Clock polarity and clock phase relative to data are programmable in the SPI control register (\texttt{SPI\textsubscript{x}CTL}) and define the transfer format (see “SPI Transfer Formats” on page 10-20).

The \texttt{SCK0} signal is dedicated. The \texttt{SCK1} and \texttt{SCK2} signals are GPIO pins \texttt{PD2} and \texttt{PD7}, respectively. Be sure to set the \texttt{PORTDIO_FER} register for peripheral use. For more information, see “General-Purpose Input/Output Ports C, D, E” on page 15-1.

Serial Peripheral Interface Slave Select Input Signals (SPI\textsubscript{x}SS)

The \texttt{SPI\textsubscript{x}SS} signal is the SPI serial peripheral slave select input signal. This is an active-low signal used to enable a processor when it is configured as a slave device. This input-only pin behaves like a chip select and is provided by the master device for the slave devices. For a master device, it can act as
SPI Compatible Port Controllers

an error signal input in case of the multimaster environment. In multi-
master mode, if the SPIxSS input signal of a master is asserted (driven low), and the PSSE bit in the SPIx_CTL register is enabled, an error has occurred. This means that another device is also trying to be the master device.

The SPI0SS signal is the same pin as the PF0 pin. Be careful to not use PF0 as an output if intended to serve as the SPI0SS. The SPI1SS and SPI2SS signals are GPIO pins (PD3 and PD8, respectively). Be sure to set the PORTDIO_FER register for peripheral use. For more information, see “General-Purpose Input/Output Ports C, D, E” on page 15-1.

Master Out Slave In (MOSIx)

The MOSI0 pin is dedicated, and the MOSI1 and MOSI2 pins are GPIO pins (PD0 and PD5, respectively). Be sure to set the PORTDIO_FER register for peripheral use. For more information, see “General-Purpose Input/Output Ports C, D, E” on page 15-1.

The MOSI0x pin is the master-out-slave-in pin, one of the bidirectional I/O data pins. If the processor is configured as a master, the MOSI0x pin becomes a data transmit (output) pin, transmitting output data. If the processor is configured as a slave, the MOSI0x pin becomes a data receive (input) pin, receiving input data. In an SPI interconnection, the data is shifted out from the MOSI0x output pin of the master and shifted into the MOSI0x input(s) of the slave(s).

Master In Slave Out (MISOx)

The MISOx pin is the master-in-slave-out pin, one of the bidirectional I/O data pins. If the processor is configured as a master, the MISOx pin becomes a data receive (input) pin, receiving input data. If the processor is configured as a slave, the MISOx pin becomes a data transmit (output) pin,
transmitting output data. In an SPI interconnection, the data is shifted out from the MISOx output pin of the slave and shifted into the MISOx input pin of the master. The following important points should be noted.

- The MISO0 pin is dedicated, and the MISO1 and MISO2 pins are GPIO pins (PD1 and PD6, respectively). Be sure to set the PORTDIO_FER register for peripheral use. For more information, see “General-Purpose Input/Output Ports C, D, E” on page 15-1.

- In a multislave environment, only one slave is allowed to transmit data at any given time.

- The processor can be booted via its SPI0 interface to allow user application code and data to be downloaded before runtime.

The SPI configuration example in Figure 10-2 illustrates how the processor can be used as the slave SPI device. The 8-bit host microcontroller is the SPI master.

![Figure 10-2. ADSP-BF538 Blackfin Processor as a Slave SPI Device](image)

**Interrupt Output**

Each SPI has two interrupt output signals: a data interrupt and an error interrupt.
The behavior of the SPI data interrupt signal depends on the transfer Initiation mode bit field (TIMOD) in the SPI control register. In DMA mode (TIMOD = 1X), the data interrupt acts as a DMA request and is generated when the DMA FIFO is ready to be written to (TIMOD = 11) or read from (TIMOD = 10). In non-DMA mode (TIMOD = 0X), a data interrupt is generated when the SPIx_TDBR is ready to be written to (TIMOD = 01) or when the SPIx_RDBR is ready to be read from (TIMOD = 00).

An SPI error interrupt is generated in a master when a mode fault error occurs, in both DMA and non-DMA modes. An error interrupt can also be generated in DMA mode when there is an underflow (TXE when TIMOD = 11) or an overflow (RBSY when TIMOD = 10) error condition. In non-DMA mode, the underflow and overflow conditions set the TXE and RBSY bits in the SPIx_STAT register, respectively, but do not generate an error interrupt.

For more information about this interrupt output, see the discussion of the TIMOD bits in “SPI Control (SPIx_CTL) Register” on page 10-9.

**SPI Registers**

The SPI peripherals include a number of user-accessible registers. Some of these registers are also accessible through the DMA bus. Four registers contain control and status information: SPIx_BAUD, SPIx_CTL, SPIx_FLG, and SPIx_STAT. Two registers are used for buffering receive and transmit data: SPIx_RDBR and SPIx_TDBR. For information about DMA-related registers, see Chapter 9, “Direct Memory Access”. The shift register, SFDR, is internal to the SPI module and is not directly accessible.

See “Error Signals and Flags” on page 10-28 for more information about how the bits in these registers are used to signal errors and other conditions. See “Register Functions” on page 10-19 for more information about SPI register and bit functions.
SPI Registers

**SPI BAUD Rate (SPIx_BAUD) Register**

The SPI baud rate register (SPIx_BAUD) is used to set the bit transfer rate for a master device. When configured as a slave, the value written to this register is ignored. The serial clock frequency is determined by this formula:

\[
SCKx \text{ frequency} = \frac{\text{peripheral clock frequency SCLK}}{(2 \times SPIx\_BAUD)}
\]

Writing a value of 0 or 1 to the register disables the serial clock. Therefore, the maximum serial clock rate is one-fourth the system clock rate.

**Table 10-1. SPI Master Baud Rate Example**

<table>
<thead>
<tr>
<th>SPIx_BAUD Decimal Value</th>
<th>SPI Clock (SCKx) Divide Factor</th>
<th>Baud Rate for SCLK at 100 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>1</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>25 MHz</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>16.7 MHz</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>12.5 MHz</td>
</tr>
<tr>
<td>65,535 (0xFFFF)</td>
<td>131,070</td>
<td>763 Hz</td>
</tr>
</tbody>
</table>

Figure 10-3. SPI Baud Rate Registers

Table 10-1 lists several possible baud rate values for SPIx_BAUD.
SPI Control (SPIx_CTL) Register

The SPI control register (SPIx_CTL) is used to configure and enable the SPI system. This register is used to enable the SPI interface, select the device as a master or slave, and determine the data transfer format and word size.

The term “word” refers to a single data transfer of either 8 bits or 16 bits, depending on the word length (SIZE) bit in SPIx_CTL. There are two special bits which can also be modified by the hardware: SPE and MSTR.

The TIMOD field is used to specify the action that initiates transfers to/from the receive/transmit buffers. When set to 00, a SPI port transaction is begun when the receive buffer is read. Data from the first read needs to be discarded since the read is needed to initiate the first SPI port transaction. When set to 01, the transaction is initiated when the transmit buffer is written. A value of 10 selects DMA receive mode and the first transaction is initiated by enabling the SPI for DMA receive mode. Subsequent individual transactions are initiated by a DMA read of the SPIx_RDBR. A value of 11 selects DMA transmit mode and the transaction is initiated by a DMA write of the SPIx_TDBR.

The PSSE bit is used to enable the SPIxSS input for master. When not used, SPIxSS can be disabled, freeing up a chip pin as general-purpose I/O.

The EMISO bit enables the MISOx pin as an output. This is needed in an environment where the master wishes to transmit to various slaves at one time (broadcast). Only one slave is allowed to transmit data back to the master. Except for the slave from whom the master wishes to receive, all other slaves should have this bit cleared.

The SPE and MSTR bits can be modified by hardware when the MODF bit of the status register is set. See “Mode Fault Error (MODF)” on page 10-28.
Figure 10-4 provides the bit descriptions for SPIx_CTL.

**SPI Control Register (SPIx_CTL)**

SPI0 – 0xFFC0 0500
SPI1 – 0xFFC0 2300
SPI1 – 0xFFC0 2400

Reset = 0x0400

**SPE (SPI Enable)**
- 0: Disabled
- 1: Enabled

**WOM (Write Open Drain Master)**
- 0: Normal
- 1: Open drain

**MSTR (Master)**
Sets the SPI module as master or slave
- 0: Slave
- 1: Master

**CPOL (Clock Polarity)**
- 0: Active high SCKx
- 1: Active low SCKx

**CPHA (Clock Phase)**
Selects transfer format and operation mode
- 0: SCLK toggles from middle of the first data bit, slave select pins controlled by hardware.
- 1: SCLK toggles from beginning of first data bit, slave select pins controller by user software.

**LSBF (LSB First)**
- 0: MSB sent/received first
- 1: LSB sent/received first

**SIZE (Size of Words)**
- 0: 8 bits
- 1: 16 bits

**TIMOD (Transfer Initiation Mode)**
- 00: Start transfer with read of SPIx_RDBR, interrupt when SPIx_RDBR is full
- 01: Start transfer with write of SPIx_TDBR, interrupt when SPIx_TDBR is empty
- 10: Start transfer with DMA read of SPIx_RDBR, request further DMA reads as long as SPI DMA FIFO is not empty
- 11: Start transfer with DMA write of SPIx_TDBR, request further DMA writes as long as SPI DMA FIFO is not full

**SZ (Send Zero)**
Send zero or last word when SPIx_TDBR is empty
- 0: Send last word
- 1: Send zeros

**GM (Get More Data)**
When SPIx_RDBR is full, get data or discard incoming data
- 0: Discard incoming data
- 1: Get more data, overwrite previous data

**PSSE (Slave Select Enable)**
- 0: Disable
- 1: Enable

**EMISO (Enable MISOx)**
- 0: MISOx disabled
- 1: MISOx enabled

Figure 10-4. SPI Control Register
SPI Flag (SPIx_FLG) Register

If the SPI is enabled as a master, the SPI uses the SPI flag register (SPIx_FLG) to enable up to seven GPIO port F pins to be used as individual slave select lines. In slave mode, the SPIx_FLG bits have no effect, and each SPI uses the SPIxSS input as a slave select. Figure 10-5 shows the SPI0_FLAG register diagram. For details specific to the SPI1 and SPI2 ports, see “Special Considerations for SPI1 and SPI2 Slave Control” on page 10-15.

Figure 10-5. SPI0 Flag Register
SPI Registers

The SPI0_FLG register consists of two sets of bits that function as follows.

- **Slave select enable (FLSx) bits**

  Each FLSx bit corresponds to a GPIO port F (PFx) pin. When a FLSx bit is set, the corresponding PFx pin is driven as a slave select. For example, if FLS1 is set in SPI0_FLG, PF1 is driven as a slave select (SPI0SEL). Table 10-2 shows the association of the FLSx bits and the corresponding PFx pins.

  If the FLSx bit is not set, the GPIO port F registers (PORTFIO_DIR and others) configure and control the corresponding PFx pin for SPI0.

- **Slave select value (FLGx) bits**

- **When a PFx pin is configured as a slave select output for SPI0, the FLGx bits can determine the value driven onto the output.** If the CPHA bit in SPI0_CTL is set, the output value is set by software control of the FLGx bits. The SPI protocol permits the slave select line to either remain asserted (low) or be deasserted between transferred words. The user must set or clear the appropriate FLGx bits. For example, to drive PF3 as a slave select, FLS3 in SPI0_FLG must be set. Clearing FLG3 in SPI0_FLG drives PF3 low; setting FLG3 drives PF3 high. The PF3 pin can be cycled high and low between transfers by setting and clearing FLG3. Otherwise, PF3 remains active (low) between transfers.

  If CPHA = 0, the SPI hardware sets the output value and the FLGx bits are ignored. The SPI protocol requires that the slave select be deasserted between transferred words. In this case, the SPI hardware controls the pins. For example, to use PF3 as a slave select pin, it is only necessary to set the FLS3 bit in SPIx_FLG. It is not necessary to write to the FLG3 bit, because the SPI hardware automatically drives the PF3 pin.
Table 10-2. SPI0_FLG Bit Mapping to PFx Pins

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Function</th>
<th>PFx Pin</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Reserved</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>FLS1</td>
<td>SPI0SEL1 Enable</td>
<td>PF1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>FLS2</td>
<td>SPI0SEL2 Enable</td>
<td>PF2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>FLS3</td>
<td>SPI0SEL3 Enable</td>
<td>PF3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>FLS4</td>
<td>SPI0SEL4 Enable</td>
<td>PF4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>FLS5</td>
<td>SPI0SEL5 Enable</td>
<td>PF5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>FLS6</td>
<td>SPI0SEL6 Enable</td>
<td>PF6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>FLS7</td>
<td>SPI0SEL7 Enable</td>
<td>PF7</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>Reserved</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>FLG1</td>
<td>SPI0SEL1 Value</td>
<td>PF1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>FLG2</td>
<td>SPI0SEL2 Value</td>
<td>PF2</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>FLG3</td>
<td>SPI0SEL3 Value</td>
<td>PF3</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>FLG4</td>
<td>SPI0SEL4 Value</td>
<td>PF4</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>FLG5</td>
<td>SPI0SEL5 Value</td>
<td>PF5</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>FLG6</td>
<td>SPI0SEL6 Value</td>
<td>PF6</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>FLG7</td>
<td>SPI0SEL7 Value</td>
<td>PF7</td>
<td>1</td>
</tr>
</tbody>
</table>

**Slave Select Inputs**

If the SPI is in slave mode, SPIxSS acts as the slave select input. When enabled as a master, SPIxSS can serve as an error detection input for the SPI in a multimaster environment. The PSSE bit in SPIx_CTL enables this feature. When PSSE = 1, the SPIxSS input is the master mode error input. Otherwise, SPIxSS is ignored.
Use of FLS Bits in SPI0_FLG for Multiple Slave
SPI Systems

The FLSx bits in the SPI0_FLG register are used in a multiple slave SPI environment. For example, if there are eight SPI devices in the system including a processor master, the master processor can support the SPI mode transactions across the other seven devices. This configuration requires only one master processor in this multislave environment. For example, assume that the SPI0 is the master. The seven GPIO port F pins (PF1-PF7) on the processor master can be connected to each of the slave SPI device’s SPIxSS slave-select input pins. In this configuration, the FLSx bits in SPI0_FLG can be used in three cases.

In cases 1 and 2, the processor is the master and the seven micro controllers/peripherals with SPI interfaces are slaves. The processor can:

1. Transmit to all seven SPI devices at the same time in a broadcast mode. Here, all FLSx bits are set.

2. Receive and transmit from one SPI device by enabling only one slave SPI device at a time.

   In case 3, all eight devices connected via SPI ports can be other processors.

3. If all the slaves are also processors, then the requester can receive data from only one processor (enabled by clearing the EMISO bit in the six other slave processors) at a time and transmit broadcast data to all seven at the same time. This EMISO feature may be available in some other micro controllers. Therefore, it is possible to use the EMISO feature with any other SPI device that includes this functionality.

Figure 10-6 shows one processor as a master with three processors (or other SPI compatible devices) as slaves.
Special Considerations for SPI1 and SPI2 Slave Control

All functionality and control for the slave-select outputs for SPI1 and SPI2 are exactly as described for SPI0 above. However, since SPI1 and SPI2 can only control one slave-select output signal, the only functional bits in the relevant SPIx_FLG registers are the FLS1/FLG1 bits. The rest of the bits in these registers are reserved. For SPI1, modifying these bit locations affects the PD4 pin. For SPI2, modifying these bit locations affects the PD9 pin. Be sure to verify that the PORTDIO_FER register is properly configured to set these pins for peripheral use. See Figure 15-8 on page 15-10 for more details.

SPI Status (SPIx_STAT) Register

The SPI status register (SPIx_STAT) is used to detect when an SPI transfer is complete or if transmission/reception errors occur. The SPIx_STAT register can be read at any time.
Some of the bits in SPIx_STAT are read-only and other bits are sticky. Bits that provide information only about the SPI are read-only. These bits are set and cleared by the hardware. Sticky bits are set when an error condition occurs. These bits are set by hardware and must be cleared by software. To clear a sticky bit, the user must write a 1 to the desired bit position of SPIx_STAT. For example, if the TXE bit is set, the user must write a 1 to bit 2 of SPIx_STAT to clear the TXE error condition. This allows the user to read SPIx_STAT without changing its value.

Sticky bits are cleared on a reset, but are not cleared on an SPI disable.

SPI Status Register (SPIx_STAT)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>SPIF (SPI Finished)</td>
<td>RO</td>
<td>Set when SPI single word transfer complete</td>
</tr>
<tr>
<td>14</td>
<td>MODF (Mode Fault Error)</td>
<td>W1C</td>
<td>Set in a master device when some other device tries to become the master</td>
</tr>
<tr>
<td>13</td>
<td>TXE (Transmission Error)</td>
<td>W1C</td>
<td>Set when transmission occurred with no new data in SPIx_TDBR</td>
</tr>
<tr>
<td>12</td>
<td>TXCOL (Transmit Collision Error)</td>
<td>W1C</td>
<td>When set, corrupt data may have been transmitted</td>
</tr>
<tr>
<td>11</td>
<td>RXS (RX Data Buffer register)</td>
<td>RO</td>
<td>0 - Empty, 1 - Full</td>
</tr>
<tr>
<td>10</td>
<td>RBSY (Receive Error)</td>
<td>W1C</td>
<td>Set when data is received with receive buffer full</td>
</tr>
<tr>
<td>9</td>
<td>TXS (SPIx_TDBR Data Buffer register)</td>
<td>RO</td>
<td>0 - Empty, 1 - Full</td>
</tr>
</tbody>
</table>

Figure 10-7. SPIx Status Register

The transmit buffer becomes full after it is written to. It becomes empty when a transfer begins and the transmit value is loaded into the shift register. The receive buffer becomes full at the end of a transfer when the shift register value is loaded into the receive buffer. It becomes empty when the receive buffer is read.
The SPIF bit is set when the SPI port is disabled.

Upon entering DMA mode, the transmit buffer and the receive buffer become empty. That is, the TXS bit and the RXS bit are initially cleared upon entering DMA mode.

When using DMA for SPI transmit, the DMA_DONE interrupt signifies that the DMA FIFO is empty. However, at this point there may still be data in the SPI DMA FIFO waiting to be transmitted. Therefore, software needs to poll TXS in the SPIx_STAT register until it goes low for two successive reads, at which point the SPI DMA FIFO will be empty. When the SPIF bit subsequently goes high, the last word has been transferred.

SPI Transmit Data Buffer (SPIx_TDBR) Register

The SPI transmit data buffer register (SPIx_TDBR) is a 16-bit read-write register. Data is loaded into this register before being transmitted. Just prior to the beginning of a data transfer, the data in SPIx_TDBR is loaded into the shift data register (SFDR). A read of SPIx_TDBR can occur at any time and does not interfere with or initiate SPI transfers.

When the DMA is enabled for transmit operation, the DMA engine loads data into this register for transmission just prior to the beginning of a data transfer. A write to SPIx_TDBR should not occur in this mode because this data overwrites the DMA data to be transmitted.

When the DMA is enabled for receive operation, the contents of SPIx_TDBR are repeatedly transmitted. A write to SPIx_TDBR is permitted in this mode, and this data is transmitted.

If the send zeros control bit (SZ in the SPIx_CTL register) is set, SPIx_TDBR may be reset to 0 under certain circumstances.
If multiple writes to SPIx_TDBR occur while a transfer is already in progress, only the last data written is transmitted. None of the intermediate values written to SPIx_TDBR are transmitted. Multiple writes to SPIx_TDBR are possible, but not recommended.

**SPI Receive Data Buffer (SPIx_RDBR) Register**

The SPI receive data buffer register (SPIx_RDBR) is a 16-bit read-only register. At the end of a data transfer, the data in the shift register is loaded into SPIx_RDBR. During a DMA receive operation, the data in SPIx_RDBR is automatically read by the DMA. When SPIx_RDBR is read via software, the RXS bit is cleared and an SPI transfer may be initiated (if TIMOD = 00).
SPI Receive Data Buffer Shadow (SPIx_SHADOW) Register

The SPI RDBR shadow register (SPIx_SHADOW), has been provided for use in debugging software. This register is at a different address than the receive data buffer, SPIx_RDBR, but its contents are identical to that of SPIx_RDBR. When a software read of SPIx_RDBR occurs, the RXS bit in SPIx_STAT is cleared and an SPI transfer may be initiated (if TIMOD = 00 in SPIx_CTL). No such hardware action occurs when the SPIx_SHADOW register is read. The SPIx_SHADOW register is read-only.

**Figure 10-10. SPIx RDBR Shadow Register**

**Register Functions**

Table 10-3 summarizes the functions of the SPI registers.

**Table 10-3. SPI Register Mapping**

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Function</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPIx_CTL</td>
<td>SPI port control</td>
<td>SPE and MSTR bits can also be modified by hardware (when MODF is set)</td>
</tr>
<tr>
<td>SPIx_FLG</td>
<td>SPI port flag</td>
<td>Bits 0 and 8 are reserved in SPI0_FLG. All bits except FLS1 and FLG1 are reserved in SPI1_FLG and SPI2_FLG.</td>
</tr>
</tbody>
</table>
SPI Transfer Formats

The SPI supports four different combinations of serial clock phase and polarity (SPI modes 0-3). These combinations are selected using the \texttt{CPOL} and \texttt{CPHA} bits in \texttt{SPIx_CTL}, as shown in Figure 10-11.

The figures “SPI Transfer Protocol for CPHA = 0” on page 10-22 and “SPI Transfer Protocol for CPHA = 1” on page 10-22 demonstrate the two basic transfer formats as defined by the \texttt{CPHA} bit. Two waveforms are shown for \texttt{SCKx}—one for \texttt{CPOL} = 0 and the other for \texttt{CPOL} = 1. The diagrams may be interpreted as master or slave timing diagrams since the \texttt{SCKx}, \texttt{MISOx}, and \texttt{MOSIx} pins are directly connected between the master and the slave. The \texttt{MISOx} signal is the output from the slave (slave transmission), and the \texttt{MOSIx} signal is the output from the master (master transmission). The \texttt{SCKx} signal is generated by the master, and the \texttt{SPIxSS} signal is the slave device select input to the slave from the master. The diagrams represent an 8-bit transfer (\texttt{SIZE} = 0) with the most significant bit.

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Function</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPIx_STAT</td>
<td>SPI port status</td>
<td>SPIF bit can be set by clearing SPE in SPIx_CTL</td>
</tr>
<tr>
<td>SPIx_TDBR</td>
<td>SPI port transmit data buffer</td>
<td>Register contents can also be modified by hardware (by DMA and/or when \texttt{SZ} = 1 in SPIx_CTL)</td>
</tr>
<tr>
<td>SPIx_RDBR</td>
<td>SPI port receive data buffer</td>
<td>When register is read, hardware events are triggered</td>
</tr>
<tr>
<td>SPIx_BAUD</td>
<td>SPI port baud control</td>
<td>Value of 0 or 1 disables the serial clock</td>
</tr>
<tr>
<td>SPIx_SHADOW</td>
<td>SPI port data</td>
<td>Register has the same contents as SPIx_RDBR, but no action is taken when it is read</td>
</tr>
</tbody>
</table>
SPI Compatible Port Controllers

The clock polarity and the clock phase should be identical for the master device and the slave device involved in the communication link. The transfer format from the master may be changed between transfers to adjust to various requirements of a slave device.

When $CPHA = 0$, the slave select line outputs must be inactive (high) between each serial transfer. This is controlled automatically by the SPI hardware logic. When $CPHA = 1$, select line outputs $SPIx_SS$ may either remain active (low) between successive transfers or be inactive (high). This must be controlled by the software via manipulation of $SPIx_FLG$.

Figure 10-12 shows the SPI transfer protocol for $CPHA = 0$. Note $SCKx$ starts toggling in the middle of the data transfer, $SIZE = 0$, and $LSBF = 0$. 

Figure 10-11. SPI Modes of Operation

(MSB) first ($LSBF = 0$). Any combination of the $SIZE$ and $LSBF$ bits of $SPIx_CTL$ is allowed. For example, a 16-bit transfer with the least significant bit (LSB) first is another possible configuration.
Figure 10-13 shows the SPI transfer protocol for $\text{CPHA} = 1$. Note $\text{SCKx}$ starts toggling at the beginning of the data transfer, $\text{SIZE} = 0$, and $\text{LSBF} = 0$.

Figure 10-12. SPI Transfer Protocol for $\text{CPHA} = 0$

Figure 10-13. SPI Transfer Protocol for $\text{CPHA} = 1$
SPI General Operation

The SPI can be used in a single master as well as multimaster environment. The MOSIx, MISOx, and the SCKx signals are all tied together in both configurations. SPI transmission and reception are always enabled simultaneously, unless the broadcast mode has been selected. In broadcast mode, several slaves can be enabled to receive, but only one of the slaves must be in transmit mode driving the MISOx line. If the transmit or receive is not needed, it can simply be ignored. This section describes the clock signals, SPI operation as a master and as a slave, and error generation.

Precautions must be taken to avoid data corruption when changing the SPI module configuration. The configuration must not be changed during a data transfer. The clock polarity should only be changed when no slaves are selected. An exception to this is when an SPI communication link consists of a single master and a single slave, CPHA = 1, and the slave select input of the slave is always tied low. In this case, the slave is always selected and data corruption can be avoided by enabling the slave only after both the master and slave devices are configured.

In a multimaster or multislave SPI system, the data output pins (MOSIx and MISOx) can be configured to behave as open drain outputs, which prevents contention and possible damage to pin drivers. An external pull-up resistor is required on both the MOSIx and MISOx pins when this option is selected.

The WOM bit controls this option. When WOM is set and the SPI is configured as a master, the MOSIx pin is three-stated when the data driven out on MOSIx is a logic high. The MOSIx pin is not three-stated when the driven data is a logic low. Similarly, when WOM is set and the SPI is configured as a slave, the MISOx pin is three-stated if the data driven out on MISOx is a logic high.
Clock Signals

The SCKx signal is a gated clock that is only active during data transfers for the duration of the transferred word. The number of active edges is equal to the number of bits driven on the data lines. The clock rate can be as high as one-fourth of the SCLK rate. For master devices, the clock rate is determined by the 16-bit value of SPIx_BAUD. For slave devices, the value in SPIx_BAUD is ignored. When the SPI device is a master, SCKx is an output signal. When the SPI is a slave, SCKx is an input signal. Slave devices ignore the serial clock if the slave select input is driven inactive (high).

The SCKx signal is used to shift out and shift in the data driven onto the MISOx and MOSIx lines. The data is always shifted out on one edge of the clock and sampled on the opposite edge of the clock. Clock polarity and clock phase relative to data are programmable into SPIx_CTL and define the transfer format (Figure 10-11).

Master Mode Operation

When the SPI0 is configured as a master (and DMA mode is not selected), the interface operates in the following manner.

1. The core writes to SPI0_FLG, setting one or more of the SPI flag select bits (FLSx). This ensures that the desired slaves are properly deselected while the master is configured.

2. The core writes to the SPI0_BAUD and SPI0_CTL registers, enabling the device as a master and configuring the SPI system by specifying the appropriate word length, transfer format, baud rate, and other necessary information.

3. If CPHA = 1, the core activates the desired slaves by clearing one or more of the SPI flag bits (FLGx) of SPI0_FLG.
4. The \texttt{TIMOD} bits in SPI0\_CTL determine the SPI transfer initiate mode. The transfer on the SPI link begins upon either a data write by the core to the transmit data buffer (SPI0\_TDBR) or a data read of the receive data buffer (SPI0\_RDBR).

5. The SPI then generates the programmed clock pulses on SCK0 and simultaneously shifts data out of MOSI0 and shifts data in from MISO0. Before a shift, the shift register is loaded with the contents of the SPI0\_TDBR register. At the end of the transfer, the contents of the shift register are loaded into SPI0\_RDBR.

6. With each new transfer initiate command, the SPI continues to send and receive words, according to the SPI transfer initiate mode.

For SPI1 and SPI2, the same sequence above applies except the core must first verify that the GPIO pins to be used by the SPIx ports are not enabled as GPIOs. By default, the pins are dedicated for SPI use. For more information, see “General-Purpose Input/Output Ports C, D, E” on page 15-1. If the transmit buffer remains empty or the receive buffer remains full, the device operates according to the states of the 

\begin{itemize}
\item \texttt{SZ} = 1 and the transmit buffer is empty, the device repeatedly transmits 0s on the MOSIx pin. One word is transmitted for each new transfer initiate command.
\item \texttt{SZ} = 0 and the transmit buffer is empty, the device repeatedly transmits the last word it transmitted before the transmit buffer became empty.
\item \texttt{GM} = 1 and the receive buffer is full, the device continues to receive new data from the MISOx pin, overwriting the older data in the SPIx\_RDBR buffer.
\item \texttt{GM} = 0 and the receive buffer is full, the incoming data is discarded, and SPIx\_RDBR is not updated.
\end{itemize}

Transfer Initiation From Master (Transfer Modes)

When a device is enabled as a master, the initiation of a transfer is defined by the two \texttt{TIMOD} bits of SPIx\_CTL. Based on those two bits and the status of the interface, a new transfer is started upon either a read of SPIx\_RDBR or a write to SPIx\_TDBR. This is summarized in Table 10-4.
If the SPI port is enabled with $\text{TIMOD} = 01$ or $\text{TIMOD} = 11$, the hardware immediately issues a first interrupt or DMA request.

Table 10-4. Transfer Initiation

<table>
<thead>
<tr>
<th>TIMOD</th>
<th>Function</th>
<th>Transfer Initiated Upon</th>
<th>Action, Interrupt</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Transmit and</td>
<td>Initiate new single word transfer upon read of SPIx_RDBR and previous transfer</td>
<td>Interrupt active when receive buffer is full.</td>
</tr>
<tr>
<td></td>
<td>Receive</td>
<td>completed.</td>
<td>Read of SPIx_RDBR clears interrupt.</td>
</tr>
<tr>
<td>01</td>
<td>Transmit and</td>
<td>Initiate new single word transfer upon write to SPIx_TDBR and previous transfer</td>
<td>Interrupt active when transmit buffer is empty.</td>
</tr>
<tr>
<td></td>
<td>Receive</td>
<td>completed.</td>
<td>Writing to SPIx_TDBR clears interrupt.</td>
</tr>
<tr>
<td>10</td>
<td>Receive with</td>
<td>Initiate new multiword transfer upon enabling SPI for DMA mode. Individual word</td>
<td>Request DMA reads as long asSPI DMA FIFO is not empty.</td>
</tr>
<tr>
<td></td>
<td>DMA</td>
<td>transfers begin with a DMA read of SPIx_RDBR, and last transfer completed.</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Transmit with</td>
<td>Initiate new multiword transfer upon enabling SPI for DMA mode. Individual word</td>
<td>Request DMA writes as long asSPI DMA FIFO is not full.</td>
</tr>
<tr>
<td></td>
<td>DMA</td>
<td>transfers begin with a DMA write to SPIx_TDBR, and last transfer completed.</td>
<td></td>
</tr>
</tbody>
</table>

**Slave Mode Operation**

When a device is enabled as a slave (and DMA mode is not selected), the start of a transfer is triggered by a transition of the $\text{SPIxSS}$ select signal to the active state (low), or by the first active edge of the clock ($\text{SCKx}$), depending on the state of $\text{CPHA}$. 
These steps illustrate SPI operation in the slave mode:

1. The core writes to SPIx_CTL to define the mode of the serial link to be the same as the mode setup in the SPI master.

2. To prepare for the data transfer, the core writes data to be transmitted into SPIx_TDBR.

3. Once the SPIxSS falling edge is detected, the slave starts shifting data out on MISO and in from MOSI on SCKx edges, depending on the states of CPHA and CPOL.

4. Reception/transmission continues until SPIxSS is released or until the slave has received the proper number of clock cycles.

5. The slave device continues to receive/transmit with each new falling edge transition on SPIxSS and/or SCKx clock edge.

For SPI1 and SPI2, the SPI pins must not be enabled for GPIO. For more information, see “General-Purpose Input/Output Ports C, D, E” on page 15-1.

If the transmit buffer remains empty or the receive buffer remains full, the device operates according to the states of the SZ and GM bits in SPIx_CTL. If SZ = 1 and the transmit buffer is empty, the device repeatedly transmits 0s on the MISOx pin. If SZ = 0 and the transmit buffer is empty, it repeatedly transmits the last word it transmitted before the transmit buffer became empty. If GM = 1 and the receive buffer is full, the device continues to receive new data from the MOSIx pin, overwriting the older data in SPIx_RDBR. If GM = 0 and the receive buffer is full, the incoming data is discarded, and SPIx_RDBR is not updated.
Slave Ready for a Transfer

When a device is enabled as a slave, the actions shown in Table 10-5 are necessary to prepare the device for a new transfer.

Table 10-5. Transfer Preparation

<table>
<thead>
<tr>
<th>TIMOD</th>
<th>Function</th>
<th>Action, Interrupt</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Transmit and</td>
<td>Interrupt active when receive buffer is full.</td>
</tr>
<tr>
<td></td>
<td>Receive</td>
<td>Read of SPIx_RDBR clears interrupt.</td>
</tr>
<tr>
<td>01</td>
<td>Transmit and</td>
<td>Interrupt active when transmit buffer is empty.</td>
</tr>
<tr>
<td></td>
<td>Receive</td>
<td>Writing to SPIx_TDBR clears interrupt.</td>
</tr>
<tr>
<td>10</td>
<td>Receive with DMA</td>
<td>Request DMA reads as long as SPI DMA FIFO is not empty.</td>
</tr>
<tr>
<td>11</td>
<td>Transmit with DMA</td>
<td>Request DMA writes as long as SPI DMA FIFO is not full.</td>
</tr>
</tbody>
</table>

Error Signals and Flags

The status of a device is indicated by the SPIx_STAT register. See “SPI Status (SPIx_STAT) Register” on page 10-15 for more information.

Mode Fault Error (MODF)

The MODF bit is set in SPIx_STAT when the SPIxSS input pin of a device enabled as a master is driven low by some other device in the system. This occurs in multimaster systems when another device is also trying to be the master. To enable this feature, the PSSE bit in SPIx_CTL must be set. This contention between two drivers can potentially damage the driving pins. As soon as this error is detected, these actions occur:
SPI Compatible Port Controllers

- The **MSTR** control bit in **SPIx_CTL** is cleared, configuring the SPI interface as a slave.
- The **SPE** control bit in **SPIx_CTL** is cleared, disabling the SPI system.
- The **MODF** status bit in **SPIx_STAT** is set.
- An SPI Error interrupt is generated.

These four conditions persist until the **MODF** bit is cleared by software. Until the **MODF** bit is cleared, the SPI cannot be re-enabled, even as a slave. Hardware prevents the user from setting either **SPE** or **MSTR** while **MODF** is set.

When **MODF** is cleared, the interrupt is deactivated. Before attempting to re-enable the SPI as a master, the state of the **SPIxSS** input pin should be checked to make sure the pin is high. Otherwise, once **SPE** and **MSTR** are set, another mode fault error condition immediately occurs.

When **SPE** and **MSTR** are cleared, the SPI data and clock pin drivers (**MOSIX**, **MISOx**, and **SCKx**)) are disabled. However, the slave select output pins revert to being controlled by the GPIO port F registers. This could lead to contention on the slave select lines if these lines are still driven by the processor. To ensure that the slave select output drivers are disabled once an **MODF** error occurs, the program must configure the GPIO port F registers appropriately.

When enabling the **MODF** feature, the program must configure all of the **PFx** pins used as slave selects as inputs. Programs can do this by configuring the direction of the slave-select pins prior to configuring the SPI. This ensures that, once the **MODF** error occurs and the slave selects are automatically reconfigured as programmable pins, the slave select output drivers are disabled.
Transmission Error (TXE)

The TXE bit is set in SPIx_STAT when all the conditions of transmission are met, and there is no new data in SPIx_TDBR (SPIx_TDBR is empty). In this case, the contents of the transmission depend on the state of the SZ bit in SPIx_CTL. The TXE bit is sticky (W1C).

Reception Error (RBSY)

The RBSY flag is set in the SPIx_STAT register when a new transfer is completed, but before the previous data can be read from SPIx_RDBR. The state of the GM bit in the SPIx_CTL register determines whether SPIx_RDBR is updated with the newly received data. The RBSY bit is sticky (W1C).

Transmit Collision Error (TXCOL)

The TXCOL flag is set in SPIx_STAT when a write to SPIx_TDBR coincides with the load of the shift register. The write to SPIx_TDBR can be via software or the DMA. The TXCOL bit indicates that corrupt data may have been loaded into the shift register and transmitted. In this case, the data in SPIx_TDBR may not match what was transmitted. This error can easily be avoided by proper software control. The TXCOL bit is sticky (W1C).

Beginning and Ending an SPI Transfer

The start and finish of an SPI transfer depend on whether the device is configured as a master or a slave, whether the CPHA mode is selected, and whether the transfer initiation mode (TIMOD) is selected. For a master SPI with CPHA = 0, a transfer starts when either SPIx_TDBR is written to or SPIx_RDBR is read, depending on TIMOD. At the start of the transfer, the enabled slave select outputs are driven active (low). However, the SCKx signal remains inactive for the first half of the first cycle of SCKx. For a slave with CPHA = 0, the transfer starts as soon as the SPIxSS input goes low.
For \( CPHA = 1 \), a transfer starts with the first active edge of \( SCKx \) for both slave and master devices. For a master device, a transfer is considered finished after it sends the last data and simultaneously receives the last data bit. A transfer for a slave device ends after the last sampling edge of \( SCKx \).

The \( RXS \) bit defines when the receive buffer can be read. The \( TXS \) bit defines when the transmit buffer can be filled. The end of a single word transfer occurs when the \( RXS \) bit is set, indicating that a new word has just been received and latched into the receive buffer, \( SPIx_RDBR \). For a master SPI, \( RXS \) is set shortly after the last sampling edge of \( SCKx \). For a slave SPI, \( RXS \) is set shortly after the last \( SCKx \) edge, regardless of \( CPHA \) or \( CPOL \). The latency is typically a few \( SCLK \) cycles and is independent of \( TIMOD \) and the baud rate. If configured to generate an interrupt when \( SPIx_RDBR \) is full (\( TIMOD = 00 \)), the interrupt goes active one \( SCLK \) cycle after \( RXS \) is set. When not relying on this interrupt, the end of a transfer can be detected by polling the \( RXS \) bit.

To maintain software compatibility with other SPI devices, the \( SPIF \) bit is also available for polling. This bit may have a slightly different behavior from that of other commercially available devices. For a slave device, \( SPIF \) is cleared shortly after the start of a transfer (\( SPIxSS \) going low for \( CPHA = 0 \), first active edge of \( SCKx \) on \( CPHA = 1 \)), and is set at the same time as \( RXS \). For a master device, \( SPIF \) is cleared shortly after the start of a transfer (either by writing the \( SPIx_TDBR \) or reading the \( SPIx_RDBR \), depending on \( TIMOD \)), and is set one-half \( SCKx \) period after the last \( SCKx \) edge, regardless of \( CPHA \) or \( CPOL \).

The time at which \( SPIF \) is set depends on the baud rate. In general, \( SPIF \) is set after \( RXS \), but at the lowest baud rate settings (\( SPIx_BAUD < 4 \)). The \( SPIF \) bit is set before \( RXS \) is set, and consequently before new data is latched into \( SPIx_RDBR \), because of the latency. Therefore, for \( SPIx_BAUD = 2 \) or \( SPIx_BAUD = 3 \), \( RXS \) must be set before \( SPIF \) to read \( SPIx_RDBR \). For larger \( SPIx_BAUD \) settings, \( RXS \) is guaranteed to be set before \( SPIF \) is set.
If the SPI port is used to transmit and receive at the same time, or to switch between receive and transmit operation frequently, then the \texttt{TIMOD = 00} mode may be the best operation option. In this mode, software performs a dummy read from the \texttt{SPIx_RDBR} register to initiate the first transfer. If the first transfer is used for data transmission, software should write the value to be transmitted into the \texttt{SPIx_TDBR} register before performing the dummy read. If the transmitted value is arbitrary, it is good practice to set the \texttt{SZ} bit to ensure zero data is transmitted rather than random values. When receiving the last word of an SPI stream, software should ensure that the read from the \texttt{SPIx_RDBR} register does not initiate another transfer. It is recommended to disable the SPI port before the final \texttt{SPIx_RDBR} read access. Reading the \texttt{SPIx_SHADOW} register is not sufficient as it does not clear the interrupt request.

In master mode with the \texttt{CPHA} bit set, software should manually assert the required slave select signal before starting the transaction. After all data has been transferred, software typically releases the slave select again. If the SPI slave device requires the slave select line to be asserted for the complete transfer, this can be done in the SPI interrupt service routine only when operating in \texttt{TIMOD = 00} or \texttt{TIMOD = 10} mode. With \texttt{TIMOD = 01} or \texttt{TIMOD = 11}, the interrupt is requested while the transfer is still in progress.

**DMA**

The SPI ports also can use direct memory Access (DMA). For more information on DMA, see “DMA and Memory DMA MMRs” on page 9-3.

**DMA Functionality**

Each SPI has a single DMA engine which can be configured to support either an SPI transmit channel or a receive channel, but not both simultaneously. Therefore, when configured as a transmit channel, the received data is essentially ignored.
When configured as a receive channel, what is transmitted is irrelevant. A 16-bit by four-word FIFO (without burst capability) is included to improve throughput on the DMA access bus (DAB).

When using DMA for SPI transmit, the DMA_DONE interrupt signifies that the DMA FIFO is empty. However, at this point there may still be data in the SPI DMA FIFO waiting to be transmitted. Therefore, software needs to poll TXS in the SPIx_STAT register until it goes low for 2 successive reads, at which point the SPI DMA FIFO will be empty. When the SPIF bit subsequently gets set, the last word has been transferred.

The four-word FIFO is cleared when the SPI port is disabled.

**Master Mode DMA Operation**

When enabled as a master with the DMA engine configured to transmit or receive data, the SPI0 interface operates as follows.

1. The processor core writes to the appropriate DMA registers to enable the SPI DMA channel and to configure the necessary work units, access direction, word count, and so on. For more information, see “DMA and Memory DMA MMRs” on page 9-3.

2. The processor core writes to the SPI0_FLG register, setting one or more of the SPI flag select bits (FLSx).

3. The processor core writes to the SPI0_BAUD and SPI0_CTL registers, enabling the device as a master and configuring the SPI system by specifying the appropriate word length, transfer format, baud rate, and so on. The TIMOD field should be configured to select either “Receive with DMA” (TIMOD = 10) or “Transmit with DMA” (TIMOD = 11) mode.
4. If configured for receive, a receive transfer is initiated upon enabling of the SPI. Subsequent transfers are initiated as the SPI reads data from the SPI0_RDBR register and writes to the SPI DMA FIFO. The SPI then requests a DMA write to memory. Upon a DMA grant, the DMA engine reads a word from the SPI DMA FIFO and writes to memory.

If configured for transmit, the SPI requests a DMA read from memory. Upon a DMA grant, the DMA engine reads a word from memory and writes to the SPI DMA FIFO. As the SPI writes data from the SPI DMA FIFO into the SPI0_TDBR register, it initiates a transfer on the SPI link.

5. The SPI then generates the programmed clock pulses on SCK0 and simultaneously shifts data out of MOSI0 and shifts data in from MISO0. For receive transfers, the value in the shift register is loaded into the SPI0_RDBR register at the end of the transfer. For transmit transfers, the value in the SPI0_TDBR register is loaded into the shift register at the start of the transfer.

6. In receive mode, as long as there is data in the SPI DMA FIFO (the FIFO is not empty), the SPI continues to request a DMA write to memory. The DMA engine continues to read a word from the SPI DMA FIFO and writes to memory until the SPI DMA word count register transitions from 1 to 0. The SPI continues receiving words until SPI DMA mode is disabled.

For SPI1 and SPI2, the same sequence above applies except the core must first verify that the GPIO pins to be used by the SPIx ports are not enabled as GPIOs. By default, the pins are dedicated for SPI use. For more information, see “General-Purpose Input/Output Ports C, D, E” on page 15-1. In transmit mode, as long as there is room in the SPI DMA FIFO (the FIFO is not full), the SPI continues to request a DMA read from memory. The DMA engine continues to read a word from memory and write to the SPI
DMA FIFO until the SPI DMA word count register transitions from 1 to 0. The SPI continues transmitting words until the SPI DMA FIFO is empty.

For receive DMA operations, if the DMA engine is unable to keep up with the receive data stream, the receive buffer operates according to the state of the \( GM \) bit. If \( GM = 1 \) and the DMA FIFO is full, the device continues to receive new data from the \( MISOx \) pin, overwriting the older data in the \text{SPIx_RDBR} register. If \( GM = 0 \), and the DMA FIFO is full, the incoming data is discarded, and the \text{SPIx_RDBR} register is not updated. While performing receive DMA, the transmit buffer is assumed to be empty (and \text{TXE} is set). If \( SZ = 1 \), the device repeatedly transmits 0s on the \( MOSIx \) pin. If \( SZ = 0 \), it repeatedly transmits the contents of the \text{SPIx_TDBR} register. The \text{TXE} underrun condition cannot generate an error interrupt in this mode.

For transmit DMA operations, the master SPI initiates a word transfer only when there is data in the DMA FIFO. If the DMA FIFO is empty, the SPI waits for the DMA engine to write to the DMA FIFO before starting the transfer. All aspects of SPI receive operation should be ignored when configured in transmit DMA mode, including the data in the \text{SPIx_RDBR} register, and the status of the \text{RXS} and \text{RBSY} bits. The \text{RBSY} overrun conditions cannot generate an error interrupt in this mode. The \text{TXE} underrun condition cannot happen in this mode (master DMA \text{TX} mode), because the master SPI does not initiate a transfer if there is no data in the DMA FIFO.

Writes to the \text{SPIx_TDBR} register during an active SPI transmit DMA operation should not occur because the DMA data will be overwritten. Writes to the \text{SPIx_TDBR} register during an active SPI receive DMA operation are allowed. Reads from the \text{SPIx_RDBR} register are allowed at any time.

DMA requests are generated when the DMA FIFO is not empty (when \( \text{TIMOD} = 10 \)), or when the DMA FIFO is not full (when \( \text{TIMOD} = 11 \)).
Error interrupts are generated when there is an RBSY overflow error condition (when TIMOD = 10).

A master SPI DMA sequence may involve back-to-back transmission and/or reception of multiple DMA work units. The SPI controller supports such a sequence with minimal core interaction.

**Slave Mode DMA Operation**

When enabled as a slave with the DMA engine configured to transmit or receive data, the start of a transfer is triggered by a transition of the SPIxSS signal to the active-low state or by the first active edge of SCKx, depending on the state of CPHA.

The following steps illustrate the SPI receive or transmit DMA sequence in an SPI slave (in response to a master command).

1. The processor core writes to the appropriate DMA registers to enable the SPI DMA channel and configure the necessary work units, access direction, word count, and so on. For more information, see “DMA and Memory DMA MMRs” on page 9-3.

2. The processor core writes to the SPIx_CTL register to define the mode of the serial link to be the same as the mode setup in the SPI master. The TIMOD field is configured to select either receive with DMA (TIMOD = 10) or transmit with DMA (TIMOD = 11) mode.

3. If configured for receive, once the slave select input is active, the slave starts receiving and transmitting data on SCKx edges. The value in the shift register is loaded into the SPIx_RDBR register at the end of the transfer. As the SPI reads data from the SPIx_RDBR register and writes to the SPI DMA FIFO, it requests a DMA write to memory. Upon a DMA grant, the DMA engine reads a word from the SPI DMA FIFO and writes to memory.
For SPI1 and SPI2, the SPI pins must not be enabled for GPIO. For more information, see “General-Purpose Input/Output Ports C, D, E” on page 15-1.

If configured for transmit, the SPI requests a DMA read from memory. Upon a DMA grant, the DMA engine reads a word from memory and writes to the SPI DMA FIFO. The SPI then reads data from the SPI DMA FIFO and writes to the SPIx_TDBR register, awaiting the start of the next transfer. Once the slave select input is active, the slave starts receiving and transmitting data on SCKx edges. The value in the SPIx_TDBR register is loaded into the shift register at the start of the transfer.

4. In receive mode, as long as there is data in the SPI DMA FIFO (FIFO not empty), the SPI slave continues to request a DMA write to memory. The DMA engine continues to read a word from the SPI DMA FIFO and writes to memory until the SPI DMA word count register transitions from 1 to 0. The SPI slave continues receiving words on SCKx edges as long as the slave select input is active.

   In transmit mode, as long as there is room in the SPI DMA FIFO (FIFO not full), the SPI slave continues to request a DMA read from memory. The DMA engine continues to read a word from memory and write to the SPI DMA FIFO until the SPI DMA word count register transitions from 1 to 0. The SPI slave continues transmitting words on SCKx edges as long as the slave select input is active.

For receive DMA operations, if the DMA engine is unable to keep up with the receive data stream, the receive buffer operates according to the state of the GM bit. If GM = 1 and the DMA FIFO is full, the device continues to receive new data from the MOSIx pin, overwriting the older data in the SPIx_RDBR register. If GM = 0 and the DMA FIFO is full, the incoming data is discarded, and the SPIx_RDBR register is not updated. While
performing receive DMA, the transmit buffer is assumed to be empty and TXE is set. If $SZ = 1$, the device repeatedly transmits 0s on the MISOx pin. If $SZ = 0$, it repeatedly transmits the contents of the SPIx_TDBR register. The TXE underrun condition cannot generate an error interrupt in this mode.

For transmit DMA operations, if the DMA engine is unable to keep up with the transmit stream, the transmit port operates according to the state of the $SZ$ bit. If $SZ = 1$ and the DMA FIFO is empty, the device repeatedly transmits 0s on the MISOx pin. If $SZ = 0$ and the DMA FIFO is empty, it repeatedly transmits the last word it transmitted before the DMA buffer became empty. All aspects of SPI receive operation should be ignored when configured in transmit DMA mode, including the data in the SPIx_RDBR register, and the status of the RXS and RBSY bits. The RBSY overrun conditions cannot generate an error interrupt in this mode.

Writes to the SPIx_TDBR register during an active SPI transmit DMA operation should not occur because the DMA data will be overwritten. Writes to the SPIx_TDBR register during an active SPI receive DMA operation are allowed. Reads from the SPIx_RDBR register are allowed at any time.

DMA requests are generated when the DMA FIFO is not empty (when TIMOD = 10), or when the DMA FIFO is not full (when TIMOD = 11).

Error interrupts are generated when there is an RBSY overflow error condition (when TIMOD = 10), or when there is a TXE underflow error condition (when TIMOD = 11).
Timing

The enable lead time (T1), the enable lag time (T2), and the sequential transfer delay time (T3) each must always be greater than or equal to one-half the $SCK_x$ period. See Figure 10-14. The minimum time between successive word transfers (T4) is two $SCK_x$ periods. This is measured from the last active edge of $SCK_x$ of one word to the first active edge of $SCK_x$ of the next word. This is independent of the configuration of the SPI (CPHA, MSTR, and so on).

![Figure 10-14. SPI Timing](image)

For a master device with $CPHA = 0$, the slave select output is inactive (high) for at least one-half the $SCK_x$ period. In this case, T1 and T2 are each always be equal to one-half the $SCK_x$ period.
Timing
11 PARALLEL PERIPHERAL INTERFACE

The parallel peripheral interface (PPI) is a half-duplex, bidirectional port accommodating up to 16 bits of data. It has a dedicated clock pin, three multiplexed frame sync pins, and four dedicated data pins. Up to 12 additional data pins are available by reconfiguring the PF pins. The highest system throughput is achieved with 8-bit data, since two 8-bit data samples can be packed as a single 16-bit word. In such a case, the earlier sample is placed in the 8 least significant bits (LSBs).

The PPI_CLK pin can accept an external clock input up to SCLK/2. It cannot source a clock internally. Table 11-1 shows the pin interface for the PPI.

If a GPIO port F pin is configured for PPI use, its bit position in the GPIO port F MMRs are read back as 0.

Table 11-1. PPI Pins

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Function</th>
<th>Direction</th>
<th>Alternate Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPI15</td>
<td>Data</td>
<td>Bidirectional</td>
<td>PF4, SPI Enable Output</td>
</tr>
<tr>
<td>PPI14</td>
<td>Data</td>
<td>Bidirectional</td>
<td>PF5, SPI Enable Output</td>
</tr>
<tr>
<td>PPI13</td>
<td>Data</td>
<td>Bidirectional</td>
<td>PF6, SPI Enable Output</td>
</tr>
<tr>
<td>PPI12</td>
<td>Data</td>
<td>Bidirectional</td>
<td>PF7, SPI Enable Output</td>
</tr>
<tr>
<td>PPI11</td>
<td>Data</td>
<td>Bidirectional</td>
<td>PF8</td>
</tr>
<tr>
<td>PPI10</td>
<td>Data</td>
<td>Bidirectional</td>
<td>PF9</td>
</tr>
<tr>
<td>PPI9</td>
<td>Data</td>
<td>Bidirectional</td>
<td>PF10</td>
</tr>
<tr>
<td>PPI8</td>
<td>Data</td>
<td>Bidirectional</td>
<td>PF11</td>
</tr>
</tbody>
</table>
PPI Registers

Table 11-1. PPI Pins (Cont'd)

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Function</th>
<th>Direction</th>
<th>Alternate Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPI7</td>
<td>Data</td>
<td>Bidirectional</td>
<td>PF12</td>
</tr>
<tr>
<td>PPI6</td>
<td>Data</td>
<td>Bidirectional</td>
<td>PF13</td>
</tr>
<tr>
<td>PPI5</td>
<td>Data</td>
<td>Bidirectional</td>
<td>PF14</td>
</tr>
<tr>
<td>PPI4</td>
<td>Data</td>
<td>Bidirectional</td>
<td>PF15</td>
</tr>
<tr>
<td>PPI3</td>
<td>Data</td>
<td>Bidirectional</td>
<td>N/A</td>
</tr>
<tr>
<td>PPI2</td>
<td>Data</td>
<td>Bidirectional</td>
<td>N/A</td>
</tr>
<tr>
<td>PPI1</td>
<td>Data</td>
<td>Bidirectional</td>
<td>N/A</td>
</tr>
<tr>
<td>PPI0</td>
<td>Data</td>
<td>Bidirectional</td>
<td>N/A</td>
</tr>
<tr>
<td>PPI_FS3</td>
<td>Frame Sync3/Field</td>
<td>Bidirectional</td>
<td>PF3, SPI Enable Output</td>
</tr>
<tr>
<td>PPI_FS2</td>
<td>Frame Sync2/VSYNC</td>
<td>Bidirectional</td>
<td>Timer 2</td>
</tr>
<tr>
<td>PPI_FS1</td>
<td>Frame Sync1/HSYNC</td>
<td>Bidirectional</td>
<td>Timer 1</td>
</tr>
<tr>
<td>PPI_CLK</td>
<td>Up to SCLK/2</td>
<td>Input Clock</td>
<td>N/A</td>
</tr>
</tbody>
</table>

PPI Registers

The PPI has five memory-mapped registers (MMRs) that regulate its operation. These registers are the PPI control register (`PPI_CONTROL`), the PPI status register (`PPI_STATUS`), the delay count register (`PPI_DELAY`), the transfer count register (`PPI_COUNT`), and the lines per frame register (`PPI_FRAME`).

Descriptions and bit diagrams for each of these MMRs are provided in the following sections.


**PPI_CONTROL Register**

The PPI control register (PPI_CONTROL) configures the PPI for operating mode, control signal polarities, and data width of the port. See Figure 11-1 for a bit diagram of this MMR.

The POLC and POLS bits allow for selective signal inversion of the PPI_CLK and PPI_FSI/PPI_FS2 signals, respectively. This provides a mechanism to connect to data sources and receivers with a wide array of control signal polarities. Often, the remote data source/receiver also offers configurable signal polarities, so the POLC and POLS bits simply add increased flexibility.

The DLEN[2:0] field is programmed to specify the width of the PPI port in any mode. Note any width from 8 to 16 bits is supported, with the exception of a 9-bit port width. Any PF pins that are unused by the PPI as a result of the DLEN setting are free to be used in their normal PF capacity.

In ITU-R 656 modes, the DLEN field should not be configured for anything greater than a 10-bit port width. If it is, the PPI will reserve extra pins, making them unusable by other peripherals.

The SKIP_EN bit, when set, enables the selective skipping of data elements being read in through the PPI. By ignoring data elements, the PPI is able to conserve DMA bandwidth.

When the SKIP_EN bit is set, the SKIP_EO bit allows the PPI to ignore either the odd or the even elements in an input data stream. This is useful, for instance, when reading in a color video signal in YCbCr format (Cb, Y, Cr, Y, Cb, Y, Cr, Y...). Skipping every other element allows the PPI to only read in the luma (Y) or chroma (Cr or Cb) values. This could also be useful when synchronizing two processors to the same incoming video stream. One processor could handle luma processing and the other (whose SKIP_EO bit is set differently from the first processor’s) could handle chroma processing. This skipping feature is valid in ITU-R 656 modes and RX modes with external frame syncs.
PPI Registers

**PPI Control Register (PPI_CONTROL)**

<table>
<thead>
<tr>
<th>Bit 15</th>
<th>Bit 14</th>
<th>Bit 13</th>
<th>Bit 12</th>
<th>Bit 11</th>
<th>Bit 10</th>
<th>Bit 9</th>
<th>Bit 8</th>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Reset = 0x0000**

- **POLC**
  
  0 - PPI samples data on rising edge and drives data on falling edge of PPI_CLK
  
  1 - PPI samples data on falling edge and drives data on rising edge of PPI_CLK

- **DLEN[2:0] (Data Length)**
  
  000 - 8 bits
  001 - 10 bits
  010 - 11 bits
  011 - 12 bits
  100 - 13 bits
  101 - 14 bits
  110 - 15 bits
  111 - 16 bits

- **POLS**
  
  0 - PPI_FS1 and PPI_FS2 are treated as rising edge asserted
  
  1 - PPI_FS1 and PPI_FS2 are treated as falling edge asserted

- **PORT_DIR (Direction)**
  
  0 - PPI in Receive mode (input)
  
  1 - PPI in Transmit mode (output)

- **XFR_TYPE[1:0] (Transfer Type)**
  
  In Input mode:
  00 - ITU-R 656, Active Field Only
  01 - ITU-R 656, Entire Field
  10 - ITU-R 656, Vertical Blanking Only
  11 - Non-ITU-R 656 mode
  
  In Output mode:
  00 - Sync-less Output mode
  01, 10 - Sync-less Output mode
  11 - Output mode with 1, 2, or 3 frame syncs

- **PORT_CFG[1:0] (Port Configuration)**
  
  In non-ITU-R 656 Input modes (PORT_DIR = 0, XFR_TYPE = 11):
  00 - 1 external frame sync
  01 - 2 or 3 internal frame syncs
  10 - 2 or 3 external frame syncs
  11 - 0 frame syncs, triggered
  
  In Output modes with frame syncs (PORT_DIR = 1, XFR_TYPE = 11):
  00 - 1 frame sync
  01 - 2 or 3 frame syncs
  10 - Reserved
  11 - Sync PPI_FS3 to assertion of PPI_FS2 rather than of PPI_FS1.

- **POLC**
  
  0 - PPI samples data on rising edge and drives data on falling edge of PPI_CLK
  
  1 - PPI samples data on falling edge and drives data on rising edge of PPI_CLK

- **PORT_En (Enable)**
  
  0 - PPI disabled
  
  1 - PPI enabled

**Figure 11-1. PPI Control Register**
The **PACK_EN** bit only has meaning when the PPI port width (selected by DLEN[2:0]) is 8 bits. Every **PPI_CLK**-initiated event on the DMA bus (that is, an input or output operation) handles 16-bit entities. In other words, an input port width of 10 bits still results in a 16-bit input word for every **PPI_CLK**; the upper 6 bits are 0s. Likewise, a port width of 8 bits also results in a 16-bit input word, with the upper 8 bits all 0s. In the case of 8-bit data, it is usually more efficient to pack this information so that there are two bytes of data for every 16-bit word. This is the function of the **PACK_EN** bit. When set, it enables packing for all RX modes.

Consider this data transported into the PPI via DMA:

0xCE, 0xFA, 0xFE, 0xCA....

- **With PACK_EN set:**
  
  This is read into the PPI, configured for an 8-bit port width: 0xCE, 0xFA, 0xFE, 0xCA...

  This is transferred onto the DMA bus: 0xFACE, 0xCafe, ...

- **With PACK_EN cleared:**

  This is read into the PPI: 0xCE, 0xFA, 0xFE, 0xCA, ...

  This is transferred onto the DMA bus: 0x00CE, 0x00FA, 0x00FE, 0x00CA, ...

For TX modes, setting **PACK_EN** enables unpacking of bytes. Consider this data in memory, to be transported out through the PPI via DMA:

0xFACE CAFE.... (0xFA and 0xCA are the two most significant bits (MSBs) of their respective 16-bit words)

- **With PACK_EN set:**

  This is DMA’ed to the PPI: 0xFACE, 0xCafe, ...
This is transferred out through the PPI, configured for an 8-bit port width (note LSBs are transferred first): 0xCE, 0xFA, 0xFE, 0xCA, ...

- With PACK_EN cleared:

  This is DMA’ed to the PPI: 0xFACE, 0xCAFE, ...

  This is transferred out through the PPI, configured for an 8-bit port width: 0xCE, 0xFE, ...

The FLD_SEL bit is used primarily in the active field only ITU-R 656 mode. The FLD_SEL bit determines whether to transfer in only Field 1 of each video frame, or both Fields 1 and 2. Thus, it allows a savings in DMA bandwidth by transferring only every other field of active video.

The PORT_CFG[1:0] field is used to configure the operating mode of the PPI. It operates in conjunction with the PORT_DIR bit, which sets the direction of data transfer for the port. The XFR_TYPE[1:0] field is also used to configure operating mode and is discussed below. See Table 11-2 for the possible operating modes for the PPI.
### Table 11-2. PPI Possible Operating Modes

<table>
<thead>
<tr>
<th>PPI Mode</th>
<th># of Syncs</th>
<th>PORT_DIR</th>
<th>PORT_CFG</th>
<th>XFR_TYPE</th>
<th>POLC</th>
<th>POLS</th>
<th>FLD_SEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>RX mode, 0 frame syncs, external trigger</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>11</td>
<td>0 or 1</td>
<td>0 or 1</td>
<td>0</td>
</tr>
<tr>
<td>RX mode, 0 frame syncs, internal trigger</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>11</td>
<td>0 or 1</td>
<td>0 or 1</td>
<td>1</td>
</tr>
<tr>
<td>RX mode, 1 external frame sync</td>
<td>1</td>
<td>0</td>
<td>00</td>
<td>11</td>
<td>0 or 1</td>
<td>0 or 1</td>
<td>X</td>
</tr>
<tr>
<td>RX mode, 2 or 3 external frame syncs</td>
<td>3</td>
<td>0</td>
<td>10</td>
<td>11</td>
<td>0 or 1</td>
<td>0 or 1</td>
<td>X</td>
</tr>
<tr>
<td>RX mode, 2 or 3 internal frame syncs</td>
<td>3</td>
<td>0</td>
<td>01</td>
<td>11</td>
<td>0 or 1</td>
<td>0 or 1</td>
<td>X</td>
</tr>
<tr>
<td>RX mode, ITU-R 656, Active Field Only</td>
<td>embedded</td>
<td>0</td>
<td>XX</td>
<td>00</td>
<td>0 or 1</td>
<td>0</td>
<td>0 or 1</td>
</tr>
<tr>
<td>RX mode, ITU-R 656, Vertical Blanking Only</td>
<td>embedded</td>
<td>0</td>
<td>XX</td>
<td>10</td>
<td>0 or 1</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>RX mode, ITU-R 656, Entire Field</td>
<td>embedded</td>
<td>0</td>
<td>XX</td>
<td>01</td>
<td>0 or 1</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>TX mode, 0 frame syncs</td>
<td>0</td>
<td>1</td>
<td>XX</td>
<td>00, 01, 10</td>
<td>0 or 1</td>
<td>0 or 1</td>
<td>X</td>
</tr>
<tr>
<td>TX mode, 1 internal or external frame sync</td>
<td>1</td>
<td>1</td>
<td>00</td>
<td>11</td>
<td>0 or 1</td>
<td>0 or 1</td>
<td>X</td>
</tr>
<tr>
<td>TX mode, 2 external frame syncs</td>
<td>2</td>
<td>1</td>
<td>01</td>
<td>11</td>
<td>0 or 1</td>
<td>0 or 1</td>
<td>X</td>
</tr>
<tr>
<td>TX mode, 2 or 3 internal frame syncs, FS3 sync'ed to FS1 assertion</td>
<td>3</td>
<td>1</td>
<td>01</td>
<td>11</td>
<td>0 or 1</td>
<td>0 or 1</td>
<td>X</td>
</tr>
<tr>
<td>TX mode, 2 or 3 internal frame syncs, FS3 sync'ed to FS2 assertion</td>
<td>3</td>
<td>1</td>
<td>11</td>
<td>11</td>
<td>0 or 1</td>
<td>0 or 1</td>
<td>X</td>
</tr>
</tbody>
</table>
The `XFR_TYPE[1:0]` field configures the PPI for various modes of operation. Refer to Table 11-2 to see how `XFR_TYPE[1:0]` interacts with other bits in `PPI_CONTROL` to determine the PPI operating mode.

The `PORT_EN` bit, when set, enables the PPI for operation.

Note that, when configured as an input port, the PPI does not start data transfer after being enabled until the appropriate synchronization signals are received. If configured as an output port, transfer (including the appropriate synchronization signals) begins as soon as the frame syncs (timer units) are enabled, so all frame syncs must be configured before this happens. Refer to the section “Frame Synchronization in GP Modes” on page 11-28 for more information.

**PPI_STATUS Register**

The PPI status register (`PPI_STATUS`) contains bits that provide information about the current operating state of the PPI.

The entire register is cleared when read, so the status word must be parsed to evaluate which bits have been set.

The `ERR_DET` bit is a sticky bit that denotes whether or not an error was detected in the ITU-R 656 control word preamble. The bit is valid only in ITU-R 656 modes. If `ERR_DET = 1`, an error was detected in the preamble. If `ERR_DET = 0`, no error was detected in the preamble.

The `ERR_NCOR` bit is sticky and is relevant only in ITU-R 656 modes. If `ERR_NCOR = 0 and ERR_DET = 1`, all preamble errors that have occurred have been corrected. If `ERR_NCOR = 1`, an error in the preamble was detected but not corrected. This situation generates a PPI error interrupt, unless this condition is masked off in the `SIC_IMASKx` register.
The **FT_ERR** bit is sticky and indicates, when set, that a frame track error has occurred. It is valid for RX modes only. In this condition, the programmed number of lines per frame in **PPI_FRAME** does not match up with the “frame start detect” condition (see the information note on page 11-12). A frame track error generates a PPI error interrupt, unless this condition is masked off in the **SIC_IMASKx** register.

The **FLD** bit is set or cleared at the same time as the change in state of **F** (in ITU-R 656 modes) or **PPI_FS3** (in other RX modes). It is valid for input modes only. The state of **FLD** reflects the current state of the **F** or **PPI_FS3** signals. In other words, the **FLD** bit always reflects the current video field being processed by the PPI.

The **OVR** bit is sticky and indicates, when set, that the PPI FIFO has overflowed and can accept no more data. A FIFO overflow error generates a PPI error interrupt, unless this condition is masked off in the **SIC_IMASKx** register.

The **PPI FIFO** is 16 bits wide and has 16 entries.

The **UNDR** bit is sticky and indicates, when set, that the PPI FIFO has underrun and is data-starved. A FIFO underrun error generates a PPI error interrupt, unless this condition is masked off in the **SIC_IMASKx** register.

**PPI_DELAY Register**

The delay count register (**PPI_DELAY**) can be used in all configurations except ITU-R 656 modes and GP modes with 0 frame syncs. It contains a count of how many **PPI_CLK** cycles to delay after assertion of **PPI_FS1** before starting to read in or write out data.
Note in TX modes using at least one frame sync, there is a one-cycle delay beyond what is specified in the PPI_DELAY register.
**PPI_COUNT Register**

The transfer count register (PPI_COUNT) is used only in cases where recurring hardware frame syncs (either externally or internally generated) are involved. It is not needed in ITU-R 656 modes or modes with 0 frame syncs. For RX modes, this register holds the number of samples to read into the PPI per line, minus one. For TX modes, it holds the number of samples to write out through the PPI per line, minus one. The register itself does not actually decrement with each transfer. Thus, at the beginning of a new line of data, there is no need to rewrite the value of this register. For example, to receive or transmit 100 samples through the PPI, set PPI_COUNT to 99.

⚠️ Take care to ensure that the number of samples programmed into PPI_COUNT is in keeping with the number of samples expected during the “horizontal” interval specified by PPI_FS1.

**Transfer Count Register (PPI_COUNT)**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Reset = 0x0000

PPI_COUNT[15:0]

- In RX modes, holds one less than the number of samples to read in to the PPI per line.
- In TX modes, holds one less than the number of samples to write out through the PPI per line.

Figure 11-4. Transfer Count Register
PPI Registers

PPI_FRAME Register

The lines per frame (PPI_FRAME) register is used in all TX and RX modes with external frame syncs. For ITU-R 656 modes, this register holds the number of lines expected per frame of data, where a frame is defined as Field 1 and Field 2 combined, designated by the F indicator in the ITU-R stream. Here, a line is defined as a complete ITU-R 656 SAV-EAV cycle.

For non-ITU-R 656 modes with external frame syncs, a frame is defined as the data bounded between PPI_FS2 assertions, regardless of the state of PPI_FS3. A line is defined as a complete PPI_FS1 cycle. In these modes, PPI_FS3 is used only to determine the original “frame start” each time the PPI is enabled. It is ignored on every subsequent field and frame, and its state (high or low) is not important except during the original frame start.

If the start of a new frame (or field, for ITU-R 656 mode) is detected before the number of lines specified by PPI_FRAME have been transferred, a frame tracker error results, and the FT_ERR bit in PPI_STATUS is set. However, the PPI still automatically re-initializes to count to the value programmed in PPI_FRAME, and data transfer continues.

- In ITU-R 656 modes, a frame start detect happens on the falling edge of F, the field indicator. This occurs at the start of Field 1.
- In RX mode with 3 external frame syncs, a frame start detect refers to a condition where a PPI_FS2 assertion is followed by an assertion of PPI_FS1 while PPI_FS3 is low. This occurs at the start of Field 1.

Note that PPI_FS3 only needs to be low when PPI_FS1 is asserted, not when PPI_FS2 asserts. Also, PPI_FS3 is only used to synchronize to the start of the very first frame after the PPI is enabled. It is subsequently ignored.

- When using RX mode with 3 external frame syncs, and only 2 syncs are needed, configure the PPI for three-frame-sync operation and provide an external pull-down to GND for the PPI_FS3 pin.
Parallel Peripheral Interface

ITU-R 656 Modes

The PPI supports three input modes for ITU-R 656-framed data. These modes are described in this section. Although the PPI does not explicitly support an ITU-R 656 output mode, recommendations for using the PPI for this situation are provided as well.

ITU-R 656 Background

According to the ITU-R 656 recommendation (formerly known as CCIR-656), a digital video stream has the characteristics shown in Figure 11-6 and Figure 11-7 for 525/60 (NTSC) and 625/50 (PAL) systems. The processor supports only the bit-parallel mode of ITU-R 656. Both 8- and 10-bit video element widths are supported.

In this mode, the horizontal (H), vertical (V), and field (F) signals are sent as an embedded part of the video data stream in a series of bytes that form a control word. The start of active video (SAV) and end of active video (EAV) signals indicate the beginning and end of data elements to read in on each line. SAV occurs on a 1-to-0 transition of H, and EAV begins on a 0-to-1 transition of H. An entire field of video is comprised of active video + horizontal blanking (the space between an EAV and SAV code) and vertical blanking (the space where \( V = 1 \)). A field of video commences on a transition of the F bit. The “odd field” is denoted by a value of \( F = 0 \),
whereas $F = 1$ denotes an even field. Progressive video makes no distinction between Field 1 and Field 2, whereas interlaced video requires each field to be handled uniquely, because alternate rows of each field combine to create the actual video image.

![ITU-R 656 8-Bit Parallel Data Stream for NTSC (PAL) Systems](image)

**Figure 11-6.** ITU-R 656 8-Bit Parallel Data Stream for NTSC (PAL) Systems
The SAV and EAV codes are shown in more detail in Table 11-3. Note there is a defined preamble of three bytes (0xFF, 0x00, 0x00), followed by the XY register word, which, aside from the \( F \) (field), \( V \) (vertical blanking) and \( H \) (horizontal blanking) bits, contains four protection bits for single-bit error detection and correction. Note \( F \) and \( V \) are only allowed to change as part of EAV sequences (that is, transition from \( H = 0 \) to \( H = 1 \)). The bit definitions are as follows:
ITU-R 656 Modes

- $F = 0$ for Field 1
- $F = 1$ for Field 2
- $V = 1$ during vertical blanking
- $V = 0$ when not in vertical blanking
- $H = 0$ at SAV
- $H = 1$ at EAV
- $P_3 = V \oplus H$
- $P_2 = F \oplus H$
- $P_1 = F \oplus V$
- $P_0 = F \oplus V \oplus H$

In many applications, video streams other than the standard NTSC/PAL formats (for example, CIF, QCIF) can be employed. Because of this, the processor interface is flexible enough to accommodate different row and field lengths. In general, as long as the incoming video has the proper EAV/SAV codes, the PPI can read it in. In other words, a CIF image could be formatted to be “656-compliant,” where EAV and SAV values define the range of the image for each line, and the $V$ and $F$ codes can be used to delimit fields and frames.
Table 11-3. Control Byte Sequences for 8-Bit and 10-Bit ITU-R 656 Video

<table>
<thead>
<tr>
<th>8-Bit Data</th>
<th>10-Bit Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>D9 (MSB)</td>
<td>D8 D7 D6 D5 D4 D3 D2 D1 D0</td>
</tr>
<tr>
<td>Preamble</td>
<td>1 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td></td>
<td>0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>control Byte</td>
<td>1 F V H P3 P2 P1 P0 0 0</td>
</tr>
</tbody>
</table>

**ITU-R 656 Input Modes**

Figure 11-8 shows a general illustration of data movement in the ITU-R 656 input modes. In the figure, the clock \( \text{CLK} \) is either provided by the video source or supplied externally by the system.

![ITU-R 656 Input Modes Diagram](image)

Figure 11-8. ITU-R 656 Input Modes

There are three sub-modes supported for ITU-R 656 inputs: entire field, active video only, and vertical blanking interval only. Figure 11-9 shows these three sub-modes.
In this mode, the entire incoming bit stream is read in through the PPI. This includes active video as well as control byte sequences and ancillary data that may be embedded in horizontal and vertical blanking intervals. Data transfer starts immediately after synchronization to Field 1 occurs, but does not include the first EAV code that contains the $F = 0$ assignment.

Note the first line transferred in after enabling the PPI will be missing its first 4-byte preamble. However, subsequent lines and frames should have all control codes intact.

One side benefit of this mode is that it enables a “loopback” feature through which a frame or two of data can be read in through the PPI and subsequently output to a compatible video display device. Of course, this requires multiplexing on the PPI pins, but it enables a convenient way to verify that 656 data can be read into and written out from the PPI.
Active Video Only

This mode is used when only the active video portion of a field is of interest, and not any of the blanking intervals. The PPI ignores (does not read in) all data between EAV and SAV, as well as all data present when \( V = 1 \). In this mode, the control byte sequences are not stored to memory; they are filtered out by the PPI. After synchronizing to the start of Field 1, the PPI ignores incoming samples until it sees an SAV.

In this mode, the user specifies the number of total (active plus vertical blanking) lines per frame in the `PPI_FRAME` MMR.

Vertical Blanking Interval (VBI) Only

In this mode, data transfer is only active while \( V = 1 \) is in the control byte sequence. This indicates that the video source is in the midst of the vertical blanking interval (VBI), which is sometimes used for ancillary data transmission. The ITU-R 656 recommendation specifies the format for these ancillary data packets, but the PPI is not equipped to decode the packets themselves. This task must be handled in software. Horizontal blanking data is logged where it coincides with the rows of the VBI. Control byte sequence information is always logged. The user specifies the number of total lines (active plus vertical blanking) per frame in the `PPI_FRAME` MMR.

Note the VBI is split into two regions within each field. From the PPI’s standpoint, it considers these two separate regions as one contiguous space. However, keep in mind that frame synchronization begins at the start of Field 1, which doesn’t necessarily correspond to the start of vertical blanking. For instance, in 525/60 systems, the start of Field 1 (\( F = 0 \)) corresponds to Line 4 of the VBI.
ITU-R 656 Modes

ITU-R 656 Output Mode

The PPI does not explicitly provide functionality for framing an ITU-R 656 output stream with proper preambles and blanking intervals. However, with the TX mode with 0 frame syncs, this process can be supported manually. Essentially, this mode provides a streaming operation from memory out through the PPI. Data and control codes can be set up in memory prior to sending out the video stream. With the 2D DMA engine, this could be performed in a number of ways. For instance, one line of blanking (H + V) could be stored in a buffer and sent out N times by the DMA controller when appropriate, before proceeding to DMA active video. Alternatively, one entire field (with control codes and blanking) can be set up statically in a buffer while the DMA engine transfers only the active video region into the buffer, on a frame-by-frame basis.

Frame Synchronization in ITU-R 656 Modes

Synchronization in ITU-R 656 modes always occurs at the falling edge of F, the field indicator. This corresponds to the start of Field 1. Consequently, up to two fields might be ignored (for example, if Field 1 just started before the PPI-to-camera channel was established) before data is received into the PPI.

Because all H and V signalling is embedded in the data stream in ITU-R 656 modes, the PPI_COUNT register is not necessary. However, the PPI_FRAME register is used in order to check for synchronization errors. The user programs this MMR for the number of lines expected in each frame of video, and the PPI keeps track of the number of EAV-to-SAV transitions that occur from the start of a frame until it decodes the end-of-frame condition (transition from F = 1 to F = 0). At this time, the actual number of lines processed is compared against the value in PPI_FRAME. If there is a mismatch, the FT_ERR bit in the PPI_STATUS register is asserted. For instance, if an SAV transition is missed, the current field will only have NUM_ROWS - 1 rows, but resynchronization will reoccur at the start of the next frame.
Upon completing reception of an entire field, the field register bit is toggled in the `PPI_STATUS` register. This way, an interrupt service routine (ISR) can discern which field was just read in.

### General-Purpose PPI Modes

The general-purpose (GP) PPI modes are intended to suit a wide variety of data capture and transmission applications. Table 11-4 summarizes these modes. If a particular mode shows a given `PPI_FSx` frame sync not being used, this implies that the pin is available for its alternate, multiplexed processor function (that is, as a timer or flag pin). The exception to this is that when the PPI is configured for a 2-frame-sync mode, `PPI_FS3` cannot be used as a general-purpose flag, even though it is not used by the PPI.

**Table 11-4. General-Purpose PPI Modes**

<table>
<thead>
<tr>
<th>GP PPI Mode</th>
<th>PPI_FS1 Direction</th>
<th>PPI_FS2 Direction</th>
<th>PPI_FS3 Direction</th>
<th>Data Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>RX mode, 0 frame syncs, external trigger</td>
<td>Input</td>
<td>Not used</td>
<td>Not used</td>
<td>Input</td>
</tr>
<tr>
<td>RX mode, 0 frame syncs, internal trigger</td>
<td>Not used</td>
<td>Not used</td>
<td>Not used</td>
<td>Input</td>
</tr>
<tr>
<td>RX mode, 1 external frame sync</td>
<td>Input</td>
<td>Not used</td>
<td>Not used</td>
<td>Input</td>
</tr>
<tr>
<td>RX mode, 2 or 3 external frame syncs</td>
<td>Input</td>
<td>Input</td>
<td>Input</td>
<td>Input</td>
</tr>
<tr>
<td>RX mode, 2 or 3 internal frame syncs</td>
<td>Output</td>
<td>Output</td>
<td>Output</td>
<td>Input</td>
</tr>
<tr>
<td>TX mode, 0 frame syncs</td>
<td>Not used</td>
<td>Not used</td>
<td>Not used</td>
<td>Output</td>
</tr>
<tr>
<td>TX mode, 1 external frame sync</td>
<td>Input</td>
<td>Not used</td>
<td>Not used</td>
<td>Output</td>
</tr>
<tr>
<td>TX mode, 2 external frame syncs</td>
<td>Input</td>
<td>Input</td>
<td>Output</td>
<td>Output</td>
</tr>
<tr>
<td>TX mode, 1 internal frame sync</td>
<td>Output</td>
<td>Not used</td>
<td>Not used</td>
<td>Output</td>
</tr>
<tr>
<td>TX mode, 2 or 3 internal frame syncs</td>
<td>Output</td>
<td>Output</td>
<td>Output</td>
<td>Output</td>
</tr>
</tbody>
</table>
Figure 11-10 illustrates the general flow of the GP modes. The top of the diagram shows an example of RX mode with 1 external frame sync. After the PPI receives the hardware frame sync pulse (PPI_FS1), it delays for the duration of the PPI_CLK cycles programmed into PPI_DELAY. The DMA controller then transfers in the number of samples specified by PPI_COUNT. Every sample that arrives after this, but before the next PPI_FS1 frame sync arrives, is ignored and not transferred onto the DMA bus.

If the next PPI_FS1 frame sync arrives before the specified PPI_COUNT samples have been read in, the sample counter re-initializes to 0 and starts to count up to PPI_COUNT again. This situation can cause the DMA channel configuration to lose synchronization with the PPI transfer process.

The bottom of Figure 11-10 shows an example of TX mode, 1 internal frame sync. After PPI_FS1 is asserted, there is a latency of 1 PPI_CLK cycle, and then there is a delay for the number of PPI_CLK cycles programmed into PPI_DELAY. Next, the DMA controller transfers out the number of samples specified by PPI_COUNT. No further DMA takes place until the next PPI_FS1 sync and programmed delay occur.

If the next PPI_FS1 frame sync arrives before the specified PPI_COUNT samples have been transferred out, the sync has priority and starts a new line transfer sequence. This situation can cause the DMA channel configuration to lose synchronization with the PPI transfer process.

Data Input (RX) Modes

The PPI supports several modes for data input. These modes differ chiefly by the way the data is framed. Refer to Table 11-2 for information on how to configure the PPI for each mode.
No Frame Syncs

These modes cover the set of applications where periodic frame syncs are not generated to frame the incoming data. There are two options for starting the data transfer, both configured by the PPI_CONTROL register.

- External trigger: An external source sends a single frame sync (tied to PPI_FS1) at the start of the transaction, when FLD_SEL = 0 and PORT_CFG = b#11.

- Internal trigger: Software initiates the process by setting PORT_EN = 1 with FLD_SEL = 1 and PORT_CFG = b#11.
General-Purpose PPI Modes

All subsequent data manipulation is handled via DMA. For example, an arrangement could be set up between alternating 1K memory buffers. When one fills up, DMA continues with the second buffer, at the same time that another DMA operation is clearing the first memory buffer for reuse.

Due to clock domain synchronization in RX modes with no frame syncs, there may be a delay of at least $2 \text{ PPI_CLK}$ cycles between when the mode is enabled and when valid data is received. Therefore, detection of the start of valid data should be managed by software.

1, 2, or 3 External Frame Syncs

The 1-sync mode is intended for analog-to-digital converter (ADC) applications. The top part of Figure 11-11 shows a typical illustration of the system setup for this mode.

Figure 11-11. RX Mode, External Frame Syncs
The 3-sync mode shown at the bottom of Figure 11-11 supports video applications that use hardware signalling (HSYNC, VSYNC, FIELD) in accordance with the ITU-R 601 recommendation. The mapping for the frame syncs in this mode is PPI_FS1 = HSYNC, PPI_FS2 = VSYNC, PPI_FS3 = FIELD. Refer to “Frame Synchronization in GP Modes” on page 11-28 for more information about frame syncs in this mode.

A 2-sync mode is implicitly supported by pulling PPI_FS3 to GND by an external resistor when configured in 3-sync mode.

2 or 3 Internal Frame Syncs

This mode can be useful for interfacing to video sources that can be slaved to a master processor. In other words, the processor controls when to read from the video source by asserting PPI_FS1 and PPI_FS2, and then reading data into the PPI. The PPI_FS3 frame sync provides an indication of which field is currently being transferred, but since it is an output, it can simply be left floating if not used. Figure 11-12 shows a sample application for this mode.

![Figure 11-12. RX Mode, Internal Frame Syncs](image)

Data Output (TX) Modes

The PPI supports several modes for data output. These modes differ chiefly by the way the data is framed. Refer to Table 11-2 for information on how to configure the PPI for each mode.
General-Purpose PPI Modes

No Frame Syncs

In this mode, data blocks specified by the DMA controller are sent out through the PPI with no framing. That is, once the DMA channel is configured and enabled, and the PPI is configured and enabled, data transfers will take place immediately, synchronized to PPI_CLK. See Figure 11-13 for an illustration of this mode.

In this mode, there is a delay of up to $16 \text{ SCLK}$ cycles (for > 8-bit data) or $32 \text{ SCLK}$ cycles (for 8-bit data) between enabling the PPI and transmission of valid data. Furthermore, DMA must be configured to transmit at least 16 samples (for > 8-bit data) or 32 samples (for 8-bit data).

![Figure 11-13. TX Mode, 0 Frame Syncs](image)

1 or 2 External Frame Syncs

In these modes, an external receiver can frame data sent from the PPI. Both 1-sync and 2-sync modes are supported. The top diagram in Figure 11-14 shows the 1-sync case, while the bottom diagram illustrates the 2-sync mode.

There is a mandatory delay of $1.5 \text{ PPI_CLK}$ cycles, plus the value programmed in $\text{PPIDelay}$, between assertion of the external frame sync(s) and the transfer of valid data out through the PPI.
1, 2, or 3 Internal Frame Syncs

The 1-sync mode is intended for interfacing to digital-to-analog converters (DACs) with a single frame sync. The top part of Figure 11-15 shows an example of this type of connection.

The 3-sync mode is useful for connecting to video and graphics displays, as shown in the bottom part of Figure 11-15. A 2-sync mode is implicitly supported by leaving PPI_FS3 unconnected in this case.
Frame Synchronization in GP Modes

Frame synchronization in GP modes operates differently in modes with internal frame syncs than in modes with external frame syncs.

Modes with Internal Frame Syncs

In modes with internal frame syncs, PPI_FS1 and PPI_FS2 link directly to the pulse-width modulation (PWM) circuits of timer 1 and timer 2, respectively. This allows for arbitrary pulse widths and periods to be programmed for these signals using the existing TIMERx registers. This capability accommodates a wide range of timing needs. Note these PWM circuits are clocked by PPI_CLK, not by SCLK or PF1 (as during conventional timer PWM operation). If PPI_FS2 is not used in the configured
PPI mode, Timer 2 operates as it normally would, unrestricted in functionality. The state of PPI_FS3 depends completely on the state of PPI_FS1 and/or PPI_FS2, so PPI_FS3 has no inherent programmability.

To program PPI_FS1 and/or PPI_FS2 for operation in an internal frame sync mode:


2. Configure the width and period for each frame sync signal via TIMER1_WIDTH and TIMER1_PERIOD (for PPI_FS1), or TIMER2_WIDTH and TIMER2_PERIOD (for PPI_FS2).

3. Set up TIMER1_CONFIG for PWM_OUT mode (for PPI_FS1). If used, configure TIMER2_CONFIG for PWM_OUT mode (for PPI_FS2). This includes setting CLK_SEL = 1 and TIN_SEL = 1 for each timer.

4. Write to PPI_CONTROL to configure and enable the PPI.

5. Write to TIMER_ENABLE to enable Timer 1 and/or Timer 2.

It is important to guarantee proper frame sync polarity between the PPI and Timer peripherals. To do this, make sure that if PPI_CONTROL[15:14] = b#10 or b#11, the PULSE_HI bit is cleared in TIMER1_CONFIG and TIMER2_CONFIG. Likewise, if PPI_CONTROL[15:14] = b#00 or b#01, the PULSE_HI bit should be set in TIMER1_CONFIG and TIMER2_CONFIG.

To switch to another PPI mode not involving internal frame syncs:

1. Disable the PPI (using PPI_CONTROL).

2. Disable the timers (using TIMER_DISABLE).
General-Purpose PPI Modes

Modes With External Frame Syncs

In RX modes with external frame syncs, the PPI_FS1 and PPI_FS2 pins become edge-sensitive inputs. In such a mode, Timers 1 and 2 can be used for a purpose not involving the TMR1 and TMR2 pins. However, timer access to a TMRx pin is disabled when the PPI is using that pin for a PPI_FSx frame sync input function. For modes that do not require PPI_FS2, Timer 2 is not restricted in functionality and can be operated as if the PPI were not being used (that is, the TMR2 pin becomes available for timer use as well). For more information on configuring and using the timers, refer to Chapter 16, “Timers”.

In RX Mode with 3 external frame syncs, the start of frame detection occurs where a PPI_FS2 assertion is followed by an assertion of PPI_FS1 while PPI_FS3 is low. This happens at the start of Field 1.

Note that PPI_FS3 only needs to be low when PPI_FS1 is asserted, not when PPI_FS2 asserts. Also, PPI_FS3 is only used to synchronize to the start of the very first frame after the PPI is enabled. It is subsequently ignored.

In TX modes with external frame syncs, the PPI_FS1 and PPI_FS2 pins are treated as edge-sensitive inputs. In this mode, it is not necessary to configure the timer(s) associated with the frame sync(s) as input(s), or to enable them via the TIMER_ENABLE register. Additionally, the actual timers themselves are available for use, even though the timer pin(s) are taken over by the PPI. In this case, there is no requirement that the time base (configured by TIN_SEL in TIMERx_CONFIG) be PPI_CLK.

However, if using a timer whose pin is connected to an external frame sync, be sure to disable the pin via the OUT_DIS bit in TIMERx_CONFIG. Then the timer itself can be configured and enabled for non-PPI use without affecting PPI operation in this mode. For more information, see Chapter 16, “Timers”.
DMA Operation

The PPI must be used with the processor’s DMA engine. This section discusses how the two interact. For additional information about the DMA engine, including explanations of DMA registers and DMA operations, refer to Chapter 9, “Direct Memory Access”.

The PPI DMA channel can be configured for either transmit or receive operation, and it has a maximum throughput of \((\text{PPI}_{-}\text{CLK}) \times (16 \text{ bits/transfer})\). In modes where data lengths are greater than 8 bits, only one element can be clocked in per \(\text{PPI}_{-}\text{CLK}\) cycle, and this results in reduced bandwidth (since no packing is possible). The highest throughput is achieved with 8-bit data and \(\text{PACK\_EN} = 1\) (packing mode enabled). Note for 16-bit packing mode, there must be an even number of data elements.

Configuring the PPI’s DMA channel is a necessary step toward using the PPI interface. It is the DMA engine that generates interrupts upon completion of a row, frame, or partial-frame transfer. It is also the DMA engine that coordinates the origination or destination point for the data that is transferred through the PPI.

The processor’s 2D DMA capability allows the processor to be interrupted at the end of a line or after a frame of video has been transferred, as well as if a DMA error occurs. In fact, the specification of the \(\text{DMA}_{\_}\text{XCOUNT}\) and \(\text{DMA}_{\_}\text{YCOUNT}\) MMRs allows for flexible data interrupt points. For example, assume the DMA registers \(\text{XMODIFY} = \text{YMODIFY} = 1\). Then, if a data frame contains 320 x 240 bytes (240 rows of 320 bytes each), these conditions hold:

- Setting \(\text{XCOUNT} = 320\), \(\text{YCOUNT} = 240\), and \(\text{DI\_SEL} = 1\) (the \(\text{DI\_SEL}\) bit is located in \(\text{DMA\_CONF}\)) will interrupt on every row transferred, for the entire frame.
Data Transfer Scenarios

- Setting \( XCOUNT = 320, YCOUNT = 240 \), and \( DI_SEL = 0 \) will interrupt only on the completion of the frame (when 240 rows of 320 bytes have been transferred).

- Setting \( XCOUNT = 38,400 \) (320 x 120), \( YCOUNT = 2 \), and \( DI_SEL = 1 \) will cause an interrupt when half of the frame has been transferred, and again when the whole frame has been transferred.

Following is the general procedure for setting up DMA operation with the PPI. Refer to “DMA and Memory DMA MMRs” on page 9-3 for details regarding configuration of DMA.

1. Configure DMA registers as appropriate for desired DMA operating mode.
2. Enable the DMA channel for operation.
3. Configure appropriate PPI registers.
4. Enable the PPI by writing a 1 to bit 0 in PPI_CONTROL.

Data Transfer Scenarios

Figure 11-16 shows two possible ways to use the PPI to transfer in video. These diagrams are very generalized, and bandwidth calculations must be made only after factoring in the exact PPI mode and settings (for example, transfer Field 1 only, transfer odd and even elements).

The top part of the diagram shows a situation appropriate for, as an example, JPEG compression. The first N rows of video are DMA’ed into L1 memory via the PPI. Once in L1, the compression algorithm operates on the data and sends the compressed result out from the processor via the SPORT. Note that no SDRAM access was necessary in this approach.
The bottom part of the diagram takes into account a more formidable compression algorithm, such as MPEG-2 or MPEG-4. Here, the raw video is transferred directly into SDRAM. Independently, a memory DMA channel transfers data blocks between SDRAM and L1 memory for intermediate processing stages. Finally, the compressed video exits the processor via the SPORT.

Figure 11-16. PPI Possible Data Transfer Scenarios
Data Transfer Scenarios
The three universal asynchronous receiver/transmitters (UART) are full-duplex peripherals, compatible with PC-style industry-standard UARTs. The ADSP-BF538 has three UARTs. Each UART converts data between serial and parallel formats. The serial communication follows an asynchronous protocol that supports various word length, stop bits, and parity generation options. Each UART includes interrupt handling hardware. Interrupts can be generated from 12 different events.

Each UART supports the half-duplex IrDA® (Infrared Data Association) SIR (9.6/115.2 Kbps rate) protocol. This is a mode-enabled feature.

Modem status and control functionality is not supported by the UART modules, but may be implemented using general-purpose I/O (GPIO) pins.

Each UART is a DMA-capable peripheral with support for separate TX and RX DMA master channels. It can be used in either DMA or programmed non-DMA mode of operation. The non-DMA mode requires software management of the data flow using either interrupts or polling. The DMA method requires minimal software intervention as the DMA engine itself moves the data. See Chapter 9, “Direct Memory Access” for more information on DMA.

UART0 has dedicated pins, but UART1 and UART2 have pins that can be used as GPIO. For more information, see “General-Purpose Input/Output Ports C, D, E” on page 15-1.
For UART0, either of the peripheral timers can be used to provide a hardware assisted autobaud detection mechanism. See Chapter 16, “Timers” for more information.

UART1 and UART2 do not provide autobaud capabilities.

Serial Communications

Each UART follows an asynchronous serial communication protocol with these options:

- 5 to 8 data bits
- 1, 1½, or 2 stop bits
- None, even, or odd parity
- Baud rate = \( \frac{SCLK}{16 \times \text{Divisor}} \), where SCLK is the system clock frequency and divisor can be a value from 1 to 65536

All data words require a start bit and at least one stop bit. With the optional parity bit, this creates a 7- to 12-bit range for each word. The format of received and transmitted character frames is controlled by the Line control register (UARTx_LCR). Data is always transmitted and received least significant bit (LSB) first.

Figure 12-1 shows a typical physical bit stream measured on the TX pin.

UART Control and Status Registers

The processor provides a set of PC-style industry-standard control and status registers for each UART. These memory-mapped registers (MMRs) are byte-wide registers that are mapped as half words with the most significant byte zero-filled.
UART Port Controllers

Consistent with industry-standard interfaces, multiple registers are mapped to the same address location. The divisor latch registers (UARTx_DLH and UARTx_DLL) share their addresses with the transmit holding register (UARTx_THR), the receive buffer register (UARTx_RBR), and the interrupt enable register (UARTx_IER). The divisor latch access bit (DLAB) in the line control register (UARTx_LCR) controls which set of registers is accessible at a given time. Software must use 16-bit word load/store instructions to access these registers.

Transmit and receive channels are both buffered. The UARTx_THR register buffers the transmit shift register (TSR) and the UARTx_RBR register buffers the receive shift register (LSR). The shift registers are not directly accessible by software.

**UART Line Control (UARTx_LCR) Register**

The UART line control register (UARTx_LCR) controls the format of received and transmitted character frames (see Figure 12-2). The SB bit functions even when the UART clock is disabled. Since the TX pin normally drives high, it can be used as a flag output pin, if the UART is not used.

![Figure 12-1. Bit Stream on the TXx Pin](image-url)
UART Control and Status Registers

UART Line Control Register (UARTx_LCR)

- **DLAB (Divisor Latch Access)**
  - 1 - Enables access to UARTx_DLL and UARTx_DLH
  - 0 - Enables access to UARTx_THR, UARTx_RBR, and UARTx_IER

- **SB (Set Break)**
  - 0 - No force
  - 1 - Force TX pin to 0

- **STP (Stick Parity)**
  - Forces parity to defined value if set and PEN = 1
  - EPS = 1, parity transmitted and checked as 0
  - EPS = 0, parity transmitted and checked as 1

- **EPS (Even Parity Select)**
  - 1 - Even parity
  - 0 - Odd parity when PEN = 1 and STP = 0

- **WLS[1:0] (Word Length Select)**
  - 00 - 5-bit word
  - 01 - 6-bit word
  - 10 - 7-bit word
  - 11 - 8-bit word

- **STB (Stop Bits)**
  - 1 - 2 stop bits for non-5-bit word length or 1 1/2 stop bits for 5-bit word length
  - 0 - 1 stop bit

- **PEN (Parity Enable)**
  - 1 - Transmit and check parity
  - 0 - Parity not transmitted or checked

Reset = 0x0000

UART0 0xFFC0 040C
UART1 0xFFC0 200C
UART2 0xFFC0 210C

Figure 12-2. UART Line Control Register

UART Modem Control (UARTx_MCR) Register

The modem control register (UARTx_MCR) controls the UART port, as shown in Figure 12-3. Even if modem functionality is not supported, the modem control register is available in order to support the loop back mode.

Loopback mode forces the TX pin to high and disconnects the receiver’s input from the RX pin, but redirects it to the transmit output internally.
UART Port Controllers

UART Modem Control Register (UARTx_MCR)

UART0 0xFFC0 0410
UART1 0xFFC0 2010
UART2 0xFFC0 2110

Loop (Loopback mode enable)
Forces TX to high and disconnects RX from RSR

UART Line Status Register (UARTx_LSR) Register

The UART line status register (UARTx_LSR) contains UART status information as shown in Figure 12-4.

UART Line Status Register (UARTx_LSR)  RO
UART0 0xFFC0 0414
UART1 0xFFC0 2014
UART2 0xFFC0 2114

DR (Data Ready)
0 - No new data
1 - UARTx_RBR holds new data

OE (Overrun Error)
0 - No overrun
1 - UARTx_RBR overwritten before read

PE (Parity Error)
0 - No parity error
1 - Parity error

Figure 12-4. UART Line Status Register
The break interrupt (BI), overrun error (OE), parity error (PE) and framing error (FE) bits are cleared when the UART line status register (UARTx_LSR) is read. The data ready (DR) bit is cleared when the UART receive buffer register (UARTx_RBR) is read.

Because of the destructive nature of these read operations, special care should be taken. For more information, see “Speculative Load Execution” on page 6-67 and “Conditional Load Behavior” on page 6-68.

The THRE bit indicates that the UART transmit channel is ready for new data and software can write to UARTx_THR. Writes to UARTx_THR clear the THRE bit. It is set again when data is copied from UARTx_THR to the transmit shift register (TSR). The TEMT bit can be evaluated to determine whether a recently initiated transmit operation has been completed.

**UART Transmit Holding (UARTx_THR) Register**

A write to the UART transmit holding register (UARTx_THR) initiates the transmit operation (see Figure 12-5). The data is moved to the internal transmit shift register (TSR) where it is shifted out at a baud rate equal to SCLK/(16 × Divisor) with start, stop, and parity bits appended as required. All data words begin with a 1-to-0-transition start bit. The transfer of data from UARTx_THR to the transmit shift register sets the transmit holding register empty (THRE) status flag in the UART line status register (UARTx_LSR).

The write-only UARTx_THR register is mapped to the same address as the read-only UARTx_RBR and UARTx_DLL registers. To access UARTx_THR, the DLAB bit in UARTx_LCR must be cleared. When the DLAB bit is cleared, writes to this address target the UARTx_THR register, and reads from this address return the UARTx_RBR register.

Note data is transmitted and received least significant bit (LSB) first (bit 0) followed by the most significant bits (MSBs).
UART Port Controllers

UART Receive Buffer (UARTx_RBR) Register

The receive operation uses the same data format as the transmit configuration, except that the number of stop bits is always assumed to be 1. After detection of the start bit, the received word is shifted into the receive shift register (RSR) at a baud rate of \( \frac{SCLK}{16 \times \text{divisor}} \). After the appropriate number of bits (including stop bit) is received, the data and any status are updated and the receive shift register is transferred to the UART receive buffer register (UARTx_RBR), shown in Figure 12-6. After the transfer of the received word to the UARTx_RBR buffer and the appropriate synchronization delay, the data ready (DR) status flag is updated.

A sampling clock equal to 16 times the baud rate samples the data as close to the midpoint of the bit as possible. Because the internal sample clock may not exactly match the asynchronous receive data rate, the sampling point drifts from the center of each bit. The sampling point is synchronized again with each start bit, so the error accumulates only over the length of a single word. A receive filter removes spurious pulses of less than two times the sampling clock period.

Figure 12-5. UART Transmit Holding Register
UART Control and Status Registers

The read-only UARTx_RBR register is mapped to the same address as the write-only UARTx_THR and UARTx_DLL registers. To access UARTx_RBR, the DLAB bit in UARTx_LCR must be cleared. When the DLAB bit is cleared, writes to this address target the UARTx_THR register, while reads from this address return the UARTx_RBR register.

UART Receive Buffer Register (UARTx_RBR)RO
UART0 0xFFC0 0400
UART1 0xFFC0 2000
UART2 0xFFC0 2100

Figure 12-6. UART Receive Buffer Register

UART Interrupt Enable (UARTx_IER) Register

The UART interrupt enable register (UARTx_IER) is used to enable requests for system handling of empty or full states of UART data registers (see Figure 12-7). Unless polling is used as a means of action, the ERBFI and/or ETBEI bits in this register are normally set.

Setting this register without enabling system DMA causes the UART to notify the processor of data inventory state by means of interrupts. For proper operation in this mode, system interrupts must be enabled, and appropriate interrupt handling routines must be present. For backward compatibility, the UARTx_IIR still reflects the correct interrupt status.

The UART features three separate interrupt channels to handle data transmit, data receive, and line status events independently, regardless whether DMA is enabled or not.
With system DMA enabled, the UART uses DMA to transfer data to or from the processor. Dedicated DMA channels are available to receive and transmit operation. Line error handling can be configured completely independently from the receive/transmit setup.

The UARTx_IER register is mapped to the same address as UARTx_DLH. To access UARTx_IER, the DLAB bit in UARTx_LCR must be cleared.

UART Interrupt Enable Register (UARTx_IER)
UART0 0xFFC0 0404
UART1 0xFFC0 2004
UART2 0xFFC0 2104

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>ERBFI (Enable Receive Buffer Full interrupt) 0 - No interrupt 1 - Generate RX interrupt if DR bit in UARTx_LSR is set</td>
</tr>
<tr>
<td>14</td>
<td>ETBEI (Enable Transmit Buffer Empty interrupt) 0 - No interrupt 1 - Generate TX interrupt if THRE bit in UARTx_LSR is set</td>
</tr>
<tr>
<td>13</td>
<td>ELSI (Enable RX status interrupt) 0 - No interrupt 1 - Generate line status interrupt if any of UARTx_LSR[4:1] is set</td>
</tr>
</tbody>
</table>

Reset = 0x0000

Figure 12-7. UART Interrupt Enable Register

The UART DMA is enabled by first setting up the system DMA control registers and then enabling the UART ERBFI and/or ETBEI interrupts in the UARTx_IER register. This is because the interrupt request lines double as DMA request lines. Depending on whether DMA is enabled or not, upon receiving these requests, the DMA control unit either generates a direct memory access or passes the UART interrupt on to the system interrupt handling unit. However, the UART error interrupt goes directly to the system interrupt handling unit, bypassing the DMA unit completely.
UART Control and Status Registers

The **ELSI** bit enables interrupt generation on an independent interrupt channel when any of the following conditions are raised by the respective bit in the UART Line status register (**UARTx_LSR**):

- Receive overrun error (**OE**)
- Receive parity error (**PE**)
- Receive framing error (**FE**)
- Break interrupt (**BI**)

When the **ETBEI** bit is set in the **UARTx_IER** register, the UART module immediately issues an interrupt or DMA request. When initiating the transmission of a string, no special handling of the first character is required. Set the **ETBEI** bit and let the interrupt service routine load the first character from memory and write it to the **UARTx_THR** register in the normal manner. Accordingly, the **ETBEI** bit should be cleared if the string transmission has completed.

UART Interrupt Identification (UARTx_IIR) Register

For legacy reasons, the UART interrupt identification register (**UARTx_IIR**) still reflects the UART interrupt status. Legacy operation may require bundling all UART interrupt sources to a single interrupt channel and servicing them all by the same software routine. This can be established by globally assigning all UART interrupts to the same interrupt priority, by using the system interrupt controllers (SICx).

When cleared, the pending interrupt bit (**NINT**) signals that an interrupt is pending. The **STATUS** field indicates the highest priority pending interrupt. The receive line status has the highest priority; the **UARTx_THR** empty interrupt has the lowest priority. In the case where both interrupts are signalling, the **UARTx_IIR** reads 0x06.
When a UART interrupt is pending, the interrupt service routine (ISR) needs to clear the interrupt latch explicitly. The following figure shows how to clear any of the three latches.

UART Interrupt Identification Register (UARTx_IIR) RO

<table>
<thead>
<tr>
<th></th>
<th>UARTx_IIR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UARTx_THR empty. Write UARTx_THR or read UARTx_IIR to clear interrupt request.</td>
</tr>
<tr>
<td>1</td>
<td>Receive line status. Read UARTx_LSR to clear interrupt request.</td>
</tr>
</tbody>
</table>

Figure 12-8. UART Interrupt Identification Register

The TX interrupt request is cleared by writing new data to the UARTx_THR register or by reading the UARTx_IIR register. Note the special role of the UARTx_IIR register read in the case where the service routine does not want to transmit further data.

If software stops transmission, it must read the UARTx_IIR register to reset the interrupt request. As long as the UARTx_IIR register reads 0x04 or 0x06 (indicating that another interrupt of higher priority is pending), the UARTx_THR empty latch cannot be cleared by reading UARTx_IIR.

If either the line status interrupt or the receive data interrupt has been assigned a lower interrupt priority by the SICx, a deadlock condition can occur. To avoid this, always assign the lowest priority of the enabled UART interrupts to the UARTx_THR empty event.

Because of the destructive nature of these read operations, special care should be taken. For more information, see “Speculative Load Execution” on page 6-67 and “Conditional Load Behavior” on page 6-68.
UART Control and Status Registers

UARTx_DLL and UARTx_DLH Registers

The bit rate is characterized by the system clock ($SCLK$) and the 16-bit divisor. The divisor is split into the UART divisor latch low byte register (UART_DLL) and the UART divisor latch high byte register (UARTx_DLH). These registers form a 16-bit divisor (see Figure 12-9). The baud clock is divided by 16 so that:

$$\text{BAUD RATE} = \frac{SCLK}{(16 \times \text{divisor})}$$

Divisor = 65,536 when UARTx_DLL = UARTx_DLH = 0

UART Divisor Latch Low Byte Register (UARTx_DLL)
UART0 0xFFC0 0400
UART1 0xFFC0 2000
UART2 0xFFC0 2100

UART Divisor Latch High Byte Register (UARTx_DLH)
UART0 0xFFC0 0404
UART1 0xFFC0 2004
UART2 0xFFC0 2104

Figure 12-9. UART Divisor Latch Registers

The UART_DLL register is mapped to the same address as the UARTx_THR and UARTx_RBR registers. The UARTx_DLH register is mapped to the same address as the interrupt enable register (UARTx_IER). The DLAB bit in UARTx_LCR must be set before the UART divisor latch registers can be accessed.
Note the 16-bit divisor formed by \texttt{UARTx\_DLH} and \texttt{UARTx\_DLL} resets to 0x0001, resulting in the highest possible clock frequency by default. If the UART is not used, disabling the UART clock saves power. The \texttt{UARTx\_DLH} and \texttt{UARTx\_DLL} registers can be programmed by software before or after setting the \texttt{UCEN} bit.

Table 12-1 provides example divide factors required to support most standard baud rates.

Table 12-1. UART Baud Rate Examples With 100 MHz SCLK

<table>
<thead>
<tr>
<th>Baud Rate</th>
<th>DL</th>
<th>Actual</th>
<th>% Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>2400</td>
<td>2604</td>
<td>2400.15</td>
<td>.006</td>
</tr>
<tr>
<td>4800</td>
<td>1302</td>
<td>4800.31</td>
<td>.007</td>
</tr>
<tr>
<td>9600</td>
<td>651</td>
<td>9600.61</td>
<td>.006</td>
</tr>
<tr>
<td>19200</td>
<td>326</td>
<td>19171.78</td>
<td>.147</td>
</tr>
<tr>
<td>38400</td>
<td>163</td>
<td>38343.56</td>
<td>.147</td>
</tr>
<tr>
<td>57600</td>
<td>109</td>
<td>57339.45</td>
<td>.452</td>
</tr>
<tr>
<td>115200</td>
<td>54</td>
<td>115740.74</td>
<td>.469</td>
</tr>
<tr>
<td>921600</td>
<td>7</td>
<td>892857.14</td>
<td>3.119</td>
</tr>
<tr>
<td>6250000</td>
<td>1</td>
<td>6250000</td>
<td>–</td>
</tr>
</tbody>
</table>

Careful selection of \texttt{SCLK} frequencies, that is, even multiples of desired baud rates, can result in lower error percentages.
UART Control and Status Registers

UART Scratch (UARTx_SCR) Register

The contents of the 8-bit UART scratch register (UARTx_SCR) shown in Figure 12-10 is reset to 0x00. It is used for general-purpose data storage and does not control the UART hardware in any way.

![UART Scratch Register (UARTx_SCR)](image)

Figure 12-10. UART Scratch Register

UART Global Control (UARTx_GCTL) Register

The UART global control register (UARTx_GCTL) contains the enable bit for internal UART clocks and for the IrDA mode of operation of the UART (see Figure 12-11).

Note that the UCEN bit was not present in previous UART implementations. It has been introduced to save power if the UART is not used. When porting code, be sure to enable this bit.

The IrDA TX polarity change bit and the IrDA RX polarity change bit are effective only in IrDA mode. The two force error bits, FPE and FFE, are intended for test purposes. They are useful for debugging software, especially in loop back mode.
UART Port Controllers

Non-DMA Mode

In non-DMA mode, data is moved to and from the UART by the processor core. To transmit a character, load it into \( \text{UART}_x\_\text{THR} \). Received data can be read from \( \text{UART}_x\_\text{RBR} \). The processor must write and read one character at a time.

To prevent any loss of data and misalignments of the serial data stream, the UART Line status register (\( \text{UART}_x\_\text{LSR} \)) provides two status flags for handshaking—\( \text{THRE} \) and \( \text{DR} \).

The \( \text{THRE} \) flag is set when \( \text{UART}_x\_\text{THR} \) is ready for new data and cleared when the processor loads new data into \( \text{UART}_x\_\text{THR} \). Writing \( \text{UART}_x\_\text{THR} \) when it is not empty overwrites the register with the new value and the previous character is never transmitted.
The DR flag signals when new data is available in UARTx_RBR. This flag is cleared automatically when the processor reads from UARTx_RBR. Reading UARTx_RBR when it is not full returns the previously received value. When UARTx_RBR is not read in time, newly received data overwrites UARTx_RBR and the overrun (OE) flag is set.

With interrupts disabled, these status flags can be polled to determine when data is ready to move. Note that because polling is processor intensive, it is not typically used in real-time signal processing environments. Software can write up to two words into the UARTx_THR register before enabling the UART clock. As soon as the UCEN bit is set, those two words are sent.

Alternatively, UART writes and reads can be accomplished by interrupt service routines (ISRs). Separate interrupt lines are provided for UART TX, UART RX, and UART Error. The independent interrupts can be enabled individually by the UARTx_IER register.

The ISRs can evaluate the status bit field within the UART interrupt identification register (UARTx_IIR) to determine the signalling interrupt source. If more than one source is signalling, the status field displays the one with the highest priority. Interrupts also must be assigned and unmasked by the processor’s interrupt controller. The ISRs must clear the interrupt latches explicitly. See Figure 12-8.

**DMA Mode**

In this mode, separate receive (RX) and transmit (TX) DMA channels move data between the UART and memory. The software does not have to move data, it just has to set up the appropriate transfers either through the descriptor mechanism or through auto buffer mode.
No additional buffering is provided in the UART DMA channel, so the latency requirements are the same as in non-DMA mode. However, the latency is determined by the bus activity and arbitration mechanism and not by the processor loading and interrupt priorities. For more information, see Chapter 9, “Direct Memory Access”.

DMA interrupt routines must explicitly write 1s to the corresponding DMA IRQ status registers to clear the latched request of the pending interrupt.

The UART DMA is enabled by first setting up the system DMA control registers and then enabling the UART ERBF1 and/or ETBE1 interrupts in the UARTx_IER register. This is because the interrupt request lines double as DMA request lines. Depending on whether DMA is enabled or not, upon receiving these requests, the DMA control unit either generates a direct memory access or passes the UART interrupt on to the system interrupt handling unit. However, the UART error interrupt goes directly to the system interrupt handling unit, bypassing the DMA unit completely.

The UART DMA supports 8-bit operation.

### Mixing Modes

Non-DMA and DMA modes use different synchronization mechanisms. Consequently, any serial communication must be complete before switching from non-DMA to DMA mode or vice versa. In other words, before switching from non-DMA transmission to DMA transmission, make sure both UARTx_THR and the internal transmit shift register (TSR) are empty by testing the THRE and the TEMT status bits in UARTx_LSR. Otherwise, the processor must wait until the 2-bit DMA buffer status field within the appropriate UART transmit DMA configuration register (UARTx_CONFIG_TX) is clear.
When switching from DMA to non-DMA operation, make sure both the receive (RX) and transmit (TX) DMA channels have completely transferred their data, including data contained in the DMA FIFOs. While the DMA RX interrupt indicates the last data word has been written to memory (and has left the DMA FIFO), the DMA TX interrupt indicates the last data word has left memory (and has entered the DMA FIFO). The processor must wait until the TX FIFO is empty, by testing that the DMA_RUN status bit in the TX channel’s IRQ_STATUS register is clear, before it is safe to disable the DMA channel.

**IrDA Support**

Aside from the standard UART functionality, the UART also supports half-duplex serial data communication via infrared signals, according to the recommendations of the Infrared Data Association (IrDA). The physical layer known as IrDA SIR (9.6/115.2 Kbps rate) is based on return-to-zero-inverted (RZI) modulation. Pulse position modulation is not supported.

Using the 16x data rate clock, RZI modulation is achieved by inverting and modulating the non-return-to-zero (NRZ) code normally transmitted by the UART. On the receive side, the 16x clock is used to determine an IrDA pulse sample window, from which the RZI-modulated NRZ code is recovered.

IrDA support is enabled by setting the IREN bit in the UART global control register. The IrDA application requires external transceivers.

**IrDA Transmitter Description**

To generate the IrDA pulse transmitted by the UART, the normal NRZ output of the transmitter is first inverted, so a 0 is transmitted as a high pulse of 16 UART clock periods and a 1 is transmitted as a low pulse for 16 UART clock periods. The leading edge of the pulse is then delayed by
six UART clock periods. Similarly, the trailing edge of the pulse is truncated by eight UART clock periods. This results in the final representation of the original 0 as a high pulse of only 3/16 clock periods in a 16-cycle UART clock period. The pulse is centered around the middle of the bit time, as shown in Figure 12-12. The final IrDA pulse is fed to the off-chip infrared driver.

![IrDA Transmit Pulse Diagram](image)

**Figure 12-12. IrDA Transmit Pulse**

This modulation approach ensures a pulse width output from the UART of three cycles high out of every 16 UART clock cycles. As shown in Table 12-1, the error terms associated with the baud rate generator are very small and well within the tolerance of most infrared transceiver specifications.

**IrDA Receiver Description**

The IrDA receiver function is more complex than the transmit function. The receiver must discriminate the IrDA pulse and reject noise. To do this, the receiver looks for the IrDA pulse in a narrow window centered around the middle of the expected pulse.
Glitch filtering is accomplished by counting 16 system clocks from the time an initial pulse is seen. If the pulse is absent when the counter expires, it is considered a glitch. Otherwise, it is interpreted as a 0. This is acceptable because glitches originating from on-chip capacitive cross-coupling typically do not last for more than a fraction of the system clock period. Sources outside of the chip and not part of the transmitter can be avoided by appropriate shielding. The only other source of a glitch is the transmitter itself. The processor relies on the transmitter to perform within specification. If the transmitter violates the specification, unpredictable results may occur. The 4-bit counter adds an extra level of protection at a minimal cost. Note because the system clock can change across systems, the longest glitch tolerated is inversely proportional to the system clock frequency.

The receive sampling window is determined by a counter that is clocked at the 16x bit-time sample clock. The sampling window is re-synchronized with each start bit by centering the sampling window around the start bit.

The polarity of receive data is selectable, using the \texttt{IRPOL} bit. Figure 12-13 gives examples of each polarity type.

- \texttt{IRPOL} = 0 assumes that the receive data input idles 0 and each active 1 transition corresponds to a UART NRZ value of 0.
- \texttt{IRPOL} = 1 assumes that the receive data input idles 1 and each active 0 transition corresponds to a UART NRZ value of 0.
Figure 12-13. IrDA Receiver Pulse Detection
IrDA Support
The ADSP-BF538 processor has four identical serial ports (SPORT). These support a variety of serial data communications protocols and can provide a direct interconnection between processors in a multiprocessor system.

The serial ports (SPORT0, SPORT1, SPORT2, SPORT3) provide an I/O interface to a wide variety of peripheral serial devices. SPORTs provide synchronous serial data transfer only; the processor provides asynchronous RS-232 data transfer via the UART. Each SPORT has one group of pins (primary data, secondary data, clock, and frame sync) for transmit and a second set of pins for receive. The receive and transmit functions are programmed separately. Each SPORT is a full duplex device, capable of simultaneous data transfer in both directions. The SPORTs can be programmed for bit rate, frame sync, and number of bits per word by writing to memory-mapped registers.

The naming conventions for registers and pins use a lower case \( x \) to represent a digit. For example, \( \text{RFS}_x \) indicates pins \( \text{RFS}_0, \text{RFS}_1, \text{RFS}_2, \text{or} \text{RFS}_3 \), corresponding to SPORT0, SPORT1, SPORT2 and SPORT3. LSB refers to least significant bit, and MSB refers to most significant bit.

All SPORTs have the same capabilities and are programmed in the same way. Each SPORT has its own set of control registers and data buffers.

The SPORTs use frame sync pulses to indicate the beginning of each word or packet, and the bit clock marks the beginning of each data bit. External bit clock and frame sync are available for the TX and RX buffers.
With a range of clock and frame synchronization options, the SPORTs allow a variety of serial communication protocols, including H.100, and provide a glueless hardware interface to many industry-standard data converters and codecs.

The SPORTs can operate at up to an $SCLK/2$ clock rate with an externally generated clock, or half the system clock rate for an internally generated serial port clock. The SPORT external clock must always be less than the $SCLK$ frequency. Independent transmit and receive clocks provide greater flexibility for serial communications.

SPORT clocks and frame syncs can be internally generated by the system or received from an external source. The SPORTs can operate with a transmission format of LSB first or MSB first, with word lengths selectable from 3 to 32 bits. They offer selectable transmit modes and optional μ-law or A-law companding in hardware. SPORT data can be automatically transferred between on-chip and off-chip memories using DMA block transfers. Additionally, each of the SPORTs offers a TDM (time-division-multiplexed) multichannel mode.

Each of the SPORTs offers these features and capabilities:

- Provides independent transmit and receive functions.

- Transfers serial data words from 3 to 32 bits in length, either MSB first or LSB first.

- Provides alternate framing and control for interfacing to I$^2$S serial devices, as well as other audio formats (for example, left-justified stereo serial data).
Serial Port Controllers

• Has FIFO plus double buffered data (both receive and transmit functions have a data buffer register and a shift register), providing additional time to service the SPORT.

• Permits chaining of DMA operations for multiple data blocks.

• Provides two synchronous transmit and two synchronous receive data pins and buffers in each SPORT to double the total supported data streams.

• Performs A-law and μ-law hardware companding on transmitted and received words. (See “Companding” on page 13-37 for more information.)

• Internally generates serial clock and frame sync signals in a wide range of frequencies or accepts clock and frame sync input from an external source.

• Operates with or without frame synchronization signals for each data word, with internally generated or externally generated frame signals, with active high or active low frame signals, and with either of two configurable pulse widths and frame signal timing.

• Performs interrupt-driven, single word transfers to and from on-chip memory under processor control.

• Provides direct memory access transfer to and from memory under DMA Master control. DMA can be auto buffer-based (a repeated, identical range of transfers) or descriptor-based (individual or repeated ranges of transfers with differing DMA parameters).

• Executes DMA transfers to and from on-chip memory. Each SPORT can automatically receive and transmit an entire block of data.

• Has a multichannel mode for TDM interfaces. Each SPORT can receive and transmit data selectively from a time-division-multiplexed serial bit stream on 128 contiguous channels from a stream.
of up to 1024 total channels. This mode can be useful as a network communication scheme for multiple processors. The 128 channels available to the processor can be selected to start at any channel location from 0 to 895 = (1023 – 128). Note the multichannel select registers and the WSIZE register control which subset of the 128 channels within the active region can be accessed.

Table 13-1 shows the pins for each SPORT.

Table 13-1. Serial Port (SPORT) Pins

<table>
<thead>
<tr>
<th>Pin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DTxPRI</td>
<td>Transmit Data Primary</td>
</tr>
<tr>
<td>DTxSEC</td>
<td>Transmit Data Secondary</td>
</tr>
<tr>
<td>TSCLKx</td>
<td>Transmit Clock</td>
</tr>
<tr>
<td>TFSx</td>
<td>Transmit Frame Sync</td>
</tr>
<tr>
<td>DRxPRI</td>
<td>Receive Data Primary</td>
</tr>
<tr>
<td>DRxSEC</td>
<td>Receive Data Secondary</td>
</tr>
<tr>
<td>RSCLKx</td>
<td>Receive Clock</td>
</tr>
<tr>
<td>RFSx</td>
<td>Receive Frame Sync</td>
</tr>
</tbody>
</table>

A lowercase x within a pin name represents a possible value of 0, 1, 2, or 3 (corresponding to SPORT0, SPORT1, SPORT2, or SPORT3).

SPORT0 and SPORT1 have dedicated pins, whereas SPORT2 and SPORT3 utilize GPIO pins. For more information, see “General-Purpose Input/Output Ports C, D, E” on page 15-1.

A SPORT receives serial data on its DRxPRI and DRxSEC inputs and transmits serial data on its DTxPRI and DTxSEC outputs. It can receive and transmit simultaneously for full-duplex operation. For transmit, the data bits (DTxPRI and DTxSEC) are synchronous to the transmit clock (TSCLKx). For receive, the data bits (DRxPRI and DRxSEC) are synchronous to the receive clock (RSCLKx). The serial clock is an output if the processor
generates it, or an input if the clock is externally generated. Frame synchronization signals \texttt{RFS}\textsubscript{x} and \texttt{TFS}\textsubscript{x} are used to indicate the start of a serial data word or stream of serial words.

The primary and secondary data pins provide a method to increase the data throughput of the serial port. They do not behave as totally separate SPORTs; rather, they operate in a synchronous manner (sharing clock and frame sync) but on separate data. The data received on the primary and secondary pins is interleaved in main memory and can be retrieved by setting a stride in the data address generators (DAG) unit. For more information about DAGs, see Chapter 5, “Data Address Generators”.

Similarly, for TX, data should be written to the TX register in an alternating manner—first primary, then secondary, then primary, then secondary, and so on. This is easily accomplished with the processor’s powerful DAGs.

In addition to the serial clock signal, data must be signalled by a frame synchronization signal. The framing signal can occur either at the beginning of an individual word or at the beginning of a block of words.

\textbf{Figure 13-1} shows a simplified block diagram of a single SPORT. Data to be transmitted is written from an internal processor register to the SPORT’s \texttt{SPORTx\_TX} register via the peripheral bus. This data is optionally compressed by the hardware and automatically transferred to the TX shift register. The bits in the shift register are shifted out on the SPORT’s \texttt{DTx-PRI/DTxSEC} pin, MSB first or LSB first, synchronous to the serial clock on the \texttt{TSCLKx} pin. The receive portion of the SPORT accepts data from the \texttt{DRxPRI/DRxSEC} pin synchronous to the serial clock on the \texttt{RSCLKx} pin. When an entire word is received, the data is optionally expanded, then automatically transferred to the SPORT’s \texttt{SPORTx\_RX} register, and then into the RX FIFO where it is available to the processor.
Figure 13-1. SPORT Block Diagram

NOTE 1: ALL WIDE ARROW DATA PATHS ARE 16 OR 32 BITS WIDE, DEPENDING ON SLEN. FOR SLEN = 2 TO 15, A 16-BIT DATA PATH WITH 8-DEEP FIFO IS USED. FOR SLEN = 16 TO 31, A 32-BIT DATA PATH WITH 4-DEEP FIFO IS USED.

NOTE 2: Tx REGISTER IS THE BOTTOM OF THE Tx FIFO, Rx REGISTER IS THE TOP OF THE Rx FIFO.
Figure 13-2 shows a possible port connection for the SPORTs. Note serial devices A and B must be synchronous, as they share common frame syncs and clocks. The same is true for serial devices C and D.

Figure 13-2. SPORT Connections
Figure 13-3 shows an example of a stereo serial device with three transmit and two receive channels connected to the processor.

Figure 13-3. Stereo Serial Connection

**SPORT Operation**

This section describes general SPORT operation, illustrating the most common use of a SPORT. Since the SPORT functionality is configurable, this description represents just one of many possible configurations.
Writing to a SPORT’s `SPORTx_TX` register readies the SPORT for transmission. The `TFSx` signal initiates the transmission of serial data. Once transmission has begun, each value written to the `SPORTx_TX` register is transferred through the FIFO to the internal transmit shift register. The bits are then sent, beginning with either the MSB or the LSB as specified in the `SPORTx_TCR1` register. Each bit is shifted out on the driving edge of `TSCLKx`. The driving edge of `TSCLKx` can be configured to be rising or falling. The SPORT generates the transmit interrupt or requests a DMA transfer as long as there is space in the TX FIFO.

As a SPORT receives bits, they accumulate in an internal receive register. When a complete word has been received, it is written to the SPORT FIFO register and the receive interrupt for that SPORT is generated or a DMA transfer is initiated. Interrupts are generated differently if DMA block transfers are performed. For information about DMA, see Chapter 9, “Direct Memory Access”.

**SPORT Disable**

The SPORTs are automatically disabled by a processor hardware or software reset. A SPORT can also be disabled directly by clearing the SPORT’s transmit or receive enable bits (TSPEN in the `SPORTx_TCR1` register and RSPEN in the `SPORTx_RCR1` register, respectively). Each method has a different effect on the SPORT.

A processor reset disables the SPORTs by clearing the `SPORTx_TCR1`, `SPORTx_TCR2`, `SPORTx_RCR1`, and `SPORTx_RCR2` registers (including the TSPEN and RSPEN enable bits) and the `SPORTx_TCLKDIV`, `SPORTx_RCLKDIV`, `SPORTx_TFSDIV`, and `SPORTx_RFSDIV` clock and frame sync divisor registers. Any ongoing operations are aborted.
Clearing the TSPEN and RSPEN enable bits disables the SPORTs and aborts any ongoing operations. Status bits are also cleared. Configuration bits remain unaffected and can be read by the software in order to be altered or overwritten. To disable the SPORT output clock, set the SPORT to be disabled.

Note that disabling a SPORT via TSPEN/RSPEN may shorten any currently active pulses on the TFSx/RFSx and TSCLKx/RSCLKx pins, if these signals are configured to be generated internally.

When disabling the SPORT from multichannel operation, first disable TSPEN and then disable RSPEN. Note both TSPEN and RSPEN must be disabled before re-enabling. Disabling only TX or RX is not allowed.

**Setting SPORT Modes**

SPORT configuration is accomplished by setting bit and field values in configuration registers. Each SPORT must be configured prior to being enabled. Once the SPORT is enabled, further writes to the SPORT configuration registers are disabled (except for SPORTx_RCLKDIV, SPORTx_TCLKDIV, and multichannel mode channel select registers). To change values in all other SPORT configuration registers, disable the SPORT by clearing TSPEN in SPORTx_TCR1 and/or RSPEN in SPORTx_RCR1.

Each SPORT has its own set of control registers and data buffers. These registers are described in detail in the following sections. All control and status bits in the SPORT registers are active high unless otherwise noted.
Register Writes and Effective Latency

When the SPORT is disabled (TSPEN and RSPEN cleared), SPORT register writes are internally completed at the end of the SCLK cycle in which they occurred, and the register reads back the newly-written value on the next cycle.

When the SPORT is enabled to transmit (TSPEN set) or receive (RSPEN set), corresponding SPORT configuration register writes are disabled (except for SPORTx_RCLKDIV, SPORTx_TCLKDIV, and multichannel mode channel select registers). The SPORTx_TX register writes are always enabled; SPORTx_RX, SPORTx_CHNL, and SPORTx_STAT are read-only registers.

After a write to a SPORT register, while the SPORT is disabled, any changes to the control and mode bits generally take effect when the SPORT is re-enabled.

Most configuration registers can only be changed while the SPORT is disabled (TSPEN/RSPEN = 0). Changes take effect after the SPORT is re-enabled. The only exceptions to this rule are the TCLKDIV/RCLKDIV registers and multichannel select registers.

SPORT Transmit Configuration
(SPORTx_TCR1, SPORTx_TCR2) Registers

The main control registers for the transmit portion of each SPORT are the transmit configuration registers, SPORTx_TCR1 and SPORTx_TCR2.

A SPORT is enabled for transmit if bit 0 (TSPEN) of the transmit configuration 1 register is set to 1. This bit is cleared during either a hard reset or a soft reset, disabling all SPORT transmission.
SPORT Transmit Configuration (SPORTx_TCR1, SPORTx_TCR2)

Registers

When the SPORT is enabled to transmit (TSPEN set), corresponding SPORT configuration register writes are not allowed except for SPORTx_TCLKDIV and multichannel mode channel select registers. Writes to disallowed registers have no effect. While the SPORT is enabled, SPORTx_TCR1 is not written except for bit 0 (TSPEN). For example:

write (SPORTx_TCR1, 0x0001); /* SPORT TX Enabled */
write (SPORTx_TCR1, 0xFF01); /* ignored, no effect */
write (SPORTx_TCR1, 0xFFF0); /* SPORT disabled, SPORTx_TCR1 still equal to 0x0000 */

The addresses for these SPORT registers are:

- SPORT0_TCR1 – 0xFFC0 0800
- SPORT0_TCR2 – 0xFFC0 0804
- SPORT1_TCR1 – 0xFFC0 0900
- SPORT1_TCR2 – 0xFFC0 0904
- SPORT2_TCR1 – 0xFFC0 2500
- SPORT2_TCR2 – 0xFFC0 2504
- SPORT3_TCR1 – 0xFFC0 2600
- SPORT3_TCR2 – 0xFFC0 2604
SPORTx Transmit Configuration 1 Register (SPORTx_TCR1)

- **TCKFE (Clock Falling Edge Select)**
  - 0: Drive data and internal frame syncs with rising edge of TSCLK. Sample external frame syncs with falling edge of TSCLK.
  - 1: Drive data and internal frame syncs with falling edge of TSCLK. Sample external frame syncs with rising edge of TSCLK.

- **LATFS (Late Transmit Frame Sync)**
  - 0: Early frame syncs
  - 1: Late frame syncs

- **LTFS (Low Transmit Frame Sync Select)**
  - 0: Active high TFS
  - 1: Active low TFS

- **DITFS (Data-Independent Transmit Frame Sync Select)**
  - 0: Data-dependent TFS generated
  - 1: Data-independent TFS generated

- **TSPEN (Transmit Enable)**
  - 0: Transmit disabled
  - 1: Transmit enabled

- **ITCLK (Internal Transmit Clock Select)**
  - 0: External transmit clock selected
  - 1: Internal transmit clock selected

- **TLSBIT (Transmit Bit Order)**
  - 00: Early frame syncs
  - 01: Late frame syncs
  - 10: Compand using μ-law
  - 11: Compand using A-law

- **TFSR (Transmit Frame Sync Required Select)**
  - 0: Does not require TFS for every data word
  - 1: Requires TFS for every data word

**Reset = 0x0000**

Figure 13-4. SPORTx Transmit Configuration 1 Register
SPORT Transmit Configuration (SPORTx_TCR1, SPORTx_TCR2) Registers

Additional information for the SPORTx_TCR1 and SPORTx_TCR2 transmit configuration register bits includes:

- **Transmit Enable** ([TSPEN](#)). This bit selects whether the SPORT is enabled to transmit (if set) or disabled (if cleared).

  Setting [TSPEN](#) causes an immediate assertion of a SPORT TX interrupt, indicating that the TX data register is empty and needs to be filled. This is normally desirable because it allows centralization of the transmit data write code in the TX interrupt service routine (ISR). For this reason, the code should initialize the ISR and be ready to service TX interrupts before setting [TSPEN](#).

  Similarly, if DMA transfers are used, DMA control should be configured correctly before setting [TSPEN](#). Set all DMA control registers before setting [TSPEN](#).

  Clearing [TSPEN](#) causes the SPORT to stop driving data, TSCLK, and frame sync pins; it also shuts down the internal SPORT circuitry. In low power applications, battery life can be extended by clearing [TSPEN](#) whenever the SPORT is not in use.

---

13-14  ADSP-BF538/ADSP-BF538F Blackfin Processor Hardware Reference
All SPORT control registers should be programmed before TSPEN is set. Typical SPORT initialization code first writes all control registers, including DMA control if applicable. The last step in the code is to write SPORTx_TCR1 with all of the necessary bits, including TSPEN.

- **Internal Transmit Clock Select.** (ITCLK). This bit selects the internal transmit clock (if set) or the external transmit clock on the TSCLK pin (if cleared). The TCLKDIV MMR value is not used when an external clock is selected.

- **Data Formatting Type Select.** The two TDTYPE bits specify data formats used for single and multichannel operation.

- **Bit Order Select.** (TLSBIT). The TLSBIT bit selects the bit order of the data words transmitted over the SPORT.

- **Serial Word Length Select.** (SLEN). The serial word length (the number of bits in each word transmitted over the SPORTs) is calculated by adding 1 to the value of the SLEN field:

  \[
  \text{Serial Word Length} = \text{SLEN} + 1;
  \]

  The SLEN field can be set to a value of 2 to 31; 0 and 1 are illegal values for this field. Three common settings for the SLEN field are 15, to transmit a full 16-bit word; 7, to transmit an 8-bit byte; and 23, to transmit a 24-bit word. The processor can load 16- or 32-bit values into the transmit buffer via DMA or an MMR write instruction; the SLEN field tells the SPORT how many of those bits to shift out of the register over the serial link. The serial port transfers bits [SLEN:0] from the transmit buffer.
SPORT Transmit Configuration (SPORTx_TCR1, SPORTx_TCR2) Registers

The frame sync signal is controlled by the SPORTx_TFSDIV and SPORTx_RFSDIV registers, not by SLEN. To produce a frame sync pulse on each byte or word transmitted, the proper frame sync divider must be programmed into the frame sync divider register; setting SLEN to 7 does not produce a frame sync pulse on each byte transmitted.

- **Internal Transmit Frame Sync Select.** (ITFS). This bit selects whether the SPORT uses an internal TFS (if set) or an external TFS (if cleared).

- **Transmit Frame Sync Required Select.** (TFSR). This bit selects whether the SPORT requires (if set) or does not require (if cleared) a transmit frame sync for every data word.

The TFSR bit is normally set during SPORT configuration. A frame sync pulse is used to mark the beginning of each word or data packet, and most systems need a frame sync to function properly.

- **Data-Independent Transmit Frame Sync Select.** (DITFS). This bit selects whether the SPORT generates a data-independent TFS (sync at selected interval) or a data-dependent TFS (sync when data is present in SPORTx_TX) for the case of internal frame sync select (ITFS = 1). The DITFS bit is ignored when external frame syncs are selected.

The frame sync pulse marks the beginning of the data word. If DITFS is set, the frame sync pulse is issued on time, whether the SPORTx_TX register has been loaded or not; if DITFS is cleared, the frame sync pulse is only generated if the SPORTx_TX data register has been loaded. If the receiver demands regular frame sync pulses, DITFS should be set, and the processor should keep loading the SPORTx_TX register on time. If the receiver can tolerate occasional
late frame sync pulses, \( \text{DITFS} \) should be cleared to prevent the SPORT from transmitting old data twice or transmitting garbled data if the processor is late in loading the \( \text{SPORTx\_TX} \) register.

- **Low Transmit Frame Sync Select.** \( (\text{LTFS}) \). This bit selects an active low \( \text{TFS} \) (if set) or active high \( \text{TFS} \) (if cleared).

- **Late Transmit Frame Sync.** \( (\text{LATFS}) \). This bit configures late frame syncs (if set) or early frame syncs (if cleared).

- **Clock Drive/Sample Edge Select.** \( (\text{TCKFE}) \). This bit selects which edge of the \( \text{TCLKx} \) signal the SPORT uses for driving data, for driving internally generated frame syncs, and for sampling externally generated frame syncs. If set, data and internally generated frame syncs are driven on the falling edge, and externally generated frame syncs are sampled on the rising edge. If cleared, data and internally generated frame syncs are driven on the rising edge, and externally generated frame syncs are sampled on the falling edge.

- **TxSec Enable.** \( (\text{TXSE}) \). This bit enables the transmit secondary side of the serial port (if set).

- **Stereo Serial Enable.** \( (\text{TSFSE}) \). This bit enables the stereo serial operating mode of the serial port (if set). By default this bit is cleared, enabling normal clocking and frame sync.

- **Left/Right Order.** \( (\text{TRFST}) \). If this bit is set, the right channel is transmitted first in stereo serial operating mode. By default this bit is cleared, and the left channel is transmitted first.
SPORT Receive Configuration (SPORTx_RCR1, SPORTx_RCR2) Registers

The main control registers for the receive portion of each SPORT are the receive configuration registers, SPORTx_RCR1 and SPORTx_RCR2.

A SPORT is enabled for receive if bit 0 (RSPEN) of the receive configuration 1 register is set to 1. This bit is cleared during either a hard reset or a soft reset, disabling all SPORT reception.

When the SPORT is enabled to receive (RSPEN set), corresponding SPORT configuration register writes are not allowed except for SPORTx_RCLKDIV and multichannel mode channel select registers. Writes to disallowed registers have no effect. While the SPORT is enabled, SPORTx_RCR1 is not written except for bit 0 (RSPEN). For example:

write (SPORTx_RCR1, 0x0001);  /* SPORT RX Enabled */
write (SPORTx_RCR1, 0xFF01);  /* ignored, no effect */
write (SPORTx_RCR1, 0xFFF0);  /* SPORT disabled, SPORTx_RCR1 still equal to 0x0000 */

The addresses for these SPORT registers are:

SPORT0_RCR1 – 0xFFC0 0820  SPORT0_RCR2 – 0xFFC0 0824
SPORT1_RCR1 – 0xFFC0 0920  SPORT1_RCR2 – 0xFFC0 0924
SPORT2_RCR1 – 0xFFC0 2520  SPORT2_RCR2 – 0xFFC0 2524
SPORT3_RCR1 – 0xFFC0 2620  SPORT3_RCR2 – 0xFFC0 2624
Serial Port Controllers

**SPORTx Receive Configuration 1 Register (SPORTx_RCR1)**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>RCKFE (Clock Falling Edge Select)</td>
<td>0 - Drive internal frame sync on rising edge of RSCLK. Sample data and external frame sync with falling edge of RSCLK. 1 - Drive internal frame sync on falling edge of RSCLK. Sample data and external frame sync with rising edge of RSCLK.</td>
</tr>
<tr>
<td>14</td>
<td>LRFS (Low Receive Frame Sync Select)</td>
<td>0 - Active high RFS 1 - Active low RFS</td>
</tr>
<tr>
<td>13</td>
<td>RLSBIT (Receive Bit Order)</td>
<td>0 - Early frame syncs 1 - Late frame syncs</td>
</tr>
<tr>
<td>12</td>
<td>LARFS (Late Receive Frame Sync)</td>
<td>0 - Early frame syncs 1 - Late frame syncs</td>
</tr>
<tr>
<td>11</td>
<td>RFSR (Receive Frame Sync Required Select)</td>
<td>0 - Does not require RFS for every data word 1 - Requires RFS for every data word</td>
</tr>
<tr>
<td>10</td>
<td>RSPEN (Receive Enable)</td>
<td>0 - Receive disabled 1 - Receive enabled</td>
</tr>
<tr>
<td>9</td>
<td>IRFS (Internal Receive Frame Sync Select)</td>
<td>0 - External receive clock selected 1 - Internal receive clock selected</td>
</tr>
<tr>
<td>8</td>
<td>RDTYPE[1:0] (Data Formatting Type Select)</td>
<td>00 - Zero-fill 01 - Sign-extend 10 - Compand using μ-law 11 - Compand using A-law</td>
</tr>
<tr>
<td>7</td>
<td>LRFS (Low Receive Frame Sync Select)</td>
<td>0 - Active high RFS 1 - Active low RFS</td>
</tr>
<tr>
<td>6</td>
<td>LARFS (Late Receive Frame Sync)</td>
<td>0 - Early frame syncs 1 - Late frame syncs</td>
</tr>
<tr>
<td>5</td>
<td>RFSR (Receive Frame Sync Required Select)</td>
<td>0 - Does not require RFS for every data word 1 - Requires RFS for every data word</td>
</tr>
<tr>
<td>4</td>
<td>RSPEN (Receive Enable)</td>
<td>0 - Receive disabled 1 - Receive enabled</td>
</tr>
<tr>
<td>3</td>
<td>IRFS (Internal Receive Frame Sync Select)</td>
<td>0 - External receive clock selected 1 - Internal receive clock selected</td>
</tr>
<tr>
<td>2</td>
<td>RDTYPE[1:0] (Data Formatting Type Select)</td>
<td>00 - Zero-fill 01 - Sign-extend 10 - Compand using μ-law 11 - Compand using A-law</td>
</tr>
<tr>
<td>1</td>
<td>LRFS (Low Receive Frame Sync Select)</td>
<td>0 - Active high RFS 1 - Active low RFS</td>
</tr>
<tr>
<td>0</td>
<td>LARFS (Late Receive Frame Sync)</td>
<td>0 - Early frame syncs 1 - Late frame syncs</td>
</tr>
</tbody>
</table>

Reset = 0x0000

**SPORTx Receive Configuration 2 Register (SPORTx_RCR2)**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>RRFST (Left/Right Order)</td>
<td>0 - Left stereo channel first 1 - Right stereo channel first</td>
</tr>
<tr>
<td>14</td>
<td>RSFSE (Receive Stereo Frame Sync Enable)</td>
<td>0 - Normal mode 1 - Frame sync becomes L/R clock</td>
</tr>
<tr>
<td>13</td>
<td>SLEN[4:0] (SPORT Word Length)</td>
<td>000000 - Illegal value 00001 - Illegal value Serial word length is value in this field plus 1</td>
</tr>
<tr>
<td>12</td>
<td>RXSE (RxSEC Enable)</td>
<td>0 - Secondary side disabled 1 - Secondary side enabled</td>
</tr>
</tbody>
</table>

Reset = 0x0000

Figure 13-6. SPORTx Receive Configuration 1 Register

Figure 13-7. SPORTx Receive Configuration 2 Register
SPORT Receive Configuration (SPORTx_RCR1, SPORTx_RCR2) Registers

Additional information for the SPORTx_RCR1 and SPORTx_RCR2 receive configuration register bits:

- **Receive Enable.** (RSPEN). This bit selects whether the SPORT is enabled to receive (if set) or disabled (if cleared). Setting the RSPEN bit turns on the SPORT and causes it to sample data from the data receive pins as well as the receive bit clock and receive frame sync pins if so programmed.

  Setting RSPEN enables the SPORTx receiver, which can generate a SPORTx RX interrupt. For this reason, the code should initialize the ISR and the DMA control registers, and should be ready to service RX interrupts before setting RSPEN. Setting RSPEN also generates DMA requests if DMA is enabled and data is received. Set all DMA control registers before setting RSPEN. Clearing RSPEN causes the SPORT to stop receiving data; it also shuts down the internal SPORT receive circuitry. In low power applications, battery life can be extended by clearing RSPEN whenever the SPORT is not in use.

  All SPORT control registers should be programmed before RSPEN is set. Typical SPORT initialization code first writes all control registers, including DMA control if applicable. The last step in the code is to write SPORTx_RCR1 with all of the necessary bits, including RSPEN.

- **Internal Receive Clock Select.** (IRCLK). This bit selects the internal receive clock (if set) or external receive clock (if cleared). The RCLK-DIV MMR value is not used when an external clock is selected.

- **Data Formatting Type Select.** (RDTYPE). The two RDTYPE bits specify one of four data formats used for single and multichannel operation.
Serial Port Controllers

- **Bit Order Select.** (RLSBIT). The RLSBIT bit selects the bit order of the data words received over the SPORTs.

- **Serial Word Length Select.** (SLEN). The serial word length (the number of bits in each word received over the SPORTs) is calculated by adding 1 to the value of the SLEN field. The SLEN field can be set to a value of 2 to 31; 0 and 1 are illegal values for this field.

  The frame sync signal is controlled by the SPORTx_TFSDIV and SPORTx_RFSDIV registers, not by SLEN. To produce a frame sync pulse on each byte or word transmitted, the proper frame sync divider must be programmed into the frame sync divider register; setting SLEN to 7 does not produce a frame sync pulse on each byte transmitted.

- **Internal Receive Frame Sync Select.** (IRFS). This bit selects whether the SPORT uses an internal RFS (if set) or an external RFS (if cleared).

- **Receive Frame Sync Required Select.** (RFSR). This bit selects whether the SPORT requires (if set) or does not require (if cleared) a receive frame sync for every data word.

- **Low Receive Frame Sync Select.** (LRFS). This bit selects an active low RFS (if set) or active high RFS (if cleared).

- **Late Receive Frame Sync.** (LARFS). This bit configures late frame syncs (if set) or early frame syncs (if cleared).
Data Word Formats

- **Clock Drive/Sample Edge Select.** (RCKFE). This bit selects which edge of the RSCLK clock signal the SPORT uses for sampling data, for sampling externally generated frame syncs, and for driving internally generated frame syncs. If set, internally generated frame syncs are driven on the falling edge, and data and externally generated frame syncs are sampled on the rising edge. If cleared, internally generated frame syncs are driven on the rising edge, and data and externally generated frame syncs are sampled on the falling edge.

- **RxSec Enable.** (RXSE). This bit enables the receive secondary side of the serial port (if set).

- **Stereo Serial Enable.** (RSFSE). This bit enables the stereo serial operating mode of the serial port (if set). By default this bit is cleared, enabling normal clocking and frame sync.

- **Left/Right Order.** (RRFST). If this bit is set, the right channel is received first in stereo serial operating mode. By default this bit is cleared, and the left channel is received first.

Data Word Formats

The format of the data words transferred over the SPORTs is configured by the combination of transmit SLEN and receive SLEN; RDTYPE; TDTYPE; RLSBIT; and TLSBIT bits of the SPORTx_TCR1, SPORTx_TCR2, SPORTx_RCR1, and SPORTx_RCR2 registers.
SPORT Transmit Data (SPORTx_TX) Register

The SPORTx transmit data register (SPORTx_TX) is a write-only register. Reads produce a peripheral access bus (PAB) error. Writes to this register cause writes into the transmitter FIFO. The 16-bit wide FIFO is 8 deep for word length <= 16 and 4 deep for word length > 16. The FIFO is common to both primary and secondary data and stores data for both. Data ordering in the FIFO is shown in the Figure 13-8.

Figure 13-8. SPORT Transmit FIFO Data Ordering

It is important to keep the interleaving of primary and secondary data in the FIFO as shown. This means that PAB/DMA writes to the FIFO must follow an order of primary first, and then secondary, if secondary is enabled. DAB/PAB writes must match their size to the data word length. For word length up to and including 16 bits, use a 16-bit write. Use a 32-bit write for word length greater than 16 bits.
SPORT Transmit Data (SPORTx_TX) Register

When transmit is enabled, data from the FIFO is assembled in the TX hold register based on TXSE and SLEN, and then shifted into the primary and secondary shift registers. From here, the data is shifted out serially on the DTxPRI and DTxSEC pins. See Figure 13-9.

The SPORT TX interrupt is asserted when TSPEN = 1 and the TX FIFO has room for additional words. This interrupt does not occur if SPORT DMA is enabled. For DMA operation, see Chapter 9, “Direct Memory Access”.

The transmit underflow register bit (TUVF) is set in the SPORT status register when a transmit frame sync occurs and no new data has been loaded into the serial shift register. In multichannel mode (MCM), TUVF is set whenever the serial shift register is not loaded, and transmission begins on the current enabled channel. The TUVF status bit is a sticky write-1-to-clear (W1C) bit and is also cleared by disabling the serial port (writing TSPEN = 0).

If software causes the core processor to attempt a write to a full TX FIFO with a SPORTx_TX write, the new data is lost and no overwrites occur to data in the FIFO. The TOVF status bit is set and a SPORT error interrupt is asserted. The TOVF bit is a sticky bit; it is only cleared by disabling the SPORT TX. To find out whether the core processor can access the SPORTx_TX register without causing this type of error, read the register’s status first. The TXF bit in the SPORT status register is 0 if space is available for another word in the FIFO.

The TXF and TOVF status bits in the SPORTx status register are updated upon writes from the core processor, even when the SPORT is disabled.

The addresses for these SPORT registers are:

<table>
<thead>
<tr>
<th>SPORTk_TX</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPORT0_TX</td>
<td>0xFFC0 0810</td>
</tr>
<tr>
<td>SPORT1_TX</td>
<td>0xFFC0 0910</td>
</tr>
<tr>
<td>SPORT2_TX</td>
<td>0xFFC0 2510</td>
</tr>
<tr>
<td>SPORT3_TX</td>
<td>0xFFC0 2610</td>
</tr>
</tbody>
</table>
SPORTx Receive Data (SPORTx_RX) Register

The SPORTx receive data register (SPORTx_RX) is a read-only register. See Figure 13-10. Writes produce a PAB error. The same location is read for both primary and secondary data. Reading from this register space causes reading of the receive FIFO. This 16-bit FIFO is 8 deep for receive word length <= 16 and 4 deep for length > 16 bits. The FIFO is shared by both primary and secondary receive data. The order for reading using PAB/DMA reads is important since data is stored in differently depending on the setting of the SLEN and RXSE configuration bits.
When reading from the FIFO for both primary and secondary data, read primary first, followed by secondary. DAB/PAB reads must match their size to the data word length. For word length up to and including 16 bits, use a 16-bit read. Use a 32-bit read for word length greater than 16 bits.

When receiving is enabled, data from the DRxPRI pin is loaded into the RX primary shift register, while data from the DRxSEC pin is loaded into the RX secondary shift register. At transfer completion of a word, data is shifted into the RX hold registers for primary and secondary data, respectively. Data from the hold registers is moved into the FIFO based on RXSE and SLEN.

The SPORT RX interrupt is generated when RSPEN = 1 and the RX FIFO has received words in it. When the core processor has read all the words in the FIFO, the RX interrupt is cleared. The SPORT RX interrupt is set only if SPORT RX DMA is disabled; otherwise, the FIFO is read by DMA reads.

If the program causes the core processor to attempt a read from an empty RX FIFO, old data is read, the RUVF flag is set in the SPORTx_STAT register, and the SPORT error interrupt is asserted. The RUVF bit is a sticky bit and is cleared only when the SPORT is disabled. To determine if the core can
access the RX registers without causing this error, first read the RX FIFO status (RXNE in the SPORTx status register). The RUVF status bit is updated even when the SPORT is disabled.

The ROVF status bit is set in the SPORTx_STAT register when a new word is assembled in the RX shift register and the RX hold register has not moved the data to the FIFO. The previously written word in the hold register is overwritten. The ROVF bit is a sticky bit; it is only cleared by disabling the SPORT receiver.

The addresses for these SPORT registers are:

- SPORT0_RX – 0xFFC0 0818
- SPORT1_RX – 0xFFC0 0918
- SPORT2_RX – 0xFFC0 2518
- SPORT3_RX – 0xFFC0 2618
SPORT Status (SPORTx_STAT) Register

Data storage and data ordering in the FIFO is shown in Figure 13-11.

![Figure 13-11. SPORT Receive FIFO Data Ordering](image)

The SPORT status register (SPORTx_STAT) is used to determine if the access to a SPORT RX or TX FIFO can be made by determining their full or empty status. See Figure 13-12.

The TXF bit in the SPORT status register indicates whether there is room in the TX FIFO. The RXNE status bit indicates whether there are words in the RX FIFO. The TXHRE bit indicates if the TX hold register is empty.
The transmit underflow status bit (TUVF) is set whenever the TFS signal occurs (from either an external or internal source) while the TX shift register is empty. The internally generated TFS may be suppressed whenever SPORTx_TX is empty by clearing the DITFS control bit in the SPORT configuration register. The TUVF status bit is a sticky write-1-to-clear (W1C) bit and is also cleared by disabling the serial port (writing TSPEN = 0).

**SPORTx Status Register (SPORTx_STAT)**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>RXNE (Receive FIFO Not Empty status)</td>
<td>0 - Empty, 1 - Data present in FIFO</td>
</tr>
<tr>
<td>14</td>
<td>TXHRE (Transmit Hold register Empty)</td>
<td>0 - Not empty, 1 - Empty</td>
</tr>
<tr>
<td>13</td>
<td>ROVF (Sticky Receive Overflow status) - W1C</td>
<td>0 - Disabled, 1 - Enabled</td>
</tr>
<tr>
<td>12</td>
<td>TUVF (Sticky Transmit Underflow status) - W1C</td>
<td>0 - Disabled, 1 - Enabled</td>
</tr>
<tr>
<td>11</td>
<td>TOVF (Sticky Transmit Overflow status) - W1C</td>
<td>0 - Disabled, 1 - Enabled</td>
</tr>
<tr>
<td>10</td>
<td>TXF (Transmit FIFO Full status)</td>
<td>0 - Not full, 1 - Full</td>
</tr>
</tbody>
</table>

Figure 13-12. SPORTx Status Register

For continuous transmission (TFSR = 0), TUVF is set at the end of a transmitted word if no new word is available in the TX hold register.

The TOVF bit is set when a word is written to the TX FIFO when it is full. It is a sticky W1C bit and is also cleared by writing TSPEN = 0. Both TXF and TOVF are updated even when the SPORT is disabled.

When the SPORT RX hold register is full, and a new receive word is received in the shift register, the receive overflow status bit (ROVF) is set in the SPORT status register. It is a sticky W1C bit and is also cleared by disabling the serial port (writing RSPEN = 0).
SPORT Status (SPORTx_STAT) Register

The RUVF bit is set when a read is attempted from the RX FIFO and it is empty. It is a sticky W1C bit and is also cleared by writing RSPEN = 0. The RUVF bit is updated even when the SPORT is disabled.

The addresses for these SPORT registers are:

SPORT0_STAT – 0xFFC0 0830
SPORT1_STAT – 0xFFC0 0930
SPORT2_STAT – 0xFFC0 2530
SPORT3_STAT – 0xFFC0 2630

SPORT RX, TX, and Error Interrupts

The SPORT RX interrupt is asserted when RSPEN is enabled and any words are present in the RX FIFO. If RX DMA is enabled, the SPORT RX interrupt is turned off and DMA services the RX FIFO.

The SPORT TX interrupt is asserted when TSPEN is enabled and the TX FIFO has room for words. If TX DMA is enabled, the SPORT TX interrupt is turned off and DMA services the TX FIFO.

The SPORT error interrupt is asserted when any of the sticky status bits (ROVF, RUVF, TOVF, TUVF) are set. The ROVF and RUVF bits are cleared by writing 0 to RSPEN. The TOVF and TUVF bits are cleared by writing 0 to TSPEN.
PAB Errors

The SPORT generates a PAB error for illegal register read or write operations. Examples include:

- Reading a write-only register (for example, SPORTx_TX)
- Writing a read-only register (for example, SPORTx_RX)
- Writing or reading a register with the wrong size (for example, 32-bit read of a 16-bit register)
- Accessing reserved register locations

SPORT Transmit Serial Clock Divider (SPORTx_TCLKDIV, SPORTx_RCLKDIV) Registers

The frequency of an internally generated clock is a function of the system clock frequency (as seen at the SCLK pin) and the value of the 16-bit serial clock divide modulus registers (the SPORTx transmit serial clock divider register, SPORTx_TCLKDIV, and the SPORTx receive serial clock divider register, SPORTx_RCLKDIV).

Figure 13-13. SPORTx Transmit Serial Clock Divider Register
The addresses for these SPORT registers are:

- SPORT0_TCLKDIV – 0xFFC0 0808
- SPORT0_RCLKDIV – 0xFFC0 0828
- SPORT1_TCLKDIV – 0xFFC0 0908
- SPORT1_RCLKDIV – 0xFFC0 0928
- SPORT2_TCLKDIV – 0xFFC0 2508
- SPORT2_RCLKDIV – 0xFFC0 2528
- SPORT3_TCLKDIV – 0xFFC0 2608
- SPORT3_RCLKDIV – 0xFFC0 2628

The 16-bit SPORTx transmit frame sync divider register (SPORTx_TFSDIV) and the SPORTx receive frame sync divider register (SPORTx_RFSDIV) specify how many transmit or receive clock cycles are counted before generating a TFSx or RFSx pulse when the frame sync is internally generated. In this way, a frame sync can be used to initiate periodic transfers. The counting of serial clock cycles applies to either internally or externally generated serial clocks.
The addresses for these SPORT registers are:

SPORT0_TFSDIV – 0xFFC0 080C  
SPORT0_RFSDIV – 0xFFC0 082C

SPORT1_TFSDIV – 0xFFC0 090C  
SPORT1_RFSDIV – 0xFFC0 092C

SPORT2_TFSDIV – 0xFFC0 250C  
SPORT2_RFSDIV – 0xFFC0 252C

SPORT3_TFSDIV – 0xFFC0 260C  
SPORT3_RFSDIV – 0xFFC0 262C

**SPORTx Transmit Frame Sync Divider Register (SPORTx_TFSDIV)**

```
  15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Reset = 0x0000

Frame Sync Divider[15:0]  
Number of transmit clock cycles counted before generating TFS pulse

**Figure 13-15. SPORTx Transmit Frame Sync Divider Register**

**SPORTx Receive Frame Sync Divider Register (SPORTx_RFSDIV)**

```
  15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Reset = 0x0000

Frame Sync Divider[15:0]  
Number of receive clock cycles counted before generating RFS pulse

**Figure 13-16. SPORTx Receive Frame Sync Divider Register**
Clock and Frame Sync Frequencies

The maximum serial clock frequency (for either an internal source or an external source) is $SCLK/2$. The frequency of an internally generated clock is a function of the system clock frequency ($SCLK$) and the value of the 16-bit serial clock divide modulus registers, $SPORTx_TCLKDIV$ and $SPORTx_RCLKDIV$.

$SPORTx_TCLK$ frequency =
$(SCLK$ frequency$/2 \times (SPORTx_TCLKDIV + 1))$

$SPORTx_RCLK$ frequency =
$(SCLK$ frequency$/2 \times (SPORTx_RCLKDIV + 1))$

If the value of $SPORTx_TCLKDIV$ or $SPORTx_RCLKDIV$ is changed while the internal serial clock is enabled, the change in $TSCLK$ or $RSCLK$ frequency takes effect at the start of the drive edge of $TSCLKx$ or $RSCLKx$ that follows the next leading edge of $TFSx$ or $RFSx$.

When an internal frame sync is selected ($ITFS = 1$ in the $SPORTx_TCR1$ register or $IRFS = 1$ in the $SPORTx_RCR1$ register) and frame syncs are not required, the first frame sync does not update the clock divider if the value in $SPORTx_TCLKDIV$ or $SPORTx_RCLKDIV$ has changed. The second frame sync will cause the update.

The $SPORTx_TFSDIV$ and $SPORTx_RFSDIV$ registers specify the number of transmit or receive clock cycles that are counted before generating a $TFSx$ or $RFSx$ pulse (when the frame sync is internally generated). This enables a frame sync to initiate periodic transfers. The counting of serial clock cycles applies to either internally or externally generated serial clocks.

The formula for the number of cycles between frame sync pulses is:

# of transmit serial clocks between frame sync assertions = $TFSDIV + 1$
# of receive serial clocks between frame sync assertions = $RFSDIV + 1$
Use the following equations to determine the correct value of $\text{TFSDIV}$ or $\text{RFSDIV}$, given the serial clock frequency and desired frame sync frequency:

\[
\text{SPORTxTFS frequency} = \frac{\text{TSCLKx frequency}}{\text{SPORTx}_{\text{TFSDIV}} + 1}
\]

\[
\text{SPORTxRFS frequency} = \frac{\text{RSCLKx frequency}}{\text{SPORTx}_{\text{RFSDIV}} + 1}
\]

The frame sync would thus be continuously active (for transmit if $\text{TFSDIV} = 0$ or for receive if $\text{RFSDIV} = 0$). However, the value of $\text{TFSDIV}$ (or $\text{RFSDIV}$) should not be less than the serial word length minus 1 (the value of the $\text{SLEN}$ field in $\text{SPORTx}_{\text{TCR2}}$ or $\text{SPORTx}_{\text{RCR2}}$). A smaller value could cause an external device to abort the current operation or have other unpredictable results. If a SPORT is not being used, the $\text{TFSDIV}$ (or $\text{RFSDIV}$) divisor can be used as a counter for dividing an external clock or for generating a periodic pulse or periodic interrupt. The SPORT must be enabled for this mode of operation to work.

**Maximum Clock Rate Restrictions**

Externally generated late transmit frame syncs also experience a delay from arrival to data output, and this can limit the maximum serial clock speed. See *ADSP-BF538/ADSP-BF538F Embedded Processor Data Sheet* for exact timing specifications.

**Frame Sync & Clock Example**

The following code fragment is a brief example of setting up the clocks and frame sync.

\[
\begin{align*}
\text{r0} &= 0x00FF; \\
\text{p0.l} &= \text{SPORT0}_{\text{RFSDIV}} \& 0xFFFF; \\
\text{p0.h} &= (\text{SPORT0}_{\text{RFSDIV}} >> 16) \& 0xFFFF; \\
\text{w}[\text{p0}] &= \text{r0.l}; \text{ssync}; \\
\text{p0.l} &= \text{SPORT0}_{\text{TFSDIV}} \& 0xFFFF; \\
\text{w}[\text{p0}] &= \text{r0.l}; \text{ssync};
\end{align*}
\]
Word Length

Each SPORT channel (transmit and receive) independently handles word lengths of 3 to 32 bits. The data is right-justified in the SPORT data registers if it is fewer than 32 bits long, residing in the LSB positions. The value of the serial word length (SLEN) field in the SPORTx_TCR2 and SPORTx_RCR2 registers of each SPORT determines the word length according to this formula:

$$\text{Serial Word Length} = \text{SLEN} + 1$$

- The SLEN value should not be set to 0 or 1; values from 2 to 31 are allowed. Continuous operation (when the last bit of the current word is immediately followed by the first bit of the next word) is restricted to word sizes of 4 or longer (so $\text{SLEN} \geq 3$).

Bit Order

Bit order determines whether the serial word is transmitted MSB first or LSB first. Bit order is selected by the RLSBIT and TLSBIT bits in the SPORTx_RCR1 and SPORTx_TCR1 registers. When RLSBIT (or TLSBIT) = 0, serial words are received (or transmitted) MSB first. When RLSBIT (or TLSBIT) = 1, serial words are received (or transmitted) LSB first.

Data Type

The TDTYPE field of the SPORTx_TCR1 register and the RDTYPE field of the SPORTx_RCR1 register specify one of four data formats for both single and multichannel operation. See Table 13-2.
These formats are applied to serial data words loaded into the \texttt{SPORTx\_RX} and \texttt{SPORTx\_TX} buffers. \texttt{SPORTx\_TX} data words are not actually zero-filled or sign-extended, because only the significant bits are transmitted.

### Companding

Companding (a contraction of COMpressing and exPANDing) is the process of logarithmically encoding and decoding data to minimize the number of bits that must be sent. The SPORTs support the two most widely used companding algorithms, \( \mu \)-law and A-law. The processor compands data according to the CCITT G.711 specification. The type of companding can be selected independently for each SPORT.

When companding is enabled, valid data in the \texttt{SPORTx\_RX} register is the right-justified, expanded value of the eight LSBs received and sign-extended to 16 bits. A write to \texttt{SPORTx\_TX} causes the 16-bit value to be compressed to eight LSBs (sign-extended to the width of the transmit word) and written to the internal transmit register. Although the companding standards support only 13-bit (A-law) or 14-bit (\( \mu \)-law) maximum word lengths, up to 16-bit word lengths can be used. If the magnitude of the word value is greater than the maximum allowed, the value is automatically compressed to the maximum positive or negative value.

Lengths greater than 16 bits are not supported for companding operation.

<table>
<thead>
<tr>
<th>TDTYPE or RDTYPE</th>
<th>SPORTx_TCR1 Data Formatting</th>
<th>SPORTx_RCR1 Data Formatting</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Normal operation</td>
<td>Zero-fill</td>
</tr>
<tr>
<td>01</td>
<td>Reserved</td>
<td>Sign-extend</td>
</tr>
<tr>
<td>10</td>
<td>Compand using ( \mu )-law</td>
<td>Compand using ( \mu )-law</td>
</tr>
<tr>
<td>11</td>
<td>Compand using A-law</td>
<td>Compand using A-law</td>
</tr>
</tbody>
</table>
Clock Signal Options

Each SPORT has a transmit clock signal (TSCLKx) and a receive clock signal (RSCLKx). The clock signals are configured by the TCKFE and RCKFE bits of the SPORTx_TCR1 and SPORTx_RCR1 registers. Serial clock frequency is configured in the SPORTx_TCLKDIV and SPORTx_RCLKDIV registers.

The receive clock pin may be tied to the transmit clock if a single clock is desired for both receive and transmit.

Both transmit and receive clocks can be independently generated internally or input from an external source. The ITCLK bit of the SPORTx_TCR1 configuration register and the IRCLK bit in the SPORTx_RCR1 Configuration register determines the clock source.

When IRCLK or ITCLK = 1, the clock signal is generated internally by the core, and the TSCLKx or RSCLKx pin is an output. The clock frequency is determined by the value of the serial clock divisor in the SPORTx_RCLKDIV register.

When IRCLK or ITCLK = 0, the clock signal is accepted as an input on the TSCLKx or RSCLKx pins, and the serial clock divisors in the SPORTx_TCLKDIV/SPORTx_RCLKDIV registers are ignored. The externally generated serial clocks do not need to be synchronous with the core system clock or with each other. The core system clock must have a higher frequency than RSCLKx and TSCLKx.

When the SPORT uses external clocks, it must be enabled for a minimal number of stable clock pulses before the first active frame sync is sampled. Failure to allow for these clocks may result in a SPORT malfunction. See the processor data sheet for details.

The first internal frame sync will occur one frame sync delay after the SPORTs are ready. External frame syncs can occur as soon as the SPORT is ready.
Frame Sync Options

Framing signals indicate the beginning of each serial word transfer. The framing signals for each SPORT are $TFS_x$ (transmit frame sync) and $RFS_x$ (receive frame sync). A variety of framing options are available; these options are configured in the SPORT configuration registers ($SPORT_x_TCR1$, $SPORT_x_TCR2$, $SPORT_x_RCR1$ and $SPORT_x_RCR2$). The $TFS_x$ and $RFS_x$ signals of a SPORT are independent and are separately configured in the control registers.

Framed Versus Unframed

The use of multiple frame sync signals is optional in SPORT communications. The $TFSR$ (transmit frame sync required select) and $RFSR$ (receive frame sync required select) control bits determine whether frame sync signals are required. These bits are located in the $SPORT_x_TCR1$ and $SPORT_x_RCR1$ registers.

When $TFSR = 1$ or $RFSR = 1$, a frame sync signal is required for every data word. To allow continuous transmitting by the SPORT, each new data word must be loaded into the $SPORT_x_TX$ hold register before the previous word is shifted out and transmitted.

When $TFSR = 0$ or $RFSR = 0$, the corresponding frame sync signal is not required. A single frame sync is needed to initiate communications but is ignored after the first bit is transferred. Data words are then transferred continuously, unframed.

⚠️ With frame syncs not required, interrupt or DMA requests may not be serviced frequently enough to guarantee continuous unframed data flow. Monitor status bits or check for a SPORT Error interrupt to detect underflow or overflow of data.
Frame Sync Options

Figure 13-17 illustrates framed serial transfers, which have these characteristics:

- **TFSR and RFSR bits in the SPORTx_TCR1 and SPORTx_RCR1 registers** determine framed or unframed mode.
- Framed mode requires a framing signal for every word. Unframed mode ignores a framing signal after the first word.
- Unframed mode is appropriate for continuous reception.
- Active low or active high frame syncs are selected with the LTFS and LRFS bits of the SPORTx_TCR1 and SPORTx_RCR1 registers.

See “Timing Examples” on page 13-67 for more timing examples.

Internal Versus External Frame Syncs

Both transmit and receive frame syncs can be independently generated internally or can be input from an external source. The ITFS and IRFS bits of the SPORTx_TCR1 and SPORTx_RCR1 registers determine the frame sync source.
When $ITFS = 1$ or $IRFS = 1$, the corresponding frame sync signal is generated internally by the SPORT, and the $TFS_x$ pin or $RFS_x$ pin is an output. The frequency of the frame sync signal is determined by the value of the frame sync divisor in the $SPORTx_TFSDIV$ or $SPORTx_RFSDIV$ register.

When $ITFS = 0$ or $IRFS = 0$, the corresponding frame sync signal is accepted as an input on the $TFS_x$ pin or $RFS_x$ pin, and the frame sync divisors in the $SPORTx_TFSDIV$/$SPORTx_RFSDIV$ registers are ignored.

All of the frame sync options are available whether the signal is generated internally or externally.

**Active Low Versus Active High Frame Syncs**

Frame sync signals may be either active high or active low (in other words, inverted). The $LTFS$ and $LRFS$ bits of the $SPORTx_TCR1$ and $SPORTx_RCR1$ registers determine frame sync logic levels:

- When $LTFS = 0$ or $LRFS = 0$, the corresponding frame sync signal is active high.
- When $LTFS = 1$ or $LRFS = 1$, the corresponding frame sync signal is active low.

Active high frame syncs are the default. The $LTFS$ and $LRFS$ bits are initialized to 0 after a processor reset.

**Sampling Edge for Data and Frame Syncs**

Data and frame syncs can be sampled on either the rising or falling edges of the SPORT clock signals. The $TCKFE$ and $RCKFE$ bits of the $SPORTx_TCR1$ and $SPORTx_RCR1$ registers select the driving and sampling edges of the serial data and frame syncs.
Frame Sync Options

For the SPORT transmitter, setting \( TCKFE = 1 \) in the \( \text{SPORTx_TCR1} \) register selects the falling edge of \( TSCLKx \) to drive data and internally generated frame syncs and selects the rising edge of \( TSCLKx \) to sample externally generated frame syncs. Setting \( TCKFE = 0 \) selects the rising edge of \( TSCLKx \) to drive data and internally generated frame syncs and selects the falling edge of \( TSCLKx \) to sample externally generated frame syncs.

For the SPORT receiver, setting \( RCKFE = 1 \) in the \( \text{SPORTx_RCR1} \) register selects the falling edge of \( RSCLKx \) to drive internally generated frame syncs and selects the rising edge of \( RSCLKx \) to sample data and externally generated frame syncs. Setting \( RCKFE = 0 \) selects the rising edge of \( RSCLKx \) to drive internally generated frame syncs and selects the falling edge of \( RSCLKx \) to sample data and externally generated frame syncs.

\( \Box \) Note externally generated data and frame sync signals should change state on the opposite edge than that selected for sampling. For example, for an externally generated frame sync to be sampled on the rising edge of the clock (\( TCKFE = 1 \) in the \( \text{SPORTx_TCR1} \) register), the frame sync must be driven on the falling edge of the clock.

The transmit and receive functions of two SPORTs connected together should always select the same value for \( TCKFE \) in the transmitter and \( RCKFE \) in the receiver, so that the transmitter drives the data on one edge and the receiver samples the data on the opposite edge.

In Figure 13-18, \( TCKFE = RCKFE = 0 \) and transmit and receive are connected together to share the same clock and frame syncs.
In Figure 13-19, $TCKFE = RCKFE = 1$ and transmit and receive are connected together to share the same clock and frame syncs.

Figure 13-18. Example of $TCKFE = RCKFE = 0$, Transmit and Receive Connected

Figure 13-19. Example of $TCKFE = RCKFE = 1$, Transmit and Receive Connected

**Early Versus Late Frame Syncs**

*(Normal Versus Alternate Timing)*

Frame sync signals can occur during the first bit of each data word (late) or during the serial clock cycle immediately preceding the first bit (early). The $LATFS$ and $LARFS$ bits of the $SPORTx_TCR1$ and $SPORTx_RCR1$ registers configure this option.
Frame Sync Options

When $\text{LATFS} = 0$ or $\text{LARFS} = 0$, early frame syncs are configured; this is the normal mode of operation. In this mode, the first bit of the transmit data word is available and the first bit of the receive data word is sampled in the serial clock cycle after the frame sync is asserted, and the frame sync is not checked again until the entire word has been transmitted or received. In multichannel operation, this corresponds to the case when multichannel frame delay is 1.

If data transmission is continuous in early framing mode (in other words, the last bit of each word is immediately followed by the first bit of the next word), then the frame sync signal occurs during the last bit of each word. Internally generated frame syncs are asserted for one clock cycle in early framing mode. Continuous operation is restricted to word sizes of 4 or longer ($\text{SLEN} \geq 3$).

When $\text{LATFS} = 1$ or $\text{LARFS} = 1$, late frame syncs are configured; this is the alternate mode of operation. In this mode, the first bit of the transmit data word is available and the first bit of the receive data word is sampled in the same serial clock cycle that the frame sync is asserted. In multichannel operation, this is the case when frame delay is 0. Receive data bits are sampled by serial clock edges, but the frame sync signal is only checked during the first bit of each word. Internally generated frame syncs remain asserted for the entire length of the data word in late framing mode. Externally generated frame syncs are only checked during the first bit.

Figure 13-20 illustrates the two modes of frame signal timing. In summary:

- For the $\text{LATFS}$ or $\text{LARFS}$ bits of the $\text{SPORTx_TCR1}$ or $\text{SPORTx_RCR1}$ registers: $\text{LATFS} = 0$ or $\text{LARFS} = 0$ for early frame syncs, $\text{LATFS} = 1$ or $\text{LARFS} = 1$ for late frame syncs.

- For early framing, the frame sync precedes data by one cycle. For late framing, the frame sync is checked on the first bit only.
• Data is transmitted MSB first ($\text{TLSBIT} = 0$ or $\text{RLSBIT} = 0$) or LSB first ($\text{TLSBIT} = 1$ or $\text{RLSBIT} = 1$).

• Frame sync and clock are generated internally or externally.

See “Timing Examples” on page 13-67 for more examples.

![Figure 13-20. Normal Versus Alternate Framing](image)

**Data Independent Transmit Frame Sync**

Normally the internally generated transmit frame sync signal ($\text{TFSx}$) is output only when the $\text{SPORTx\_TX}$ buffer has data ready to transmit. The data-independent transmit frame sync select bit ($\text{DITFS}$) allows the continuous generation of the $\text{TFSx}$ signal, with or without new data. The $\text{DITFS}$ bit of the $\text{SPORTx\_TCR1}$ register configures this option.

When $\text{DITFS} = 0$, the internally generated $\text{TFSx}$ is only output when a new data word has been loaded into the $\text{SPORTx\_TX}$ buffer. The next $\text{TFSx}$ is generated once data is loaded into $\text{SPORTx\_TX}$. This mode of operation allows data to be transmitted only when it is available.

When $\text{DITFS} = 1$, the internally generated $\text{TFSx}$ is output at its programmed interval regardless of whether new data is available in the $\text{SPORTx\_TX}$ buffer. Whatever data is present in $\text{SPORTx\_TX}$ is transmitted again with each assertion of $\text{TFSx}$. The $\text{TUVF}$ (transmit underflow status)
bit in the \texttt{SPORTx_STAT} register is set when this occurs and old data is retransmitted. The \texttt{TUVF} status bit is also set if the \texttt{SPORTx_TX} buffer does not have new data when an externally generated \texttt{TFSx} occurs. Note that in this mode of operation, data is transmitted only at specified times.

If the internally generated \texttt{TFSx} is used, a single write to the \texttt{SPORTx_TX} data register is required to start the transfer.

## Moving Data Between SPORTs and Memory

Transmit and receive data can be transferred between the SPORTs and on-chip memory in one of two ways: with single word transfers or with DMA block transfers.

If no SPORT DMA channel is enabled, the SPORT generates an interrupt every time it has received a data word or needs a data word to transmit. SPORT DMA provides a mechanism for receiving or transmitting an entire block or multiple blocks of serial data before the interrupt is generated. The SPORT’s DMA controller handles the DMA transfer, allowing the processor core to continue running until the entire block of data is transmitted or received. Interrupt service routines (ISRs) can then operate on the block of data rather than on single words, significantly reducing overhead.

For information about DMA, see Chapter 9, “Direct Memory Access”.

## Stereo Serial Operation

Several stereo serial modes can be supported by the SPORT, including the popular I\textsuperscript{2}S format. To use these modes, set bits in the \texttt{SPORT_RCR2} or \texttt{SPORT_TCR2} registers. Setting \texttt{RSFSE} or \texttt{TSFSE} in \texttt{SPORT_RCR2} or \texttt{SPORT_TCR2} changes the operation of the frame sync pin to a left/right clock as
required for I²S and left-justified stereo serial data. Setting this bit enables the SPORT to generate or accept the special LRCLK-style frame sync. All other SPORT control bits remain in effect and should be set appropriately. Figure 13-21 and Figure 13-22 show timing diagrams for stereo serial mode operation.

Figure 13-21. SPORT Stereo Serial Modes, Transmit

NOTES:
1. DSP MODE DOES NOT IDENTIFY CHANNEL. 
2. TFSx NORMALLY OPERATES AT fs EXCEPT FOR DSP MODE WHICH IS 2 x fs. 
3. TSCLKx FREQUENCY IS NORMALLY 64 x TFS BUT MAY BE OPERATED IN BURST MODE.
Table 13-3 shows several modes that can be configured using bits in SPORTx_TCR1 and SPORTx_RCR1. The table shows bits for the receive side of the SPORT, but corresponding bits are available for configuring the transmit portion of the SPORT. A control field which may be either set or cleared depending on the user’s needs, without changing the standard, is indicated by an “X.”

### Table 13-3. Stereo Serial Settings

<table>
<thead>
<tr>
<th>Bit Field</th>
<th>Stereo Audio Serial Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$I^2S$</td>
</tr>
<tr>
<td>RSFSE</td>
<td>1</td>
</tr>
<tr>
<td>RRFST</td>
<td>0</td>
</tr>
<tr>
<td>LARFS</td>
<td>0</td>
</tr>
<tr>
<td>LRFS</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 13-22. SPORT Stereo Serial Modes, Receive

Table 13-3 shows several modes that can be configured using bits in SPORTx_TCR1 and SPORTx_RCR1. The table shows bits for the receive side of the SPORT, but corresponding bits are available for configuring the transmit portion of the SPORT. A control field which may be either set or cleared depending on the user’s needs, without changing the standard, is indicated by an “X.”

### Table 13-3. Stereo Serial Settings

<table>
<thead>
<tr>
<th>Bit Field</th>
<th>Stereo Audio Serial Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$I^2S$</td>
</tr>
<tr>
<td>RSFSE</td>
<td>1</td>
</tr>
<tr>
<td>RRFST</td>
<td>0</td>
</tr>
<tr>
<td>LARFS</td>
<td>0</td>
</tr>
<tr>
<td>LRFS</td>
<td>0</td>
</tr>
</tbody>
</table>
Note most bits shown as a 0 or 1 may be changed depending on the user’s preference, creating many other “almost standard” modes of stereo serial operation. These modes may be of use in interfacing to codecs with slightly non-standard interfaces. The settings shown in Table 13-3 provide glueless interfaces to many popular codecs.

Note RFSDIV or TFSDIV must still be greater than or equal to SLEN. For I2S operation, RFSDIV or TFSDIV is usually 1/64 of the serial clock rate. With RSFSE set, the formulas to calculate frame sync period and frequency (discussed in “Clock and Frame Sync Frequencies” on page 13-34) still apply, but now refer to one half the period and twice the frequency. For instance, setting RFSDIV or TFSDIV = 31 produces an LRCLK that transitions every 32 serial clock cycles and has a period of 64 serial clock cycles.

The LRFS bit determines the polarity of the RFS or TFS frame sync pin for the channel that is considered a “right” channel. Therefore, setting LRFS = 0 (meaning that it is an active high signal) indicates that the frame sync is high for the “right” channel, therefore implying that it is low for the “left” channel. This is the default setting.

### Table 13-3. Stereo Serial Settings (Cont’d)

<table>
<thead>
<tr>
<th>Bit Field</th>
<th>Stereo Audio Serial Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>I2S</td>
</tr>
<tr>
<td>RFSR</td>
<td>1</td>
</tr>
<tr>
<td>RCKFE</td>
<td>1</td>
</tr>
<tr>
<td>SLEN</td>
<td>2 – 31</td>
</tr>
<tr>
<td>RLSBIT</td>
<td>0</td>
</tr>
<tr>
<td>RFSDIV</td>
<td>2 – Max</td>
</tr>
<tr>
<td>(If internal FS is selected.)</td>
<td></td>
</tr>
<tr>
<td>RXSE</td>
<td>X</td>
</tr>
<tr>
<td>(Secondary Enable is available for RX and TX.)</td>
<td></td>
</tr>
</tbody>
</table>
Multichannel Operation

The **RRFST** and **TRFST** bits determine whether the first word received or transmitted is a left or a right channel. If the bit is set, the first word received or transmitted is a right channel. The default is to receive or transmit the left channel word first.

The secondary **DRxSEC** and **DTxSEC** pins are useful extensions of the serial port which pair well with stereo serial mode. Multiple I²S streams of data can be transmitted or received using a single SPORT. Note the primary and secondary pins are synchronous, as they share clock and **LRCLK** (frame sync) pins. The transmit and receive sides of the SPORT need not be synchronous, but may share a single clock in some designs. See Figure 13-3, which shows multiple stereo serial connections being made between the processor and an AD1836 codec.

Multichannel Operation

The SPORTs offer a multichannel mode of operation which allows the SPORT to communicate in a time-division-multiplexed (TDM) serial system. In multichannel communications, each data word of the serial bit stream occupies a separate channel. Each word belongs to the next consecutive channel so that, for example, a 24-word block of data contains one word for each of 24 channels.

The SPORT can automatically select words for particular channels while ignoring the others. Up to 128 channels are available for transmitting or receiving; each SPORT can receive and transmit data selectively from any of the 128 channels. These 128 channels can be any 128 out of the 1024 total channels. RX and TX must use the same 128-channel region to selectively enable channels. The SPORT can do any of the following on each channel:
Serial Port Controllers

- Transmit data
- Receive data
- Transmit and receive data
- Do nothing

Data companding and DMA transfers can also be used in multichannel mode.

The DTxPRI pin is always driven (not three-stated) if the SPORT is enabled (TSPEN = 1 in the SPORTx_TCR1 register), unless it is in multichannel mode and an inactive time slot occurs. The DTxSEC pin is always driven (not three-stated) if the SPORT is enabled and the secondary transmit is enabled (TXSE = 1 in the SPORTx_TCR2 register), unless the SPORT is in multichannel mode and an inactive time slot occurs.

In multichannel mode, RSCLKx can either be provided externally or generated internally by the SPORT, and it is used for both transmit and receive functions. Leave TSCLKx disconnected if the SPORT is used only in multichannel mode. If RSCLKx is externally or internally provided, it will be internally distributed to both the receiver and transmitter circuitry.

The SPORT multichannel transmit select register and the SPORT multichannel receive select register must be programmed before enabling SPORTx_TX or SPORTx_RX operation for multichannel mode. This is especially important in DMA data unpacked mode, since SPORT FIFO operation begins immediately after RSPEN and TSPEN are set, enabling both RX and TX. The MCMEN bit (in SPORTx_MCMC2) must be enabled prior to enabling SPORTx_TX or SPORTx_RX operation. When disabling the SPORT from multichannel operation, first disable TSPEN and then disable RSPEN. Note both TSPEN and RSPEN must be disabled before re-enabling. Disabling only TX or RX is not allowed.
**Figure 13-23** shows example timing for a multichannel transfer that has these characteristics:

- Use TDM method where serial data is sent or received on different channels sharing the same serial bus
- Can independently select transmit and receive channels
- \( RFS_x \) signals start of frame
- \( TFS_x \) is used as transmit data valid for external logic, true only during transmit channels
- Receive on channels 0 and 2, transmit on channels 1 and 2
- Multichannel frame delay is set to 1

See **“Timing Examples” on page 13-67** for more examples.
SPORT Multichannel Configuration (SPORTx_MCMCn) Registers

There are two SPORTx multichannel configuration registers (SPORTx_MCMCn) for each SPORT (see Figure 13-24 and Figure 13-25). The SPORTx_MCMCn registers are used to configure the multichannel operation of the SPORT. The two control registers are shown below.

The addresses for these SPORT registers are:

- SPORT0_MCMC1 – 0xFFC0 0838
- SPORT0_MCMC2 – 0xFFC0 083C
- SPORT1_MCMC1 – 0xFFC0 0938
- SPORT1_MCMC2 – 0xFFC0 093C
- SPORT2_MCMC1 – 0xFFC0 2538
- SPORT2_MCMC2 – 0xFFC0 253C
- SPORT3_MCMC1 – 0xFFC0 2638
- SPORT3_MCMC2 – 0xFFC0 263C

**Figure 13-24. SPORTx Multichannel Configuration Register 1 (SPORTx_MCMC1)**

*WSIZE[3:0] (Window Size)*
Value in field = [(Desired window size) / 8 – 1]

*WOFF[9:0] (Window Offset)*
Places start of window anywhere in the 0 to 1023 channel range

Reset = 0x0000
Multichannel Operation

**SPORTx Multichannel Configuration Register 2 (SPORTx_MCMC2)**

- **MFD[3:0] (Multichannel Frame Delay)**
  - Delay between frame sync pulse and the first data bit in Multichannel mode

- **FSDR (Frame Sync to Data Relationship)**
  - 0 - Normal
  - 1 - Reversed, H.100 mode

- **MCMEN (Multichannel Frame Mode Enable)**
  - 0 - Multichannel operations disabled
  - 1 - Multichannel operations enabled

- **MCSTM[1:0] (2X Clock Recovery Mode)**
  - 0x - Bypass mode
  - 10 - Recover 2 MHz clock from 4 MHz
  - 11 - Recover 8 MHz clock from 16 MHz

- **MCDTXPE (Multichannel DMA Transmit Packing)**
  - 0 - Disabled
  - 1 - Enabled

- **MCDRXPE (Multichannel DMA Receive Packing)**
  - 0 - Disabled
  - 1 - Enabled

- **Delay between frame sync pulse and the first data bit in Multichannel mode**

**Figure 13-25. SPORTx Multichannel Configuration Register 2**

**Multichannel Enable**

Setting the **MCMEN** bit in the **SPORTx_MCM2** register enables multichannel mode. When **MCMEN = 1**, multichannel operation is enabled; when **MCMEN = 0**, all multichannel operations are disabled.

- **Setting the MCMEN bit enables multichannel operation for both the receive and transmit sides of the SPORT. Therefore, if a receiving SPORT is in multichannel mode, the transmitting SPORT must also be in multichannel mode.**

- **When in multichannel mode, do not enable the stereo serial frame sync modes or the late frame sync feature, as these features are incompatible with multichannel mode.**
Table 13-4 shows the dependencies of bits in the SPORT configuration register when the SPORT is in multichannel mode.

Table 13-4. Multichannel Mode Configuration

<table>
<thead>
<tr>
<th>SPORTx_RCR1 or SPORTx_RCR2</th>
<th>SPORTx_TCR1 or SPORTx_TCR2</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSPEN</td>
<td>TSPEN</td>
<td>Set or clear both</td>
</tr>
<tr>
<td>IRCLK</td>
<td>–</td>
<td>Independent</td>
</tr>
<tr>
<td>–</td>
<td>ITCLK</td>
<td>Ignored</td>
</tr>
<tr>
<td>RDTYPE</td>
<td>TDTYPE</td>
<td>Independent</td>
</tr>
<tr>
<td>RLSBIT</td>
<td>TLSBIT</td>
<td>Independent</td>
</tr>
<tr>
<td>IRFS</td>
<td>–</td>
<td>Independent</td>
</tr>
<tr>
<td>–</td>
<td>ITFS</td>
<td>Ignored</td>
</tr>
<tr>
<td>RFSR</td>
<td>TFSR</td>
<td>Ignored</td>
</tr>
<tr>
<td>–</td>
<td>DITFS</td>
<td>Ignored</td>
</tr>
<tr>
<td>LRFS</td>
<td>LTFS</td>
<td>Independent</td>
</tr>
<tr>
<td>LARFS</td>
<td>LATFS</td>
<td>Both must be 0</td>
</tr>
<tr>
<td>RCKFE</td>
<td>TCKFE</td>
<td>Set or clear both to same value</td>
</tr>
<tr>
<td>SLEN</td>
<td>SLEN</td>
<td>Set or clear both to same value</td>
</tr>
<tr>
<td>RXSE</td>
<td>TXSE</td>
<td>Independent</td>
</tr>
<tr>
<td>RSFSE</td>
<td>TSFSE</td>
<td>Both must be 0</td>
</tr>
<tr>
<td>RRFST</td>
<td>TRFST</td>
<td>Ignored</td>
</tr>
</tbody>
</table>

Frame Syncs in Multichannel Mode

All receiving and transmitting devices in a multichannel system must have the same timing reference. The RFSx signal is used for this reference, indicating the start of a block or frame of multichannel data words.
When multichannel mode is enabled on a SPORT, both the transmitter and the receiver use \texttt{RFSx} as a frame sync. This is true whether \texttt{RFSx} is generated internally or externally. The \texttt{RFSx} signal is used to synchronize the channels and restart each multichannel sequence. Assertion of \texttt{RFSx} indicates the beginning of the channel 0 data word.

Since \texttt{RFSx} is used by both the \texttt{SPORTx\_TX} and \texttt{SPORTx\_RX} channels of the SPORT in multichannel mode configuration, the corresponding bit pairs in \texttt{SPORTx\_RCR1} and \texttt{SPORTx\_TCR1}, and in \texttt{SPORTx\_RCR2} and \texttt{SPORTx\_TCR2}, should always be programmed identically, with the possible exception of the RXSE and TXSE pair and the RDTYPE and TDTYPE pair. This is true even if \texttt{SPORTx\_RX} operation is not enabled.

In multichannel mode, \texttt{RFSx} timing similar to late (alternative) frame mode is entered automatically; the first bit of the transmit data word is available and the first bit of the receive data word is sampled in the same serial clock cycle that the frame sync is asserted, provided that \texttt{MFD} is set to 0.

The \texttt{TFSx} signal is used as a transmit data valid signal which is active during transmission of an enabled word. The SPORT’s data transmit pin is three-stated when the time slot is not active, and the \texttt{TFSx} signal serves as an output-enabled signal for the data transmit pin. The SPORT drives \texttt{TFSx} in multichannel mode whether or not \texttt{ITFS} is cleared. The \texttt{TFSx} pin in multichannel mode still obeys the \texttt{LTFS} bit. If \texttt{LTFS} is set, the transmit data valid signal will be active low—a low signal on the \texttt{TFSx} pin indicates an active channel.

Once the initial \texttt{RFSx} is received, and a frame transfer has started, all other \texttt{RFSx} signals are ignored by the SPORT until the complete frame has been transferred.

If \texttt{MFD > 0}, the \texttt{RFSx} may occur during the last channels of a previous frame. This is acceptable, and the frame sync is not ignored as long as the delayed channel 0 starting point falls outside the complete frame.
In multichannel mode, the $RFS_x$ signal is used for the block or frame start reference, after which the word transfers are performed continuously with no further $RFS$ signals required. Therefore, internally generated frame syncs are always data independent.

**The Multichannel Frame**

A multichannel frame contains more than one channel, as specified by the window size and window offset. A complete multichannel frame consists of $1 – 1024$ channels, starting with channel 0. The particular channels of the multichannel frame that are selected for the SPORT are a combination of the window offset, the window size, and the multichannel select registers. See Figure 13-26.

![Figure 13-26. Relationships for Multichannel Parameters](image-url)
Multichannel Operation

Multichannel Frame Delay

The 4-bit $\text{MFD}$ field in $\text{SPORTx\_MCMC2}$ specifies a delay between the frame sync pulse and the first data bit in multichannel mode. The value of $\text{MFD}$ is the number of serial clock cycles of the delay. Multichannel frame delay allows the processor to work with different types of interface devices.

A value of 0 for $\text{MFD}$ causes the frame sync to be concurrent with the first data bit. The maximum value allowed for $\text{MFD}$ is 15. A new frame sync may occur before data from the last frame has been received, because blocks of data occur back-to-back.

Window Size

The window size ($\text{WSIZE}[3:0]$) defines the number of channels that can be enabled/disabled by the multichannel select registers. This range of words is called the active window. The number of channels can be any value in the range of 0 to 15, corresponding to active window size of 8 to 128, in increments of 8; the default value of 0 corresponds to a minimum active window size of 8 channels. To calculate the active window size from the $\text{WSIZE}$ register, use this equation:

$$\text{Number of words in active window} = 8 \times (\text{WSIZE} + 1)$$

Since the DMA buffer size is always fixed, it is possible to define a smaller window size (for example, 32 words), resulting in a smaller DMA buffer size (in this example, 32 words instead of 128 words) to save DMA bandwidth. The window size cannot be changed while the SPORT is enabled.

Multichannel select bits that are enabled but fall outside the window selected are ignored.
Window Offset

The window offset \( \text{WOFF}[9:0] \) specifies where in the 1024-channel range to place the start of the active window. A value of 0 specifies no offset and 896 is the largest value that permits using all 128 channels. As an example, a program could define an active window with a window size of 8 \( (\text{FSIZE} = 0) \) and an offset of 93 \( (\text{WOFF} = 93) \). This 8-channel window would reside in the range from 93 to 100. Neither the window offset nor the window size can be changed while the SPORT is enabled.

If the combination of the window size and the window offset would place any portion of the window outside of the range of the channel counter, none of the out-of-range channels in the frame are enabled.

SPORT Current Channel (SPORTx_CHNL) Register

The 10-bit \texttt{CHNL} field in the SPORTx current channel register \( \text{SPORTx\_CHNL} \) indicates which channel is currently being serviced during multichannel operation (see Figure 13-27). This field is a read-only status indicator. The \texttt{CHNL}[9:0] field increments by one as each channel is serviced. The counter stops at the upper end of the defined window. The channel select register restarts at 0 at each frame sync. As an example, for a window size of 8 and an offset of 148, the counter displays a value between 0 and 156.

Once the window size has completed, the channel counter resets to 0 in preparation for the next frame. Because there are synchronization delays between \texttt{RSCLKx} and the processor clock, the channel register value is approximate. It is never ahead of the channel being served, but it may lag behind.
Multichannel Operation

The addresses for these SPORT registers are:

SPORT0_CHNL – 0xFFC0 0834

SPORT1_CHNL – 0xFFC0 0934

SPORT2_CHNL – 0xFFC0 2534

SPORT3_CHNL – 0xFFC0 2634

**SPORTx Current Channel Register (SPORTx_CHNL)**

<table>
<thead>
<tr>
<th>CHNL (Current Channel Indicator)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

Reset = 0x0000

Figure 13-27. SPORTx Current Channel Register

**Other Multichannel Fields in SPORTx_MCMC2**

The FSDR bit in the SPORTx_MCMC2 register changes the timing relationship between the frame sync and the clock received. This change enables the SPORT to comply with the H.100 protocol.

Normally (When FSDR = 0), the data is transmitted on the same edge that the TFSx is generated. For example, a positive edge on TFSx causes data to be transmitted on the positive edge of the TSCLKx—either the same edge or the following one, depending on when LATFS is set.

When the frame sync/data relationship is used (FSDR = 1), the frame sync is expected to change on the falling edge of the clock and is sampled on the rising edge of the clock. This is true even though data received is sampled on the negative edge of the receive clock.
Channel Selection Register

A channel is a multi-bit word from 3 to 32 bits in length that belongs to one of the TDM channels. Specific channels can be individually enabled or disabled to select which words are received and transmitted during multichannel communications. Data words from the enabled channels are received or transmitted, while disabled channel words are ignored. Up to 128 contiguous channels may be selected out of 1024 available channels. The SPORTx_MRCSn and SPORTx_MTCSn multichannel select registers are used to enable and disable individual channels; the SPORTx_MRCSn registers specify the active receive channels, and the SPORTx_MTCSn registers specify the active transmit channels.

Four registers make up each multichannel select register (see Figure 13-28). Each of the four registers has 32 bits, corresponding to 32 channels. Setting a bit enables that channel, so the SPORT selects its word from the multiple word block of data (for either receive or transmit).

![Channel Select 0 – 127](image)

Figure 13-28. Multichannel Select Registers

Channel select bit 0 always corresponds to the first word of the active window. To determine a channel’s absolute position in the frame, add the window offset words to the channel select position. For example, setting bit 7 in MCS2 selects word 71 of the active window to be enabled. Setting bit 2 in MCS1 selects word 34 of the active window, and so on.

Setting a particular bit in the SPORTx_MTCSn register causes the SPORT to transmit the word in that channel’s position of the data stream. Clearing the bit in the SPORTx_MTCSn register causes the SPORT’s data transmit pin to three-state during the time slot of that channel.
Multichannel Operation

Setting a particular bit in the SPORTx_MRCSn register causes the SPORT to receive the word in that channel’s position of the data stream; the received word is loaded into the SPORTx_RX buffer. Clearing the bit in the SPORTx_MRCSn register causes the SPORT to ignore the data.

Companding may be selected for all channels or for no channels. A-law or μ-law companding is selected with the TDTYPE field in the SPORTx_TCR1 register and the RDTYPE field in the SPORTx_RCR1 register, and applies to all active channels. (See “Companding” on page 13-37 for more information about companding.)

SPORT Multichannel Receiv e Selection (SPORTx_MRCSn) Registers

The multichannel selection registers are used to enable and disable individual channels. The SPORTx multichannel receive select registers (SPORTx_MRCSn, see Figure 13-29 and Table 13-5) specify the active receive channels. There are four registers, each with 32 bits, corresponding to the 128 channels. Setting a bit enables that channel so that the serial port selects that word for receive from the multiple word block of data. For example, setting bit 0 selects word 0, setting bit 12 selects word 12, and so on.

Setting a particular bit in the SPORTx_MRCSn register causes the serial port to receive the word in that channel’s position of the data stream; the received word is loaded into the RX buffer. Clearing the bit in the SPORTx_MRCSn register causes the serial port to ignore the data.
SPORTx Multichannel Receive Select Registers (SPORTx_MRCSn)
For all bits, 0 - Channel disabled, 1 - Channel enabled, so SPORT selects that word from multiple word
block of data.
For memory-mapped addresses, see Table 13-5.

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPORT0_MRCS0</td>
<td>0xFFC0 0850</td>
<td>SPORT2_MRCS0</td>
<td>0xFFC0 2550</td>
</tr>
<tr>
<td>SPORT0_MRCS1</td>
<td>0xFFC0 0854</td>
<td>SPORT2_MRCS1</td>
<td>0xFFC0 2554</td>
</tr>
<tr>
<td>SPORT0_MRCS2</td>
<td>0xFFC0 0858</td>
<td>SPORT2_MRCS2</td>
<td>0xFFC0 2558</td>
</tr>
<tr>
<td>SPORT0_MRCS3</td>
<td>0xFFC0 085C</td>
<td>SPORT2_MRCS3</td>
<td>0xFFC0 255C</td>
</tr>
<tr>
<td>SPORT1_MRCS0</td>
<td>0xFFC0 0950</td>
<td>SPORT3_MRCS0</td>
<td>0xFFC0 2650</td>
</tr>
<tr>
<td>SPORT1_MRCS1</td>
<td>0xFFC0 0954</td>
<td>SPORT3_MRCS1</td>
<td>0xFFC0 2654</td>
</tr>
<tr>
<td>SPORT1_MRCS2</td>
<td>0xFFC0 0958</td>
<td>SPORT3_MRCS2</td>
<td>0xFFC0 2658</td>
</tr>
<tr>
<td>SPORT1_MRCS3</td>
<td>0xFFC0 095C</td>
<td>SPORT3_MRCS3</td>
<td>0xFFC0 265C</td>
</tr>
</tbody>
</table>
SPORT Multichannel Transmit Selection (SPORTx_MTCSn) Registers

The multichannel selection registers are used to enable and disable individual channels. The four SPORTx multichannel transmit select registers (SPORTx_MTCSn, see Figure 13-30 and Table 13-6) specify the active transmit channels. There are four registers, each with 32 bits, corresponding to the 128 channels. Setting a bit enables that channel so that the serial port selects that word for transmit from the multiple word block of data. For example, setting bit 0 selects word 0, setting bit 12 selects word 12, and so on.

Setting a particular bit in a SPORTx_MTCSn register causes the serial port to transmit the word in that channel’s position of the data stream. Clearing the bit in the SPORTx_MTCSn register causes the serial port’s data transmit pin to three-state during the time slot of that channel.

<table>
<thead>
<tr>
<th>SPORTx_MTCS0</th>
<th>Channel number</th>
<th>Bit number in register</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MTCS0</td>
<td>0000000000000000000000000000000000000000000000000000000000000000</td>
<td>0x0000 0000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SPORTx_MTCS1</th>
<th>Channel number</th>
<th>Bit number in register</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>MTCS1</td>
<td>0000000000000000000000000000000000000000000000000000000000000000</td>
<td>0x0000 0000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SPORTx_MTCS2</th>
<th>Channel number</th>
<th>Bit number in register</th>
</tr>
</thead>
<tbody>
<tr>
<td>95</td>
<td>64</td>
<td>0</td>
</tr>
<tr>
<td>MTCS2</td>
<td>0000000000000000000000000000000000000000000000000000000000000000</td>
<td>0x0000 0000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SPORTx_MTCS3</th>
<th>Channel number</th>
<th>Bit number in register</th>
</tr>
</thead>
<tbody>
<tr>
<td>127</td>
<td>96</td>
<td>0</td>
</tr>
<tr>
<td>MTCS3</td>
<td>0000000000000000000000000000000000000000000000000000000000000000</td>
<td>0x0000 0000</td>
</tr>
</tbody>
</table>

Figure 13-30. SPORTx Multichannel Transmit Select Registers
Multichannel DMA Data Packing

Multichannel DMA data packing and unpacking are specified with the MCDTXPE and MCDRXPE bits in the SPORTx_MCMC2 multichannel configuration register.

If the bits are set, indicating that data is packed, the SPORT expects the data contained by the DMA buffer corresponds only to the enabled SPORT channels. For example, if an MCM frame contains 10 enabled channels, the SPORT expects the DMA buffer to contain 10 consecutive words for each frame. It is not possible to change the total number of enabled channels without changing the DMA buffer size, and reconfiguring is not allowed while the SPORT is enabled.

If the bits are cleared (the default, indicating that data is not packed), the SPORT expects the DMA buffer to have a word for each of the channels in the active window, whether enabled or not, so the DMA buffer size must be equal to the size of the window. For example, if channels 1 and 10 are enabled, and the window size is 16, the DMA buffer size would have to be 16 words. The data to be transmitted or received would be placed at

Table 13-6. SPORTx Multichannel Transmit Select Register Memory-Mapped Addresses

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPORT0_MTCS0</td>
<td>0xFFC0 0840</td>
<td>SPORT2_MTCS0</td>
<td>0xFFC0 2540</td>
</tr>
<tr>
<td>SPORT0_MTCS1</td>
<td>0xFFC0 0844</td>
<td>SPORT2_MTCS1</td>
<td>0xFFC0 2544</td>
</tr>
<tr>
<td>SPORT0_MTCS2</td>
<td>0xFFC0 0848</td>
<td>SPORT2_MTCS2</td>
<td>0xFFC0 2548</td>
</tr>
<tr>
<td>SPORT0_MTCS3</td>
<td>0xFFC0 084C</td>
<td>SPORT2_MTCS3</td>
<td>0xFFC0 254C</td>
</tr>
<tr>
<td>SPORT1_MTCS0</td>
<td>0xFFC0 0940</td>
<td>SPORT3_MTCS0</td>
<td>0xFFC0 2640</td>
</tr>
<tr>
<td>SPORT1_MTCS1</td>
<td>0xFFC0 0944</td>
<td>SPORT3_MTCS1</td>
<td>0xFFC0 2644</td>
</tr>
<tr>
<td>SPORT1_MTCS2</td>
<td>0xFFC0 0948</td>
<td>SPORT3_MTCS2</td>
<td>0xFFC0 2648</td>
</tr>
<tr>
<td>SPORT1_MTCS3</td>
<td>0xFFC0 094C</td>
<td>SPORT3_MTCS3</td>
<td>0xFFC0 264C</td>
</tr>
</tbody>
</table>
Support for H.100 Standard Protocol

addresses 1 and 10 of the buffer, and the rest of the words in the DMA buffer would be ignored. This mode allows changing the number of enabled channels while the SPORT is enabled, with some caution. First read the channel register to make sure that the active window is not being serviced. If the channel count is 0, then the multichannel select registers can be updated.

Support for H.100 Standard Protocol

The processor supports the H.100 standard protocol. The following SPORT parameters must be set to support this standard.

• Set for external frame sync. Frame sync generated by external bus master.
• TFSR/RFSR set (frame syncs required)
• LTFS/LRFS set (active low frame syncs)
• Set for external clock
• MCMEN set (multichannel mode selected)
• MFD = 0 (no frame delay between frame sync and first data bit)
• SLEN = 7 (8-bit words)
• FSDR = 1 (set for H.100 configuration, enabling half-clock-cycle early frame sync)

2X Clock Recovery Control

The SPORTs can recover the data rate clock from a provided 2X input clock. This enables the implementation of H.100 compatibility modes for MVIP-90 (2 Mbps data) and HMVIP (8 Mbps data), by recovering 2 MHz from 4 MHz or 8 MHz from the 16 MHz incoming clock with
the proper phase relationship. A 2-bit mode signal (MCCRM[1:0] in the
SPORTx_MCMC2 register) chooses the applicable clock mode, which includes
a non-divide or bypass mode for normal operation. A value of MCCRM = 00
chooses non-divide or bypass mode (H.100-compatible), MCCRM = 10
chooses MVIP-90 clock divide (extract 2 MHz from 4 MHz), and
MCCRM = 11 chooses HMVIP clock divide (extract 8 MHz from 16 MHz).

**SPORT Pin/Line Terminations**

The processor has very fast drivers on all output pins, including the
SPORTs. If connections on the data, clock, or frame sync lines are longer
than six inches, consider using a series termination for strip lines on
point-to-point connections. This may be necessary even when using low
speed serial clocks, because of the edge rates.

**Timing Examples**

Several timing examples are included within the text of this chapter (in the
sections “Framed Versus Unframed” on page 13-39, “Early Versus Late
Frame Syncs (Normal Versus Alternate Timing)” on page 13-43, and
“Frame Syncs in Multichannel Mode” on page 13-55). This section con-
tains additional examples to illustrate other possible combinations of the
framing options.

These timing examples show the relationships between the signals but are
not scaled to show the actual timing parameters of the processor. Consult
the *ADSP-BF538/ADSP-BF538F Embedded Processor Data Sheet* for actual
timing parameters and values.

These examples assume a word length of four bits (SLEN = 3). Framing
signals are active high (LRFS = 0 and LTFS = 0).

Figure 13-31 through Figure 13-36 show framing for receiving data.
In Figure 13-31 and Figure 13-32, the normal framing mode is shown for non-continuous data (any number of $TSCLK_x$ or $RSCLK_x$ cycles between words) and continuous data (no $TSCLK_x$ or $RSCLK_x$ cycles between words).

Figure 13-31. SPORT Receive, Normal Framing

Figure 13-32. SPORT Continuous Receive, Normal Framing

Figure 13-33 and Figure 13-34 show non-continuous and continuous receiving in the alternate framing mode. These four figures show the input timing requirement for an externally generated frame sync and also the output timing characteristic of an internally generated frame sync. Note the output meets the input timing requirement; therefore, with two SPORT channels used, one SPORT channel could provide $RFS_x$ for the other SPORT channel.
Serial Port Controllers

Figure 13-35 and Figure 13-36 show the receive operation with normal framing and alternate framing, respectively, in the unframed mode. A single frame sync signal occurs only at the start of the first word, either one RSCLKx before the first bit (in normal mode) or at the same time as the first bit (in alternate mode). This mode is appropriate for multi word bursts (continuous reception).
Timing Examples

Figure 13-35. SPORT Receive, Unframed Mode, Normal Framing

Figure 13-36. SPORT Receive, Unframed Mode, Alternate Framing

Figure 13-37 through Figure 13-42 show framing for transmitting data and are very similar to Figure 13-31 through Figure 13-36.

In Figure 13-37 and Figure 13-38, the normal framing mode is shown for non-continuous data (any number of TSCLKx cycles between words) and continuous data (no TSCLKx cycles between words). Figure 13-39 and Figure 13-40 show non-continuous and continuous transmission in the alternate framing mode. As noted previously for the receive timing diagrams, the RFSx output meets the RFSx input timing requirement.
Serial Port Controllers

Figure 13-37. SPORT Transmit, Normal Framing

Figure 13-38. SPORT Continuous Transmit, Normal Framing

Figure 13-39. SPORT Transmit, Alternate Framing
Figure 13-40. SPORT Continuous Transmit, Alternate Framing

Figure 13-41 and Figure 13-42 show the transmit operation with normal framing and alternate framing, respectively, in the unframed mode. A single frame sync signal occurs only at the start of the first word, either one TSCLK before the first bit (in normal mode) or at the same time as the first bit (in alternate mode).

Figure 13-41. SPORT Transmit, Unframed Mode, Normal Framing

Figure 13-42. SPORT Transmit, Unframed Mode, Alternate Framing
The processor supports 16 bidirectional general-purpose I/O pins, \( PF[15:0] \) on GPIO port F. Each pin can be individually configured as either an input or an output by using the GPIO port F direction register \( \text{PORTFIO\_DIR} \). When configured as output, the GPIO port F data register \( \text{PORTFIO} \) can be directly written to specify the state of all \( PFx \) pins. When configured as an output, the state written to the GPIO port F set \( \text{PORTFIO\_SET} \), GPIO port F clear \( \text{PORTFIO\_CLEAR} \), and GPIO port F toggle \( \text{PORTFIO\_TOGGLE} \) registers determines the state driven by the output \( PFx \) pin. Regardless of whether the pins are configured, as inputs or outputs, reading any of these registers \( \text{PORTFIO, PORTFIO\_SET, PORTFIO\_CLEAR, PORTFIO\_TOGGLE} \) returns the state of each pin.

Each \( PFx \) pin can be configured to generate an interrupt. When a \( PFx \) pin is configured as an input, an interrupt can be generated according to the state of the pin (either high or low), an edge transition (low to high or high to low), or on both edge transitions (low to high and high to low). Input sensitivity is defined on a per-bit basis by the polarity register \( \text{PORTFIO\_POLAR} \), the GPIO port F interrupt sensitivity register \( \text{PORTFIO\_EDGE} \) and the GPIO port F set on both edges register \( \text{PORTFIO\_BOTH} \). Input polarity is defined on a per-bit basis by the \( \text{PORTFIO\_POLAR} \) register. When the \( PFx \) inputs are enabled and a \( PFx \) pin is configured as an output, enabling interrupts for the pin allows an interrupt to be generated by setting the \( PFx \) pin.
The processor provides two independent interrupt channels for the PFx pins. Identical in functionality, these are called GPIO interrupt A and GPIO interrupt B. Each interrupt channel has four mask registers associated with it, a GPIO interrupt mask data register (PORTFIO_MASKx), a GPIO interrupt mask set register (PORTFIO_MASKx_SET), a GPIO interrupt mask clear register (PORTFIO_MASKx_CLEAR), and a GPIO interrupt mask toggle register (PORTFIO_MASKx_TOGGLE).

Each PFx pin is represented by a bit in each of these eight registers. Writing a 1 to a bit in a PORTFIO_MASKx_SET register enables interrupt generation for that PFx pin, while writing a 1 to a bit in a PORTFIO_MASKx_CLEAR register disables interrupt generation for that PFx pin.

The interrupt masking can be toggled by writing a 1 to a bit in the PORTFIO_MASKx_TOGGLE register. Additionally, the mask bits can be directly written by writing to the PORTFIO_MASKx register. This flexible mechanism allows each bit to generate GPIO interrupt A, GPIO interrupt B, both GPIO interrupts A and B, or neither.

When a PFx pin is not used in a system, the input buffer can be disabled so that no external pull-ups or pull-downs are required on the unused pins. By default, the input buffers are disabled. They can be enabled via bits in the GPIO port F input enable register (PORTFIO_INEN).

The PFx pins are multiplexed for use by the parallel peripheral interface (PPI), timers, and serial peripheral interface (SPI0). Table 14-1 shows the programmable GPIO pins and their multiplexed functionality.
### Table 14-1. GPIO Port F Pins and Functionality

<table>
<thead>
<tr>
<th>PFx Pin</th>
<th>Peripheral That Shares the PFx Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PPI</td>
</tr>
<tr>
<td>PF0</td>
<td>Slave Select Input (SPI0SS)</td>
</tr>
<tr>
<td>PF1</td>
<td>Slave Select Enable 1 (SPI0SEL1)</td>
</tr>
<tr>
<td>PF2</td>
<td>Slave Select Enable 2 (SPI0SEL2)</td>
</tr>
<tr>
<td>PF3</td>
<td>Frame Sync 3 (PPI_FS3)</td>
</tr>
<tr>
<td>PF4</td>
<td>I/O #15 (PPI15)</td>
</tr>
<tr>
<td>PF5</td>
<td>I/O #14 (PPI14)</td>
</tr>
<tr>
<td>PF6</td>
<td>I/O #13 (PPI13)</td>
</tr>
<tr>
<td>PF7</td>
<td>I/O #12 (PPI12)</td>
</tr>
<tr>
<td>PF8</td>
<td>I/O #11 (PPI11)</td>
</tr>
<tr>
<td>PF9</td>
<td>I/O #10 (PPI10)</td>
</tr>
<tr>
<td>PF10</td>
<td>I/O #9 (PPI9)</td>
</tr>
<tr>
<td>PF11</td>
<td>I/O #8 (PPI8)</td>
</tr>
<tr>
<td>PF12</td>
<td>I/O #7 (PPI7)</td>
</tr>
<tr>
<td>PF13</td>
<td>I/O #6 (PPI6)</td>
</tr>
<tr>
<td>PF14</td>
<td>I/O #5 (PPI5)</td>
</tr>
<tr>
<td>PF15</td>
<td>I/O #4 (PPI4)</td>
</tr>
</tbody>
</table>

Table 14-2 describes how to use the peripheral function that shares the PFx pins.
Table 14-2. Peripheral Function Usage That Shares the PFx Pin

<table>
<thead>
<tr>
<th>PFx Pin</th>
<th>To Use the Peripheral Function That Shares the PFx Pin... (Assumes Peripheral is Enabled)</th>
<th>SPI</th>
<th>Timers 0, 1,2</th>
</tr>
</thead>
<tbody>
<tr>
<td>PF0</td>
<td>Set PSSE in SPI0_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PF1</td>
<td>Set FLS1 in SPI0_FLG</td>
<td></td>
<td>Write 1 to CLK_SEL in TIMERx_CONFIG</td>
</tr>
<tr>
<td>PF2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PF3</td>
<td>In PPI_CTL:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>If PORT_DIR = 1, write 01 to PORT_CFG.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>If PORT_DIR = 0, write 10 to PORT_CFG.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PF4</td>
<td>Write b#111 to DLEN in PPI_CTL</td>
<td></td>
<td>Set FLS4 in SPI0_FLG</td>
</tr>
<tr>
<td>PF5</td>
<td>Write b#110 to DLEN in PPI_CTL</td>
<td></td>
<td>Set FLS5 in SPI0_FLG</td>
</tr>
<tr>
<td>PF6</td>
<td>Write b#101 to DLEN in PPI_CTL</td>
<td></td>
<td>Set FLS6 in SPI0_FLG</td>
</tr>
<tr>
<td>PF7</td>
<td>Write b#100 to DLEN in PPI_CTL</td>
<td></td>
<td>Set FLS7 in SPI0_FLG</td>
</tr>
<tr>
<td>PF8</td>
<td>Write b#011 to DLEN in PPI_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PF9</td>
<td>Write b#010 to DLEN in PPI_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PF10</td>
<td>Write b#001 to DLEN in PPI_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PF11</td>
<td>Write b#001 to DLEN in PPI_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PF12</td>
<td>Always enabled when PPI enabled</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PF13</td>
<td>Always enabled when PPI enabled</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PF14</td>
<td>Always enabled when PPI enabled</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PF15</td>
<td>Always enabled when PPI enabled</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For more information, see Chapter 11, “Parallel Peripheral Interface”, Chapter 10, “SPI Compatible Port Controllers” and Chapter 16, “Timers”.
**GPIO Port F Registers (MMRs)**

The GPIO port F registers are part of the system memory-mapped registers (MMRs). The addresses of the programmable GPIO MMRs appear in Appendix B, “System MMR Assignments”. Core access to the GPIO configuration registers is through the system bus.

**GPIO Port F Direction (PORTFIO_DIR) Register**

The GPIO port F direction register (PORTFIO_DIR) is a read-write register. (See Figure 14-1.) Each bit position corresponds to a PFx pin. A logic 1 configures the PFx pin as an output, driving the state contained in the PORTFIO register. A logic 0 configures the PFx pin as an input. The reset value of this register is 0x0000, making all PFx pins inputs upon reset.

> When using the PFx pin as an input, the corresponding bit should also be set in the input enable (PORTFIO_INEN) register.

![Figure 14-1. GPIO Port F Direction Register](chart)
GPIO Port F Registers (MMRs)

GPIO Port F Value Registers Overview

The processor has four GPIO value registers:

- GPIO data register (PORTFIO)
- GPIO set register (PORTFIO_SET)
- GPIO clear register (PORTFIO_CLEAR)
- GPIO toggle register (PORTFIO_TOGGLE)

These registers are used to:

- Sense the value of the PFx pins defined as inputs
- Specify the state of PFx pins defined as outputs
- Clear interrupts generated by the PFx pins

Each PFx pin is represented by a bit in each of the four registers.

Reading any of the PORTFIO, PORTFIO_SET, PORTFIO_CLEAR, or PORTFIO_TOGGLE registers returns the value of the PFx pins. The value returned shows the state of the PFx pins defined as outputs and the sense of PFx pins defined as inputs, based on the polarity and sensitivity settings of each pin.

Reading the PORTFIO, PORTFIO_SET, PORTFIO_CLEAR, or PORTFIO_TOGGLE register after reset results in 0x0000 because although the pins are inputs, the input buffers are not enabled. See Table 14-3 for guidance on how to interpret a value read from one of these registers, based on the settings of the PORTFIO_POLAR, PORTFIO_EDGE, and PORTFIO_BOTH registers.
For pins configured as edge-sensitive, a read back of 1 from one of these registers is sticky. That is, once the bit is set, it remains set until cleared by the program. For level-sensitive pins, the pin state is checked every cycle, so the read back value changes when the level on the pin changes.

For more information about the GPIO set, GPIO clear, and GPIO toggle registers, see “GPIO Port F Set (PORTFIO_SET), GPIO Port F Clear (PORTFIO_CLEAR), and GPIO Port F Toggle (PORTFIO_TOGGLE) Registers” on page 14-8.

### Table 14-3. GPIO Port F Value Register Pin Interpretation

<table>
<thead>
<tr>
<th>PORTFIO_POLAR</th>
<th>PORTFIO_EDGE</th>
<th>PORTFIO_BOTH</th>
<th>Effect of MMR Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>Pin that is high reads as 1; pin that is low reads as 0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>If rising edge occurred, pin reads as 1; otherwise, pin reads as 0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
<td>Pin that is low reads as 1; pin that is high reads as 0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>If falling edge occurred, pin reads as 1; otherwise, pin reads as 0</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>1</td>
<td>If any edge occurred, pin reads as 1; otherwise, pin reads as 0</td>
</tr>
</tbody>
</table>
GPIO Port F Data (PORTFIO) Register

When written, the GPIO data register (PORTFIO), shown in Figure 14-2, directly specifies the state of all PFx pins. When read, the register returns the value of the PFx pins.

GPIO Port F Set (PORTFIO_SET), GPIO Port F Clear (PORTFIO_CLEAR), and GPIO Port F Toggle (PORTFIO_TOGGLE) Registers

The GPIO set (PORTFIO_SET), GPIO clear (PORTFIO_CLEAR), and GPIO toggle (PORTFIO_TOGGLE) registers are used to:

- Set, clear or toggle the output state associated with each output PFx pin
- Clear the latched interrupt state captured from each input PFx pin

This mechanism is used to avoid the potential issues with more traditional read-modify-write mechanisms. Reading any of these registers returns the GPIO pin state.
Figure 14-3 and Figure 14-4 represent the PORTFIO_SET and PORTFIO_CLEAR registers, respectively. Figure 14-5 represents the PORTFIO_TOGGLE register.

**GPIO Port F Set Register (PORTFIO_SET)**
Write-1-to-set

```
0xFFC0 0708
```

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Reset = 0x00000xFFC0 0708

- Set PF15
- Set PF14
- Set PF13
- Set PF12
- Set PF11
- Set PF10
- Set PF9
- Set PF8

**GPIO Port F Clear Register (PORTFIO_CLEAR)**
Write-1-to-clear

```
0xFFC0 0704
```

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Reset = 0x00000xFFC0 0704

- Clear PF15
- Clear PF14
- Clear PF13
- Clear PF12
- Clear PF11
- Clear PF10
- Clear PF9
- Clear PF8
- Clear PF7
- Clear PF6
- Clear PF5
- Clear PF4
- Clear PF3
- Clear PF2
- Clear PF1
- Clear PF0

Figure 14-3. GPIO Port F Set Register

Figure 14-4. GPIO Port F Clear Register
As an example of how these registers work, assume that PF0 is configured as an output (PORTFIO_DIR = 0x0001). Writing 0x0001 to the PORTFIO_SET register drives a logic 1 on the PF0 pin without affecting the state of any other PFx pins. Writing 0x0001 to the PORTFIO_CLEAR register drives a logic 0 on the PF0 pin without affecting the state of any other PFx pins. Writing a 0x0001 to the PORTFIO_TOGGLE register changes the pin state on PF0 from logic 0 to logic 1 or from logic 1 to logic 0, depending upon the existing pin state.

Writing a 0 to the PORTFIO_SET, PORTFIO_CLEAR, or PORTFIO_TOGGLE registers has no effect on the value of the GPIO pin and is therefore ignored.

Reading the PORTFIO_SET or PORTFIO_CLEAR registers returns:

- 0s for PFx pins defined as outputs and driven low
- 1s for pins (including PF0 in the example above) defined as outputs and driven high
- The present sense of PFx pins defined as inputs

Figure 14-5. GPIO Port F Toggle Register
Input sense is based on PORTFIO_POLAR and PORTFIO_EDGE register settings, as well as the logic level at each pin.

**General-Purpose Input/Output Port F**

**GPIO Port F Mask Interrupt Registers Overview**

The processor supports up to two interrupt sources for each of the GPIO pins—GPIO interrupt A and GPIO interrupt B. These interrupts are configurable in a set of GPIO mask interrupt registers (PORTFIO_MASKA, PORTFIO_MASKA_CLEAR, PORTFIO_MASKA_SET, PORTFIO_MASKA_TOGGLE, PORTFIO_MASKB, PORTFIO_MASKB_CLEAR, PORTFIO_MASKB_SET, and PORTFIO_MASKB_TOGGLE) which are implemented as complementary pairs of write-1-to-set, write-1-to-clear, and write-1-to-toggle registers.

Both GPIO interrupt A and GPIO interrupt B are supported by a set of four dedicated registers:

- GPIO mask interrupt data registers (PORTFIO_MASKA and PORTFIO_MASKB)
- GPIO mask interrupt set registers (PORTFIO_MASKA_SET and PORTFIO_MASKB_SET)
- GPIO mask interrupt clear registers (PORTFIO_MASKA_CLEAR and PORTFIO_MASKB_CLEAR)
- GPIO interrupt toggle registers (PORTFIO_MASKA_TOGGLE and PORTFIO_MASKB_TOGGLE)

This implementation provides the ability to enable or disable a PFx pin to act as a processor interrupt without requiring read-modify-write accesses—or to directly specify the mask value with the data register. For diagrams of the registers that support GPIO interrupt A, see “GPIO Port F Interrupt A (PORTFIO_MASKA, PORTFIO_MASKA_CLEAR, PORTFIO_MASKA_SET, PORTFIO_MASKA_TOGGLE) Registers” on page 14-15.
For diagrams of the registers that support GPIO interrupt B, see “GPIO Port F Interrupt B (PORTFIO_MASKB, PORTFIO_MASKB_CLEAR, PORTFIO_MASKB_SET, PORTFIO_MASKB_TOGGLE) Registers” on page 14-17.

Each PFx pin is represented by a bit in each of the eight registers. Table 14-4 shows the effect of writing 1 to a bit in a PORTFIO_MASKx_SET, PORTFIO_MASKx_CLEAR, or PORTFIO_MASKx_TOGGLE registers.

Table 14-4. Effect of Writing 1 to a Bit

<table>
<thead>
<tr>
<th>Register</th>
<th>Effect of Writing 1 to a Bit in the Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>PORTFIO_MASKx_SET</td>
<td>Enables GPIO x interrupt generation for that PFx pin</td>
</tr>
<tr>
<td>PORTFIO_MASKx_CLEAR</td>
<td>Disables GPIO x interrupt generation for that PFx pin</td>
</tr>
<tr>
<td>PORTFIO_MASKx_TOGGLE</td>
<td>Changes the state of GPIO x interrupt generation capability</td>
</tr>
</tbody>
</table>

Reading any of the PORTFIO_MASKx, PORTFIO_MASKx_SET, PORTFIO_MASKx_CLEAR, or PORTFIO_MASKx_TOGGLE registers returns the value of the current PORTFIO_MASKx register.

GPIO interrupt A and GPIO interrupt B operate independently. For example, writing 1 to a bit in the PORTFIO_MASKA_SET register does not affect GPIO interrupt B. This facility allows PFx pins to generate GPIO interrupt A, GPIO interrupt B, both GPIO interrupts A and B, or neither.

Note a GPIO interrupt is generated by a logical OR of all unmasked PFx pins for that interrupt. For example, if PF0 and PF1 are both unmasked for GPIO interrupt A, GPIO interrupt A is generated when triggered by PF0 or PF1.
When using either rising or falling edge-triggered interrupts, the interrupt condition must be cleared each time a corresponding interrupt is serviced by writing 1 to the appropriate PORTFIO_CLEAR register bit.

At reset, all interrupts are masked.

**GPIO Port F Interrupt Generation Flow**

*Figure 14-6* shows the process by which GPIO interrupt A or GPIO interrupt B generates an event. Note the flow is shown for only one programmable GPIO, “GPIOOn.” However, a GPIO interrupt is generated by a logical OR of all unmasked PF$x$ pins for that interrupt. For example, if only PF0 and PF1 are unmasked for GPIO interrupt A, this interrupt is generated when triggered by either PF0 or PF1.
Figure 14-6. GPIO Port F Interrupt Generation Flow
General-Purpose Input/Output Port F

**GPIO Port F Interrupt A (PORTFIO_MASKA, PORTFIO_MASKA_CLEAR, PORTFIO_MASKA_SET, PORTFIO_MASKA_TOGGLE) Registers**

The registers shown in Figure 14-7 through Figure 14-10 support GPIO interrupt A.

**GPIO Port F Mask Interrupt A Data Register (PORTFIO_MASKA)**

For all bits, 1 - Enable

![Figure 14-7. GPIO Port F Mask Interrupt A Data Register](image)

**GPIO Port F Mask Interrupt A Set Register (PORTFIO_MASKA_SET)**

For all bits, 1 - Set

![Figure 14-8. GPIO Port F Mask Interrupt A Set Register](image)
GPIO Port F Registers (MMRs)

GPIO Port F Mask Interrupt A Clear Register (PORTFIO_MASKA_CLEAR)
For all bits, 1 - Clear

0xFFF0 0714

Reset = 0x0000

Clear PF15 interrupt mask
Clear PF14 interrupt mask
Clear PF13 interrupt mask
Clear PF12 interrupt mask
Clear PF11 interrupt mask
Clear PF10 interrupt mask
Clear PF9 interrupt mask
Clear PF8 interrupt mask

Figure 14-9. GPIO Port F Mask Interrupt A Clear Register

GPIO Port F Mask Interrupt A Toggle Register (PORTFIO_MASKA_TOGGLE)
For all bits, 1 - Toggle

0xFFF0 071C

Reset = 0x0000

Toggle PF15 interrupt mask
Toggle PF14 interrupt mask
Toggle PF13 interrupt mask
Toggle PF12 interrupt mask
Toggle PF11 interrupt mask
Toggle PF10 interrupt mask
Toggle PF9 interrupt mask
Toggle PF8 interrupt mask

Figure 14-10. GPIO Port F Mask Interrupt A Toggle Register
General-Purpose Input/Output Port F

GPIO Port F Interrupt B (PORTFIO_MASKB, PORTFIO_MASKB_CLEAR, PORTFIO_MASKB_SET, PORTFIO_MASKB_TOGGLE) Registers

The registers shown in Figure 14-11 through Figure 14-14 support GPIO interrupt B.

**GPIO Port F Mask Interrupt B Data Register (PORTFIO_MASKB)**
For all bits, 1 - Enable

0xFFF0 0720

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Reset = 0x0000

- Enable PF15 interrupt mask
- Enable PF14 interrupt mask
- Enable PF13 interrupt mask
- Enable PF12 interrupt mask
- Enable PF11 interrupt mask
- Enable PF10 interrupt mask
- Enable PF9 interrupt mask
- Enable PF8 interrupt mask

**Figure 14-11. GPIO Port F Mask Interrupt B Data Register**

**GPIO Port F Mask Interrupt B Set Register (PORTFIO_MASKB_SET)**
For all bits, 1 - Set

0xFFF0 0728

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Reset = 0x0000

- Set PF15 interrupt mask
- Set PF14 interrupt mask
- Set PF13 interrupt mask
- Set PF12 interrupt mask
- Set PF11 interrupt mask
- Set PF10 interrupt mask
- Set PF9 interrupt mask
- Set PF8 interrupt mask

**Figure 14-12. GPIO Port F Mask Interrupt B Set Register**
The GPIO polarity register (PORTFIO_POLAR) shown in Figure 14-15 is used to configure the polarity of the GPIO input source. To select active high or rising edge, set the bits in this register to 0. To select active low or falling edge, set the bits in this register to 1.
This register has no effect on PFx pins that are defined as outputs. The contents of this register are cleared at reset, defaulting to active high polarity.

**GPIO Port F Polarity Register (PORTFIO_POLAR)**

For all bits, 0 - Active high or rising edge, 1 - Active low or falling edge

![GPIO Port F Polarity Register](image)

Figure 14-15. GPIO Port F Polarity Register

**GPIO Port F Interrupt Sensitivity (PORTFIO_EDGE) Register**

The GPIO interrupt sensitivity register (PORTFIO_EDGE) shown in Figure 14-16 is used to configure each of the GPIOs as either a level-sensitive or an edge-sensitive source. When using an edge-sensitive mode, an edge detection circuit is used to prevent a situation where a short event is missed because of the system clock rate. This register has no effect on PFx pins that are defined as outputs.

The contents of this register are cleared at reset, defaulting to level sensitivity.
GPIO Port F Registers (MMRs)

GPIO Port F Interrupt Sensitivity Register (PORTFIO_EDGE)
For all bits, 0 - Level, 1 - Edge

Figure 14-16. GPIO Port F Interrupt Sensitivity Register

GPIO Port F Set on Both Edges (PORTFIO_BOTH) Register

The GPIO set on both edges register (PORTFIO_BOTH) shown in Figure 14-17 is used to enable interrupt generation on both rising and falling edges.

When a given PFx pin has been set to edge-sensitive in the GPIO interrupt sensitivity register, setting the PFx pin’s bit in the GPIO set on both edges register to both edges results in an interrupt being generated on both the rising and falling edges. This register has no effect on PFx pins that are defined as level-sensitive or as outputs.
General-Purpose Input/Output Port F

GPIO Port F Input Enable (PORTFIO_INEN) Register

The GPIO input enable register (PORTFIO_INEN) shown in Figure 14-18 is used to enable the input buffers on any GPIO pin that is being used as an input. Leaving the input buffer disabled eliminates the need for pull-ups and pull-downs when a particular PFx pin is not used in the system. By default, the input buffers are disabled.

If the PFx pin is being used as an input, the corresponding bit in the PORTFIO_INEN register must be set. Otherwise, changes at the GPIO pins will not be recognized by the processor.
The PFx pins are synchronized to the system clock (SCLK). When configured as outputs, the programmable GPIOs can transition once every system clock cycle.

When configured as inputs, the overall system design should take into account the potential latency between the core and system clocks. Changes in the state of PFx pins have a latency of 3 SCLK cycles before being detectable by the processor. When configured for level-sensitive interrupt generation, there is a minimum latency of 4 SCLK cycles between the time the GPIO is asserted and the time that program flow is interrupted. When configured for edge-sensitive interrupt generation, an additional SCLK cycle of latency is introduced, giving a total latency of 5 SCLK cycles between the time the edge is asserted and the time that the core program flow is interrupted.
15 GENERAL-PURPOSE INPUT/OUTPUT PORTS C, D, E

The ADSP-BF538/ADSP-BF538F processors have an extensive set of peripherals, all of which may not be used in an application. A GPIO (general-purpose input/output) function is multiplexed with many of the peripheral pins. GPIO functionality may be enabled on a per-pin basis in lieu of peripheral functionality.

GPIO pins are grouped onto ports C through E. Each pin within a group is individually programmable. If an application’s peripheral implementation does not require all of its pins, the remaining pins may be configured as GPIO. Table 15-1 shows how the peripherals are mapped to the GPIO ports.

Register nomenclature for the GPIO ports use a prefix of PORTxIO, where x can be C, D, or E.

GPIO functionality on ports C, D, and E differs from port F in two ways.

1. Interrupt capabilities are not associated with GPIO pins on ports C, D, and E.

2. GPIO enable and control on ports C, D, and E are not associated with reads and writes of any peripheral registers.

Following a system reset, all GPIO capability is disabled and a pin’s functionality matches the peripheral pin’s functionality. Therefore, out of reset, a peripheral pin with GPIO capability functions as if the pin was dedicated as a peripheral pin with no GPIO multiplexed functionality.
A pin can be configured to be a GPIO by writing the corresponding bit in its GPIO configuration register \texttt{(PORTxIO\_FER)}. Once configured as a GPIO, the pin’s direction can be set using the GPIO direction registers \texttt{(PORTxIO\_DIR)}. It is permissible to pre-set the GPIO output data value before setting the GPIO direction as an output.

Table 15-1. Peripheral Multiplexing

<table>
<thead>
<tr>
<th>Port</th>
<th>Primary Function After Reset</th>
<th>Alternative Function GPIO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port C</td>
<td>CAN</td>
<td>PC 0:1</td>
</tr>
<tr>
<td></td>
<td>GPIO</td>
<td>PC 4:9</td>
</tr>
<tr>
<td>Port D</td>
<td>SPI1</td>
<td>PD 0:4</td>
</tr>
<tr>
<td></td>
<td>SPI2</td>
<td>PD 5:9</td>
</tr>
<tr>
<td></td>
<td>UART1</td>
<td>PD 10:11</td>
</tr>
<tr>
<td></td>
<td>UART2</td>
<td>PD 12:13</td>
</tr>
<tr>
<td>Port E</td>
<td>SPORT2</td>
<td>PE 0:7</td>
</tr>
<tr>
<td></td>
<td>SPORT3</td>
<td>PE 8:15</td>
</tr>
</tbody>
</table>

There are a number of methods for controlling a GPIO’s output data value through register writes. These registers include GPIO data \texttt{(PORTxIO)}, data set \texttt{(PORTxIO\_SET)}, data clear \texttt{(PORTxIO\_CLEAR)}, and data toggle \texttt{(PORTxIO\_TOGGLE)}. These data output control methods eliminate any coherency issues normally associated with a read-modify-write sequence.

Tables Table 15-2, Table 15-3, and Table 15-4 list all of the peripheral pins which have GPIO capabilities. Each GPIO port has a complete set of registers associated with its control. The bit identified in these tables should be used when making accesses to any of the GPIO registers, as this relative bit position holds true for all the GPIO registers for that port. For example, if the application doesn’t use the CAN controller, the two CAN pins (\texttt{CANTX} and \texttt{CANRX}) can be freed as GPIO by setting bits 0 and 1 in
PORTCIO_FER. From that point forward, software can configure the pins independently and control the values driven on them by making writes to bits 0 and 1 in the associated port C GPIO registers.

Table 15-2. GPIO Port C Multiplexed Functionality Pin List

<table>
<thead>
<tr>
<th>GPIO Port C Pin</th>
<th>Peripheral Pin Multiplexed with GPIO Pin</th>
<th>Associated Bit in Port C GPIO Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 0</td>
<td>CAN Transmit (CANTX)</td>
<td>Bit 0</td>
</tr>
<tr>
<td>PC 1</td>
<td>CAN Receive (CANRX)</td>
<td>Bit 1</td>
</tr>
<tr>
<td>PC 4–9</td>
<td>GPIO</td>
<td>Bits 4–9</td>
</tr>
</tbody>
</table>

Table 15-3. GPIO Port D Multiplexed Functionality Pin List

<table>
<thead>
<tr>
<th>GPIO Port D Pin</th>
<th>Peripheral Pin Multiplexed with GPIO Pin</th>
<th>Associated Bit in Port D GPIO Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>PD 0</td>
<td>SPI1 Master Out Slave In (MOSI1)</td>
<td>Bit 0</td>
</tr>
<tr>
<td>PD 1</td>
<td>SPI1 Master In Slave Out (MISO1)</td>
<td>Bit 1</td>
</tr>
<tr>
<td>PD 2</td>
<td>SPI1 Clock (SCK1)</td>
<td>Bit 2</td>
</tr>
<tr>
<td>PD 3</td>
<td>SPI1 Slave Select Input (SPI1SS)</td>
<td>Bit 3</td>
</tr>
<tr>
<td>PD 4</td>
<td>SPI1 Slave Select Enable (SPI1SELECT)</td>
<td>Bit 4</td>
</tr>
<tr>
<td>PD 5</td>
<td>SPI2 Master Out Slave In (MOSI2)</td>
<td>Bit 5</td>
</tr>
<tr>
<td>PD 6</td>
<td>SPI2 Master In Slave Out (MISO2)</td>
<td>Bit 6</td>
</tr>
<tr>
<td>PD 7</td>
<td>SPI2 Clock (SCK2)</td>
<td>Bit 7</td>
</tr>
<tr>
<td>PD 8</td>
<td>SPI2 Slave Select Input (SPI2SS)</td>
<td>Bit 8</td>
</tr>
<tr>
<td>PD 9</td>
<td>SPI2 Slave Select Enable (SPI2SELECT)</td>
<td>Bit 9</td>
</tr>
<tr>
<td>PD 10</td>
<td>UART1 Receive (RX1)</td>
<td>Bit 10</td>
</tr>
<tr>
<td>PD 11</td>
<td>UART1 Transmit (TX1)</td>
<td>Bit 11</td>
</tr>
<tr>
<td>PD 12</td>
<td>UART2 Receive (RX2)</td>
<td>Bit 12</td>
</tr>
<tr>
<td>PD 13</td>
<td>UART2 Transmit (TX2)</td>
<td>Bit 13</td>
</tr>
</tbody>
</table>
GPIO Memory-Mapped Registers (MMRs)

Table 15-4. GPIO Port E Multiplexed Functionality Pin List

<table>
<thead>
<tr>
<th>GPIO Port E Pin</th>
<th>Peripheral Pin Multiplexed with GPIO Pin</th>
<th>Associated Bit in Port E GPIO Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>PE 0</td>
<td>SPORT2 Receive Serial Clock (RSCLK2)</td>
<td>Bit 0</td>
</tr>
<tr>
<td>PE 1</td>
<td>SPORT2 Receive Frame Sync (RFS2)</td>
<td>Bit 1</td>
</tr>
<tr>
<td>PE 2</td>
<td>SPORT2 Receive Data Primary (DR2PRI)</td>
<td>Bit 2</td>
</tr>
<tr>
<td>PE 3</td>
<td>SPORT2 Receive Data Secondary (DR2SEC)</td>
<td>Bit 3</td>
</tr>
<tr>
<td>PE 4</td>
<td>SPORT2 Transmit Serial Clock (TSCLK2)</td>
<td>Bit 4</td>
</tr>
<tr>
<td>PE 5</td>
<td>SPORT2 Transmit Frame Sync (TFS2)</td>
<td>Bit 5</td>
</tr>
<tr>
<td>PE 6</td>
<td>SPORT2 Transmit Data Primary (DT2PRI)</td>
<td>Bit 6</td>
</tr>
<tr>
<td>PE 7</td>
<td>SPORT2 Transmit Data Secondary (DT2SEC)</td>
<td>Bit 7</td>
</tr>
<tr>
<td>PE 8</td>
<td>SPORT3 Receive Serial Clock (RSCLK3)</td>
<td>Bit 8</td>
</tr>
<tr>
<td>PE 9</td>
<td>SPORT3 Receive Frame Sync (RFS3)</td>
<td>Bit 9</td>
</tr>
<tr>
<td>PE 10</td>
<td>SPORT3 Receive Data Primary (DR3PRI)</td>
<td>Bit 10</td>
</tr>
<tr>
<td>PE 11</td>
<td>SPORT3 Receive Data Secondary (DR3SEC)</td>
<td>Bit 11</td>
</tr>
<tr>
<td>PE 12</td>
<td>SPORT3 Transmit Serial Clock (TSCLK3)</td>
<td>Bit 12</td>
</tr>
<tr>
<td>PE 13</td>
<td>SPORT3 Transmit Frame Sync (TFS3)</td>
<td>Bit 13</td>
</tr>
<tr>
<td>PE 14</td>
<td>SPORT3 Transmit Data Primary (DT3PRI)</td>
<td>Bit 14</td>
</tr>
<tr>
<td>PE 15</td>
<td>SPORT3 Transmit Data Secondary (DT3SEC)</td>
<td>Bit 15</td>
</tr>
</tbody>
</table>

GPIO Memory-Mapped Registers (MMRs)

These registers are part of the system memory-mapped registers (MMRs). The addresses of the GPIO MMRs appear in Appendix B, “System MMR Assignments”. Core access to the GPIO registers is through the system bus.
GPIO Function Enable (PORTxIO_FER) Register

The GPIO function enable register (PORTxIO_FER) is a read-write register. Each bit position corresponds to a GPIO pin. The reset value of these registers is 0x0000. In this state, all GPIO capable pins are disabled and function per their respective peripheral pin definitions. A logic one written to a bit position disables the peripheral pin’s definition and enables the pin as a GPIO. A logic 0 returns the pin’s function to the peripheral pin’s definition. A read of unused PORTxIO_FER bits always returns a value of 0, while a write of unused bits has no effect.

Note that in Figure 15-1:

- The PC1 bit (bit 1) can be configured as GPIO, but it is a 5V-tolerant input pin that can provide open-drain functionality when configured as an output.
- The PC4 bit (bit 4) can be configured as GPIO, but it is a 5V-tolerant input pin that can provide open-drain functionality when configured as an output.
- The PC4 through PC9 bits (bits 4 through 9) are always configured as GPIO, regardless of whether these bits are set or cleared.

Figure 15-1. GPIO Port C Function Enable Register
GPIO Memory-Mapped Registers (MMRs)

GPIO Port D Function Enable Register (PORTDIO_FER)
For all bits, 0 - peripheral pin function, 1 - enable GPIO mode

Address = 0xFFC01504

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Reset = 0x0000

Figure 15-2. GPIO Port D Function Enable Register

GPIO Port E Function Enable Register (PORTEIO_FER)
For all bits, 0 - peripheral pin function, 1 - enable GPIO mode

Address = 0xFFC01508

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Reset = 0x0000

Figure 15-3. GPIO Port E Function Enable Register

GPIO Direction (PORTxIO_DIR) Register

The GPIO direction register is a read-write register. Each bit position corresponds to a GPIO pin. If a pin is configured as a GPIO (PORTxIO_FER), a logic 1 enables the GPIO pin as an output, driving the state contained in
the associated PORTxIO register. A logic 0 configures the GPIO pin as an input. The reset value is 0x0000, making all GPIO pins inputs by default when enabled as GPIO. A read of unused PORTxIO_DIR bits always returns a value of 0, while a write of unused bits has no effect.

Note that in Figure 15-4:

- The PC1 bit (bit 1) can be configured as GPIO, but it is a 5V-tolerant input pin that can provide open-drain functionality when configured as an output. This bit can be driven low by application code but will three-state when driven high. An external pull-up resistor should be connected to this bit if full GPIO functionality is desired.

- The PC4 bit (bit 4) can be configured as GPIO, but it is a 5V-tolerant input pin that can provide open-drain functionality when configured as an output. This bit can be driven low by application code but will three-state when driven high. An external pull-up resistor should be connected to this bit if full GPIO functionality is desired.

Figure 15-4. GPIO Port C Direction Register (PORTCIO_DIR)

For all bits, 0 - input, 1 - output

Address = 0xFFC01550

Reset = 0x0000

PC9 Direction
PC8 Direction
PC7 Direction
PC6 Direction
PC0 Direction
PC1 Direction
PC4 Direction
PC5 Direction

Figure 15-4. GPIO Port C Direction Register
### GPIO Port D Direction Register (PORTDIO_DIR)
For all bits, 0 - input, 1 - output

![GPIO Port D Direction Register](image)

<table>
<thead>
<tr>
<th>Bit</th>
<th>PD0 Direction</th>
<th>PD1 Direction</th>
<th>PD2 Direction</th>
<th>PD3 Direction</th>
<th>PD4 Direction</th>
<th>PD5 Direction</th>
<th>PD6 Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0000000000</td>
<td>0000000000</td>
<td>0000000000</td>
<td>0000000000</td>
<td>0000000000</td>
<td>0000000000</td>
<td>0000000000</td>
</tr>
</tbody>
</table>

Reset = 0x0000
Address = 0xFFC01554

---

### GPIO Port E Direction Register (PORTEIO_DIR)
For all bits, 0 - input, 1 - output

![GPIO Port E Direction Register](image)

<table>
<thead>
<tr>
<th>Bit</th>
<th>PE0 Direction</th>
<th>PE1 Direction</th>
<th>PE2 Direction</th>
<th>PE3 Direction</th>
<th>PE4 Direction</th>
<th>PE5 Direction</th>
<th>PE6 Direction</th>
<th>PE7 Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0000000000</td>
<td>0000000000</td>
<td>0000000000</td>
<td>0000000000</td>
<td>0000000000</td>
<td>0000000000</td>
<td>0000000000</td>
<td>0000000000</td>
</tr>
</tbody>
</table>

Reset = 0x0000
Address = 0xFFC01558

---

Figure 15-5. GPIO Port D Direction Register

Figure 15-6. GPIO Port E Direction Register
General-Purpose Input/Output Ports C, D, E

GPIO Input Enable (PORTxIO_INEN) Register

The GPIO input enable register is used to enable the input buffers on any GPIO pin that is being used as an input. Leaving the input buffer disabled eliminates the need for pullups and pulldowns when a particular GPIO pin is not used in the system. By default, the input buffers are disabled. A read of unused PORTxIO_INEN bits always returns a value of 0, while a write of unused bits has no effect.

If the GPIO pin is being used as an input, the corresponding bit in the GPIO input enable register must be set.

Note that in Figure 15-7:

- The PC1 input enable bit (bit 1) can be configured as GPIO. This bit is 5V-tolerant when configured as an input.
- The PC4 input enable bit (bit 4) can be configured as GPIO. This bit is 5V-tolerant when configured as an input.

Figure 15-7. GPIO Port C Input Enable Register
GPIO Memory-Mapped Registers (MMRs)

**GPIO Port D Input Enable Register (PORTDIO_INEN)**

For all bits, 0 - input buffer disabled, 1 - input buffer enabled

![GPIO Port D Input Enable Register](image)

Address = 0xFFC01564

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Reset = 0x0000

- PD13 Input Enable
- PD12 Input Enable
- PD11 Input Enable
- PD10 Input Enable
- PD9 Input Enable
- PD8 Input Enable
- PD7 Input Enable
- PD6 Input Enable
- PD5 Input Enable
- PD4 Input Enable
- PD3 Input Enable
- PD2 Input Enable
- PD1 Input Enable
- PD0 Input Enable
- PD6 Input Enable

Figure 15-8. GPIO Port D Input Enable Register

**GPIO Port E Input Enable Register (PORTEIO_INEN)**

For all bits, 0 - input buffer disabled, 1 - input buffer enabled

![GPIO Port E Input Enable Register](image)

Address = 0xFFC01568

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Reset = 0x0000

- PE15 Input Enable
- PE14 Input Enable
- PE13 Input Enable
- PE12 Input Enable
- PE11 Input Enable
- PE10 Input Enable
- PE9 Input Enable
- PE8 Input Enable
- PE7 Input Enable
- PE6 Input Enable
- PE5 Input Enable
- PE4 Input Enable
- PE3 Input Enable
- PE2 Input Enable
- PE1 Input Enable
- PE0 Input Enable

Figure 15-9. GPIO Port E Input Enable Register
GPIO Value Registers

The processor has four GPIO value registers:

- GPIO data register (PORTxIO)
- GPIO set register (PORTxIO_SET)
- GPIO clear register (PORTxIO_CLEAR)
- GPIO toggle register (PORTxIO_TOGGLE)

These registers are used to:

- Sense the value of the GPIO pins defined as inputs
- Specify the state of GPIO pins defined as outputs

Each GPIO pin is represented by a bit in each of the four value registers. Reading any of the GPIO data, GPIO set, GPIO clear, or GPIO toggle registers returns the value of the GPIO pins. The value returned shows the state of the GPIO pins defined as outputs and the sense of GPIO pins defined as inputs.

Reading the GPIO data, GPIO set, GPIO clear, or GPIO toggle register after reset results in 0x0000 because the pins are not enabled, even though they are reset as inputs.

For more information about the GPIO set, GPIO clear, and GPIO toggle registers, see “GPIO Set (PORTxIO_SET), GPIO Clear (PORTxIO_CLEAR), and GPIO Toggle (PORTxIO_TOGGLE) Registers” on page 15-13.
GPIO Data (PORTxIO) Register

When written, the GPIO data register (Figure 15-10) directly specifies a GPIO pin’s state. When read, the register returns the value of the GPIO pins. A read of unused GPIO data bits always returns a value of 0. A write of unused PORTxIO bits has no effect.

GPIO Port C Data Register (PORTCIO)
For all bits, 0 - clear, 1 - set

Address = 0xFFC01510

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Reset = 0x0000

PC9 Data
PC8 Data
PC7 Data
PC6 Data

Figure 15-10. GPIO Port C Data Register

GPIO Port D Data Register (PORTDIO)
For all bits, 0 - clear, 1 - set

Address = 0xFFC01514

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Reset = 0x0000

PD13 Data
PD12 Data
PD11 Data
PD10 Data
PD9 Data
PD8 Data
PD7 Data

Figure 15-11. GPIO Port D Data Register
General-Purpose Input/Output Ports C, D, E

GPIO Set (PORTxIO_SET), GPIO Clear (PORTxIO_CLEAR), and GPIO Toggle (PORTxIO_TOGGLE) Registers

The GPIO set, GPIO clear, and GPIO toggle registers are used to set, clear, or toggle the output state associated with each output GPIO pin.

This mechanism is used to avoid the potential issues with more traditional read-modify-write mechanisms. Reading any of these registers returns the GPIO pin state. A read of unused bits in these registers always returns a value of 0. A write of unused bits has no effect.

Figure 15-13 through Figure 15-18 represent the GPIO set and GPIO clear registers.
GPIO Value Registers

GPIO Port C Set Register (PORTCIO_SET)
For all bits, write-1-to-set

Address = 0xFFFF01530

PC9 Set
PC8 Set
PC7 Set
PC6 Set

Reset = 0x0000

The PC1 and PC4 pins function as an open-drain when configured as an output. Software can drive this pin low by writing a 0 to this bit location. If software writes a 1 to this bit location, then the pins will be three-stated.

Figure 15-13. GPIO Port C Set Register

GPIO Port D Set Register (PORTDIO_SET)
For all bits, 0 - write-1-to-set

Address = 0xFFFF01534

PD13 Set
PD12 Set
PD11 Set
PD10 Set
PD9 Set
PD8 Set
PD7 Set

PD0 Set
PD1 Set
PD2 Set
PD3 Set
PD4 Set
PD5 Set
PD6 Set

Reset = 0x0000

Figure 15-14. GPIO Port D Set Register
General-Purpose Input/Output Ports C, D, E

GPIO E Set Register (PORTEIO_SET)
For all bits, write-1-to-set

Address = 0xFFC01538

<table>
<thead>
<tr>
<th>Bit 15</th>
<th>Bit 14</th>
<th>Bit 13</th>
<th>Bit 12</th>
<th>Bit 11</th>
<th>Bit 10</th>
<th>Bit 9</th>
<th>Bit 8</th>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Reset = 0x0000

PE15 Set
PE14 Set
PE13 Set
PE12 Set
PE11 Set
PE10 Set
PE9 Set
PE8 Set

PE0 Set
PE1 Set
PE2 Set
PE3 Set
PE4 Set
PE5 Set
PE6 Set
PE7 Set

Figure 15-15. GPIO Port E Set Register

GPIO Port C Clear Register (PORTCIO_CLEAR)
For all bits, write-1-to-clear

Address = 0xFFC01520

<table>
<thead>
<tr>
<th>Bit 15</th>
<th>Bit 14</th>
<th>Bit 13</th>
<th>Bit 12</th>
<th>Bit 11</th>
<th>Bit 10</th>
<th>Bit 9</th>
<th>Bit 8</th>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Reset = 0x0000

PC9 Clear
PC8 Clear
PC7 Clear
PC6 Clear

PC0 Clear
PC1 Clear
PC4 Clear
PC5 Clear

The PC1 and PC4 pins function as an open-drain when configured as an output. Software can drive this pin low by writing a 0 to this bit location. If software writes a 1 to this bit location, then the pins will be three-stated.

Figure 15-16. GPIO Port C Clear Register
GPIO Value Registers

GPIO Port D Clear Register (PORTDIO_CLEAR)
For all bits, write-1-to-clear

Address = 0xFFC01524

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Reset = 0x0000

PD13 Clear
PD12 Clear
PD11 Clear
PD10 Clear
PD9 Clear
PD8 Clear
PD7 Clear

Figure 15-17. GPIO Port D Clear Register

GPIO Port E Clear Register (PORTEIO_CLEAR)
For all bits, write-1-to-clear

Address = 0xFFC01528

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Reset = 0x0000

PE15 Clear
PE14 Clear
PE13 Clear
PE12 Clear
PE11 Clear
PE10 Clear
PE9 Clear
PE8 Clear

Figure 15-18. GPIO Port E Clear Register
**General-Purpose Input/Output Ports C, D, E**

**Figure 15-19, Figure 15-20, and Figure 15-21** show the GPIO toggle registers.

**GPIO Port C Toggle Register (PORTCIO_TOGGLE)**
For all bits, write-1-to-toggle

<table>
<thead>
<tr>
<th>Address = 0xFFC01540</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>Reset = 0x0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC9 Toggle</td>
<td>0</td>
<td>PC0 Toggle</td>
</tr>
<tr>
<td>PC8 Toggle</td>
<td>0</td>
<td>PC1 Toggle</td>
</tr>
<tr>
<td>PC7 Toggle</td>
<td>0</td>
<td>PC4 Toggle</td>
</tr>
<tr>
<td>PC6 Toggle</td>
<td>0</td>
<td>PC5 Toggle</td>
</tr>
</tbody>
</table>

The PC1 and PC4 pins function as an open-drain when configured as an output. Software can drive this pin low by writing a 0 to this bit location. If software writes a 1 to this bit location, then the pins will be three-stated.

**Figure 15-19. GPIO Port C Toggle Register**

**GPIO Port D Toggle Register (PORTDIO_TOGGLE)**
For all bits, write-1-to-toggle

<table>
<thead>
<tr>
<th>Address = 0xFFC01544</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>Reset = 0x0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>PD13 Toggle</td>
<td>0</td>
<td>PD0 Toggle</td>
</tr>
<tr>
<td>PD12 Toggle</td>
<td>0</td>
<td>PD1 Toggle</td>
</tr>
<tr>
<td>PD11 Toggle</td>
<td>0</td>
<td>PD2 Toggle</td>
</tr>
<tr>
<td>PD10 Toggle</td>
<td>0</td>
<td>PD3 Toggle</td>
</tr>
<tr>
<td>PD9 Toggle</td>
<td>0</td>
<td>PD4 Toggle</td>
</tr>
<tr>
<td>PD8 Toggle</td>
<td>0</td>
<td>PD5 Toggle</td>
</tr>
<tr>
<td>PD7 Toggle</td>
<td>0</td>
<td>PD6 Toggle</td>
</tr>
</tbody>
</table>

**Figure 15-20. GPIO Port D Toggle Register**
GPIO Port E Toggle Register (PORTEIO_TOGGLE)
For all bits, write-1-to-toggle

Figure 15-21. GPIO Port E Toggle Register

As an example of how these registers work, assume that PD0 is configured as an output (PORTDIO_FER = 0x0001 and PORTDIO_DIR = 0x0001). Writing 0x0001 to the PORTDIO_SET register drives a logic 1 on the PD0 pin without affecting the state of any other GPIO pins. Writing 0x0001 to the PORTDIO_CLEAR register drives a logic 0 on the PD0 pin without affecting the state of any other GPIO pins. Writing a 0x0001 to the PORTDIO_TOGGLE register changes the pin state on PD0 from logic zero to logic one or from logic one to logic zero, depending upon the existing pin state, without affecting the state of any other GPIO pins.

Writing a 0 to one of these registers has no effect on the value of the PD0 pin, and is therefore ignored.

Reading the GPIO set or GPIO clear register returns:

- 0 for GPIO pins defined as outputs and driven low
- 1 for pins (including PD0 in the example above) defined as outputs and driven high
- The present sense of GPIO pins defined as inputs
Performance/Throughput

The GPIO pins are synchronized to the system clock (SCLK). When configured as outputs, the GPIOs can transition once every system clock cycle.
Performance/Throughput
The processor features three identical 32-bit general-purpose timers, a core timer, and a watchdog timer.

The general-purpose timers can be individually configured in any of three modes:

- Pulse-width modulation (PWM_OUT) mode
- Pulse width count and capture (WDTH_CAP) mode
- External event (EXT_CLK) mode

The core timer is available to generate periodic interrupts for a variety of system timing functions.

The watchdog timer can be used to implement a software watchdog function. A software watchdog can improve system availability by generating an event to the Blackfin processor core if the timer expires before being updated by software.

**General-Purpose Timers**

Each general-purpose timer has one dedicated bidirectional chip pin, TMRx. This pin functions as an output pin in the PWM_OUT mode and as an input pin in the WDTH_CAP and EXT_CLK modes. To provide these functions, each timer has four registers. For range and precision, the timer counter (TIMERx_COUNTER), timer period (TIMERx_PERIOD), and timer pulse width (TIMERx_WIDTH) registers are 32 bits wide. See Figure 16-1.
The registers for each general-purpose timer are:

- Timer configuration (*TIMERx_CONFIG*) registers
- Timer counter (*TIMERx_COUNTER*) registers
- Timer period (*TIMERx_PERIOD*) registers
- Timer pulse width (*TIMERx_WIDTH*) registers

When clocked internally, the clock source is the processor’s peripheral clock (*SCLK*). Assuming the peripheral clock is running at 133 MHz, the maximum period for the timer count is 

\[
\frac{2^{32} - 1}{133 \text{ MHz}} = 32.2 \text{ seconds}
\]
The timer enable (\texttt{TIMER\_ENABLE}) register can be used to enable all three
timers simultaneously. The register contains three “write-1-to-set” control
bits, one for each timer. Correspondingly, the timer disable
(\texttt{TIMER\_DISABLE}) register contains three “write-1-to-clear” control bits to
allow simultaneous or independent disabling of the three timers. Either
the timer enable or the timer disable register can be read back to check the
enable status of the timers. A 1 indicates that the corresponding timer is
enabled. The timer starts counting three \texttt{SCLK} cycles after the \texttt{TIMENx} bit is
set.

The timer status (\texttt{TIMER\_STATUS}) register contains an interrupt latch bit
(\texttt{TIMILx}) and an overflow/error indicator bit (\texttt{TOVF\_ERRx}) for each timer.
These sticky bits are set by the timer hardware and may be polled by soft-
ware. They need to be cleared by software explicitly, by writing a 1 to the
bit.

To enable a timer’s interrupts, set the \texttt{IRQ\_ENA} bit in the timer’s configura-
tion (\texttt{TIMER\_CONFIG}) register and unmask the timer’s interrupt by setting
the corresponding bits of the \texttt{IMASK} and \texttt{SIC\_IMASKx} registers. With the
\texttt{IRQ\_ENA} bit cleared, the timer does not set its timer interrupt latch
(\texttt{TIMILx}) bits. To poll the \texttt{TIMILx} bits without permitting a timer inter-
rupt, programs can set the \texttt{IRQ\_ENA} bit while leaving the timer’s interrupt
masked.

With interrupts enabled, make sure that the interrupt service routine
(ISR) clears the \texttt{TIMILx} latch before the \texttt{RTI} instruction, to ensure that the
interrupt is not reissued. To make sure that no timer event is missed, the
latch should be reset at the very beginning of the interrupt routine when
in external clock (\texttt{EXT\_CLK}) mode. To enable timer interrupts, set the
\texttt{IRQ\_ENA} bit in the proper timer configuration (\texttt{TIMER\_CONFIG}) register.
Timer Registers

The timer peripheral module provides general-purpose timer functionality. It consists of three identical timer units.

Each timer provides four registers:

- \texttt{TIMERx\_CONFIG[15:0]} – Timer configuration register
- \texttt{TIMERx\_WIDTH[31:0]} – Timer pulse width register
- \texttt{TIMERx\_PERIOD[31:0]} – Timer period register
- \texttt{TIMERx\_COUNTER[31:0]} – Timer counter register

Three registers are shared between the three timers:

- \texttt{TIMER\_ENABLE[15:0]} – Timer enable register
- \texttt{TIMER\_DISABLE[15:0]} – Timer disable register
- \texttt{TIMER\_STATUS[15:0]} – Timer status register

The size of accesses is enforced. A 32-bit access to a timer configuration register or a 16-bit access to a timer pulse width, timer period, or timer counter register results in a memory-mapped register (MMR) error. Both 16- and 32-bit accesses are allowed for the timer enable, timer disable, and timer status registers. On a 32-bit read, the upper word returns all 0s.

\textbf{TIMER\_ENABLE Register}

The timer enable register (\texttt{TIMER\_ENABLE}) allows all three timers to be enabled simultaneously in order to make them run completely synchronously. (See Figure 16-2.) For each timer there is a single W1S control bit. Writing a 1 enables the corresponding timer; writing a 0 has no effect. The three bits can be set individually or in any combination. A read of the...
timer enable register shows the status of the enable for the corresponding
timer. A 1 indicates that the timer is enabled. All unused bits return 0
when read.

**Timer Enable Register (TIMER_ENABLE)**

```
| 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | Reset = 0x0000 |
```

**TIMER_DISABLE Register**

The timer disable register (TIMER_DISABLE) allows all three timers to be
disabled simultaneously. (See Figure 16-3.) For each timer there is a single
W1C control bit. Writing a 1 disables the corresponding timer; writing a
0 has no effect. The three bits can be cleared individually or in any combi-
nation. A read of the timer disable register returns a value identical to a
read of the timer enable register. A 1 indicates that the timer is enabled.
All unused bits return 0 when read.

In PWM_OUT mode, a write of a 1 to TIMER_DISABLE does not stop the corre-
sponding timer immediately. Rather, the timer continues running and
stops cleanly at the end of the current period (if PERIOD_CNT = 1) or pulse
(if PERIOD_CNT = 0). If necessary, the processor can force a timer in
PWM_OUT mode to stop immediately by first writing a 1 to the corre-
sponding bit in TIMER_DISABLE, and then writing a 1 to the corre-
sponding TRUNx bit in TIMER_STATUS. See “Stopping the Timer in
PWM_OUT Mode” on page 16-19.
Timer Registers

**Timer Disable Register (TIMER_DISABLE)**

![Timer Disable Register](image)

- *TIMDIS0 (Timer0 Disable)*
  - 1 - Disable timer
  - Read as 1 if this timer is enabled
- *TIMDIS1 (Timer1 Disable)*
  - 1 - Disable timer
  - Read as 1 if this timer is enabled
- *TIMDIS2 (Timer2 Disable)*
  - 1 - Disable timer
  - Read as 1 if this timer is enabled

*Figure 16-3. Timer Disable Register*

In WDTH_CAP and EXT_CLK modes, a write of a 1 to TIMER_DISABLE stops the corresponding timer immediately.

**TIMER_STATUS Register**

The timer status register (TIMER_STATUS) indicates the status of all three timers and is used to check the status of all three timers with a single read. (See Figure 16-4.) The status bits are sticky and W1C. The TRUNx bits can clear themselves, which they do when a PWM_OUT mode timer stops at the end of a period. During a status register read access, all reserved or unused bits return a 0.

Each timer generates a unique interrupt request signal, which is gated by the corresponding IRQ_ENA bit in the TIMERx_CONFIG register. The shared timer status register (TIMER_STATUS) latches these interrupts so the user can determine the interrupt source without reference to the unique interrupt signal (for example, in the case where all three timers have been assigned to the same interrupt priority). Interrupt bits are sticky and must be cleared by the interrupt service routine (ISR) to assure that the interrupt is not reissued.
The TIMILx bits work along with the IRQ_ENA bit of the timer configuration register to indicate interrupt requests. If an interrupt condition or error occurs and IRQ_ENA is set, then the TIMILx bit is set and the interrupt to the core is asserted. This interrupt may be masked by the system interrupt controllers. If an interrupt condition or error occurs and IRQ_ENA is cleared, then the TIMILx bit is not set and the interrupt is not asserted. If TIMILx is already set and IRQ_ENA is written to 0, TIMILx stays set and the interrupt stays asserted. See Figure 16-24.

Timer Status Register (TIMER_STATUS)

![Figure 16-4. Timer Status Register](image-url)

The read value of the TRUNx bits reflects the timer slave enable status in all modes—TRUNx set indicates running and TRUNx cleared indicates stopped. While reading the TIMENx or TIMDISx bits in the TIMER_ENABLE and TIMER_DISABLE registers will reflect whether a timer is enabled, the TRUNx
Timer Registers

bits indicate whether the timer is actually running. In WDTH_CAP and EXT_CLK modes, reads from TIMENx and TRUNx always return the same value.

A W1C operation to the Timer_DISABLE register disables the corresponding timer in all modes. In PWM_OUT mode, a disabled timer continues running until the ongoing period (PERIOD_CNT = 1) or pulse (PERIOD_CNT = 0) completes. During this final period the TIMENx bit returns 0, but the TRUNx bit still reads as a 1. See Figure 16-10 on page 16-14. In this state only, TRUNx becomes a W1C bit. During this final period with the timer disabled, writing a 1 to TRUNx clears TRUNx and stops the timer immediately without waiting for the timer counter to reach the end of its current cycle.

Writing the TRUNx bits has no effect in other modes or when a timer has not been enabled. Writing the TRUNx bits to 1 in PWM_OUT mode has no effect on a timer that has not first been disabled.

TIMERx_CONFIG Registers

The operating mode for each timer is specified by its timer configuration register (TIMERx_CONFIG), as shown in Figure 16-5. The TIMERx_CONFIG register may be written only when the timer is not running. After disabling the timer in PWM_OUT mode, make sure the timer has stopped running by checking its TRUNx bit in TIMER_STATUS before attempting to reprogram TIMERx_CONFIG. The TIMERx_CONFIG registers may be read at any time. The ERR_TYP field is read-only. It is cleared at reset and when the timer is enabled. Each time TOVF_ERRx is set, ERR_TYP[1:0] is loaded with a code that identifies the type of error that was detected. This value is held until the next error or timer enable occurs. For an overview of error conditions, see Table 16-1. The TIMERx_CONFIG register also controls the behavior of the TMRx pin, which becomes an output in PWM_OUT mode (TMODE = 01) when the OUT_DIS bit is cleared.
These read-only registers retain their state when disabled. (See Figure 16-6.) When enabled, the timer counter register (TIMERx_COUNTER) is reinitialized by hardware based on configuration and mode. The timer counter register may be read at any time (whether the timer is running or stopped), and it returns a coherent 32-bit value. Depending on the operation mode, the incrementing counter can be clocked by four different sources: SCLK, the TMRx pin, the GPIO port F pin PF1, or the parallel port clock PPI_CLK.
Timer Registers

While the processor core is being accessed by an external emulator debugger, all code execution stops. By default, the TIMERx_COUNTER also halts its counting during an emulation access in order to remain synchronized with the software. While stopped, the count does not advance—in PWM_OUT mode, the TMRx pin waveform is “stretched”; in WDTH_CAP mode, measured values are incorrect; in EXT_CLK mode, input events on TMRx may be missed. All other timer functions such as register reads and writes, interrupts previously asserted (unless cleared), and the loading of TIMERx_PERIOD and TIMERx_WIDTH in WDTH_CAP mode remain active during an emulation stop.

Some applications may require the timer to continue counting asynchronously to the emulation-halted processor core. Set the EMU_RUN bit in TIMERx_CONFIG to enable this behavior.

Timer Counter Registers (TIMERx_COUNTER)

![Timer Counter Registers](image)

**TIMERx_PERIOD and TIMERx_WIDTH Registers**

When a timer is enabled and running, and the software writes new values to the TIMERx.PERIOD register and the timer pulse width register, the writes are buffered and do not update the registers until the end of the current period (when the timer counter register equals the TIMERx.PERIOD register).
Usage of the TIMERx_PERIOD register and TIMERx_WIDTH register (see Figure 16-7 and Figure 16-8) varies depending on the mode of the timer:

- In pulse-width modulation mode (PWM_OUT), both the TIMERx_PERIOD and TIMERx_WIDTH register values can be updated “on-the-fly” since the timer period and timer pulse width (duty cycle) register values change simultaneously.

- In pulse width and period capture mode (WDTH_CAP), the timer period and timer pulse width buffer values are captured at the appropriate time. The TIMERx_PERIOD and TIMERx_WIDTH registers are then updated simultaneously from their respective buffers. Both registers are read-only in this mode.

- In external event capture mode (EXT_CLK), the TIMERx_PERIOD register is writable and can be updated “on-the-fly.” The TIMERx_WIDTH register is not used.

If new values are not written to the TIMERx_PERIOD register or the TIMERx_WIDTH register, the value from the previous period is reused. Writes to the 32-bit TIMERx_PERIOD register and TIMERx_WIDTH register are atomic; it is not possible for the high word to be written without the low word also being written.

Values written to the TIMERx_PERIOD registers or TIMERx_WIDTH registers are always stored in the buffer registers. Reads from the TIMERx_PERIOD or TIMERx_WIDTH registers always return the current, active value of period or pulse width. Written values are not read back until they become active. When the timer is enabled, they do not become active until after the TIMERx_PERIOD and TIMERx_WIDTH registers are updated from their respective buffers at the end of the current period. See Figure 16-1 on page 16-2.

When the timer is disabled, writes to the buffer registers are immediately copied through to the TIMERx_PERIOD or TIMERx_WIDTH register so that they will be ready for use in the first timer period. For example, to change
the values for the `TIMERx_PERIOD` and/or `TIMERx_WIDTH` registers in order to use a different setting for each of the first three timer periods after the timer is enabled, the procedure to follow is:

1. Program the first set of register values.
2. Enable the timer.
3. Immediately program the second set of register values.
4. Wait for the first timer interrupt.
5. Program the third set of register values.

Each new setting is then programmed when a timer interrupt is received.

In `PWM_OUT` mode with very small periods (less than 10 counts), there may not be enough time between updates from the buffer registers to write both the `TIMERx_PERIOD` register and the `TIMERx_WIDTH` register. The next period may use one old value and one new value. In order to prevent pulse width >= period errors, write the `TIMERx_WIDTH` register before the `TIMERx_PERIOD` register when decreasing the values, and write the `TIMERx_PERIOD` register before the `TIMERx_WIDTH` register when increasing the value.

**Figure 16-7. Timer Period Registers**
Using the Timer

To enable an individual timer, set that timer’s TIMEN bit in the TIMER_ENABLE register. To disable an individual timer, set that timer’s TIMDIS bit in the TIMER_DISABLE register. To enable all three timers in parallel, set all three TIMEN bits in the TIMER_ENABLE register.

Before enabling a timer, always program the corresponding timer configuration (TIMERx_CONFIG) register. This register defines the timer operating mode, the polarity of the TMRx pin, and the timer interrupt behavior. Do not alter the operating mode while the timer is running.

Examples of timer enable and disable timing appear in Figure 16-9, Figure 16-10, and Figure 16-11.

When timers are disabled, the timer counter registers retain their state; when a timer is re-enabled, the timer counter is reinitialized based on the operating mode. The timer counter registers are read-only. Software cannot overwrite or preset the timer counter value directly.
Using the Timer

Figure 16-9. Timer Enable Timing

Figure 16-10. Timer Disable Timing
Pulse-Width Modulation (PWM_OUT) Mode

Setting the TMODE field to 01 in the timer configuration (TIMERx_CONFIG) register enables PWM_OUT mode. In PWM_OUT mode, the timer TMRx pin is an output. The output can be disabled by setting the OUT_DIS bit in the timer configuration register.

In PWM OUT mode, the bits PULSE_HI, PERIOD_CNT, IRQ_ENA, OUT_DIS, CLK_SEL, EMU_RUN, and TOGGLE_HI enable orthogonal functionality. They may be set individually or in any combination, although some combinations are not useful (such as TOGGLE_HI = 1 with OUT_DIS = 1 or PERIOD_CNT = 0). See Figure 16-12.

Once a timer has been enabled, the timer counter register is loaded with a starting value. If CLK_SEL = 0, the timer counter starts at 0x1. If CLK_SEL = 1, it is reset to 0x0 as in EXT_CLK mode. The timer counts upward to the value of the TIMERx_PERIOD register. For either setting of CLK_SEL, when the timer counter equals the timer period, the timer counter is reset to 0x1 on the next clock.
In **PWM_OUT** mode, the **PERIOD_CNT** bit controls whether the timer generates one pulse or many pulses. When **PERIOD_CNT** is cleared (PWM_OUT single pulse mode), the timer uses the **TIMERx_WIDTH** register, generates one asserting and one deasserting edge, then generates an interrupt (if enabled) and stops. When **PERIOD_CNT** is set (PWM_OUT continuous pulse mode), the timer uses both the **TIMERx_PERIOD** and **TIMERx_WIDTH** registers and
generates a repeating (and possibly modulated) waveform. It generates an interrupt (if enabled) at the end of each period and stops only after it is disabled. A setting of \texttt{PERIOD_CNT} = 0 counts to the end of the width; a setting of \texttt{PERIOD_CNT} = 1 counts to the end of the period.

The \texttt{TIMERx_PERIOD} and \texttt{TIMERx_WIDTH} registers are read-only in some operation modes. Be sure to set the \texttt{TMODE} field in the \texttt{TIMERx_CONFIG} register to b#01 before writing to these registers.

### Output Pad Disable

The output pin can be disabled in \texttt{PWM_OUT} mode by setting the \texttt{OUT_DIS} bit in the timer configuration register. The \texttt{TMRx} pin is then three-stated regardless of the setting of \texttt{PULSE_HI} and \texttt{TOGGLE_HI}. This can reduce power consumption when the output signal is not being used.

### Single Pulse Generation

If the \texttt{PERIOD_CNT} bit is cleared, the \texttt{PWM_OUT} mode generates a single pulse on the \texttt{TMRx} pin. This mode can also be used to implement a precise delay. The pulse width is defined by the \texttt{TIMERx_WIDTH} register, and the \texttt{TIMERx_PERIOD} register is not used.

At the end of the pulse, the timer interrupt latch bit \texttt{TIMILx} gets set, and the timer is stopped automatically. If the \texttt{PULSE_HI} bit is set, an active high pulse is generated on the \texttt{TMRx} pin. If \texttt{PULSE_HI} is not set, the pulse is active low.

### Pulse-Width Modulation Waveform Generation

If the \texttt{PERIOD_CNT} bit is set, the internally clocked timer generates rectangular signals with well-defined period and duty cycle. This mode also generates periodic interrupts for real-time signal processing.
Using the Timer

The 32-bit \((\text{TIMERx}_{-\text{PERIOD}})\) and \((\text{TIMERx}_{-\text{WIDTH}})\) registers are programmed with the values of the timer count period and pulse width modulated output pulse width.

When the timer is enabled in this mode, the \(\text{TMRx}\) pin is pulled to a deasserted state each time the timer counter equals the value of the \(\text{TIMERx}_{-\text{WIDTH}}\) register, and the pin is asserted again when the period expires (or when the timer gets started).

To control the assertion sense of the \(\text{TMRx}\) pin, the \(\text{PULSE}_{-\text{HI}}\) bit in the corresponding \(\text{TIMERx}_{-\text{CONFIG}}\) register is used. For a low assertion level, clear this bit. For a high assertion level, set this bit. When the timer is disabled in \(\text{PWM}_{-\text{OUT}}\) mode, the \(\text{TMRx}\) pin is driven to the deasserted level.

If enabled, a timer interrupt is generated at the end of each period. An interrupt service routine (ISR) must clear the interrupt latch bit \((\text{TIMILx})\) and might alter period and/or width values. In pulse-width modulation (PWM) applications, the software needs to update period and pulse width values while the timer is running. When software updates either period or pulse width registers, the new values are held by special buffer registers until the period expires. Then the new period and pulse width values become active simultaneously. New \(\text{TIMERx}_{-\text{PERIOD}}\) and \(\text{TIMERx}_{-\text{WIDTH}}\) register values are written while the old values are being used. The new values are loaded in to be used when the timer counter value equals the current timer period value. Reads from \(\text{TIMERx}_{-\text{PERIOD}}\) and \(\text{TIMERx}_{-\text{WIDTH}}\) registers return the old values until the period expires.

The \(\text{TOVF}_{-\text{ERRx}}\) status bit signifies an error condition in \(\text{PWM}_{-\text{OUT}}\) mode. The \(\text{TOVF}_{-\text{ERRx}}\) bit is set if \(\text{TIMERx}_{-\text{PERIOD}} = 0\) or \(\text{TIMERx}_{-\text{PERIOD}} = 1\) at startup, or when the timer counter register rolls over. It is also set when the timer counter register rolls over if the \(\text{TIMERx}_{-\text{WIDTH}}\) register is greater than or equal to the \(\text{TIMERx}_{-\text{PERIOD}}\) register. The \(\text{ERR}_{-\text{TYP}}\) bits are set when the \(\text{TOVF}_{-\text{ERRx}}\) bit is set.
Timers

To generate the maximum frequency on the TMRx output pin, set the period value to 2 and the pulse width to 1. This makes TMRx toggle each SCLK clock, producing a duty cycle of 50%. The period may be programmed to any value from 2 to \((2^{32} - 1)\), inclusive. The pulse width may be programmed to any value from 1 to \((\text{Period} - 1)\), inclusive. When \(\text{PERIOD_CNT} = 0\), the pulse width may be programmed to any value from 1 to \((2^{32} - 1)\), inclusive.

Although the hardware reports an error if the \(\text{TIMERx_WIDTH}\) value equals the \(\text{TIMERx_PERIOD}\) value, this is still a valid operation to implement PWM patterns with 100% duty cycle. If doing so, software must generally ignore the \(\text{TOVL_ERRx}\) flags. Pulse width values greater than the period value are not recommended. Similarly, \(\text{TIMERx_WIDTH} = 0\) is not a valid operation. Duty cycles of 0% are not supported.

Stopping the Timer in PWM_OUT Mode

In all PWM_OUT mode variants, the timer treats a disable operation (W1C to \(\text{TIMER_DISABLE}\)) as a “stop is pending” condition. When disabled, it automatically completes the current waveform and then stops cleanly. This prevents truncation of the current pulse and unwanted PWM patterns at the TMRx pin. The processor can determine when the timer stops running by polling for the corresponding TRUNx bit in the \(\text{TIMER_STATUS}\) register to read 0 or by waiting for the last interrupt (if enabled). Note the timer cannot be reconfigured (\(\text{TIMERx_CONFIG}\) cannot be written to a new value) until after the timer stops and TRUNx reads 0.

In PWM_OUT single pulse mode (\(\text{PERIOD_CNT} = 0\)), it is not necessary to write \(\text{TIMER_DISABLE}\) to stop the timer. At the end of the pulse, the timer stops automatically, the corresponding bit in \(\text{TIMER_ENABLE}\) (and \(\text{TIMER_DISABLE}\)) is cleared, and the corresponding TRUNx bit is cleared. See Figure 16-11. To generate multiple pulses, write a 1 to \(\text{TIMER_ENABLE}\), wait for the timer to stop, then write another 1 to \(\text{TIMER_ENABLE}\).
Using the Timer

If necessary, the processor can force a timer in PWM_OUT mode to stop immediately. Do this by first writing a 1 to the corresponding bit in TIMER_DISABLE, and then writing a 1 to the corresponding TRUNx bit in TIMER_STATUS. This stops the timer whether the pending stop was waiting for the end of the current period (PERIOD_CNT = 1) or the end of the current pulse width (PERIOD_CNT = 0). This feature may be used to regain immediate control of a timer during an error recovery sequence.

⚠️ Use this feature carefully, because it may corrupt the PWM pattern generated at the TMRx pin.

In PWM_OUT continuous pulse mode (PERIOD_CNT = 1), each timer samples its TIMENx bit at the end of each period. It stops cleanly at the end of the first period when TIMENx is low. This implies (barring any W1C to TRUNx) that a timer that is disabled and then re-enabled all before the end of the current period will continue to run as if nothing happened. Typically, software should disable a PWM_OUT timer and then wait for it to stop itself. The timer will always stop at the end of the first pulse when PERIOD_CNT = 0.

Externally Clocked PWM_OUT

By default, the timer is clocked internally by SCLK. Alternatively, if the CLK_SEL bit in the timer configuration (TIMERx_CONFIG) register is set, then the timer is clocked by PWM_CLK. The PWM_CLK is normally input from the PF1 pin, but may be taken from the PPI_CLK pin when the timers are configured to work with the PPI. Different timers may receive different signals on their PWM_CLK inputs, depending on configuration. As selected by the PERIOD_CNT bit, the PWM_OUT mode either generates pulse-width modulation waveforms or generates a single pulse with pulse width defined by the TIMERx_WIDTH register.

When CLK_SEL is set, the counter resets to 0x0 at startup and increments on each rising edge of PWM_CLK. The TMRx pin transitions on rising edges of PWM_CLK. There is no way to select the falling edges of PWM_CLK. In this mode, the PULSE_HI bit controls only the polarity of the pulses produced.
The timer interrupt may occur slightly before the corresponding edge on the TMRx pin (the interrupt occurs on an SCLK edge, the pin transitions on a later PWM_CLK edge). It is still safe to program new period and pulse width values as soon as the interrupt occurs. After a period expires, the counter rolls over to a value of 0x1.

The PWM_CLK clock waveform is not required to have a 50% duty cycle, but the minimum PWM_CLK clock low time is one SCLK period, and the minimum PWM_CLK clock high time is one SCLK period. This implies the maximum PWM_CLK clock frequency is SCLK/2.

The PF1 pin can only clock the timer when PF1 functions as an input pin. When any timer is in PWM_OUT mode with CLK_SEL = 1 and TIN_SEL = 0, then the PF1 bit in the FIO_DIR register is ignored and PF1 is forced to be an input.

**PULSE_HI Toggle Mode**

The waveform produced in PWM_OUT mode with PERIOD_CNT = 1 normally has a fixed assertion time and a programmable deassertion time (via the TIMERx_WIDTH register). When two timers are running synchronously by the same period settings, the pulses are aligned to the asserting edge as shown in Figure 16-13.

The TOGGLE_HI mode enables control of the timing of both the asserting and deasserting edges of the output waveform produced. The phase between the asserting edges of two timer outputs is programmable. The effective state of the PULSE_HI bit alternates every period. The adjacent active low and active high pulses, taken together, create two halves of a fully arbitrary rectangular waveform. The effective waveform is still active high when PULSE_HI is set and active low when PULSE_HI is cleared. The value of TOGGLE_HI has no effect unless the mode is PWM_OUT and PERIOD_CNT = 1.
In **TOGGLE_HI** mode, when **PULSE_HI** is set, an active low pulse is generated in the first, third, and all odd-numbered periods, and an active high pulse is generated in the second, fourth, and all even-numbered periods. When **PULSE_HI** is cleared, an active high pulse is generated in the first, third, and all odd-numbered periods, and an active low pulse is generated in the second, fourth, and all even-numbered periods.

![Figure 16-13. Timers With Pulses Aligned to Asserting Edge](image)

The deasserted state at the end of one period matches the asserted state at the beginning of the next period, so the output waveform only transitions when $\text{Count} = \text{Pulse Width}$. The net result is an output waveform pulse that repeats every two counter periods and is centered around the end of the first period (or the start of the second period).

**Figure 16-14** shows an example with all three timers running with the same period settings. When software does not alter the PWM settings at runtime, the duty cycle is 50%. The values of the **TIMERx_WIDTH** registers control the phase between the signals.
Similarly, two timers can generate non-overlapping clocks, by center-aligning the pulses while inverting the signal polarity for one of the timers (See Figure 16-15).

Figure 16-15. Two Timers With Non-Overlapping Clocks

---

**Figure 16-14. Three Timers With Same Period Settings**

Similarly, two timers can generate non-overlapping clocks, by center-aligning the pulses while inverting the signal polarity for one of the timers (See Figure 16-15).

**Figure 16-15. Two Timers With Non-Overlapping Clocks**
When TOGGLE_HI = 0, software updates the TIMERx_PERIOD and TIMERx_WIDTH registers once per waveform period. When TOGGLE_HI = 1, software updates the TIMERx_PERIOD and TIMERx_WIDTH registers twice per waveform period with values that are half as large. In odd-numbered periods, write (Period – Width) instead of Width to the TIMERx_WIDTH register in order to obtain center-aligned pulses.

For example, if the pseudo-code when TOGGLE_HI = 0 is:

```c
int period, width;
for (;;) {
    period = generate_period(...);
    width = generate_width(...);

    waitfor (interrupt);

    write(TIMERx_PERIOD, period);
    write(TIMERx_WIDTH, width);
}
```

Then when TOGGLE_HI = 1, the pseudo-code would be:

```c
int period, width;
int per1, per2, wid1, wid2;

for (;;) {
    period = generate_period(...);
    width = generate_width(...);

    per1 = period/2;
    wid1 = width/2;

    per2 = period/2;
    wid2 = width/2;
```
As shown in this example, the pulses produced do not need to be symmetric (\( \text{wid1} \) does not need to equal \( \text{wid2} \)). The period can be offset to adjust the phase of the pulses produced (\( \text{per1} \) does not need to equal \( \text{per2} \)).

The timer slave enable bit (\( \text{TRUNx} \) bit in the \text{TIMER_STATUS} register) is updated only at the end of even-numbered periods in TOGGLE_HI mode. When \text{TIMER_DISABLE} is written to 1, the current pair of counter periods (one waveform period) completes before the timer is disabled.

As when \text{TOGGLE_HI} = 0, errors are reported if:

\[
\text{TIMERx_WIDTH} \geq \text{TIMERx_PERIOD}, \text{TIMERx_PERIOD} = 0, \text{or} \quad \text{TIMERx_PERIOD} = 1
\]

### Pulse Width Count and Capture (WDTH_CAP) Mode

In WDTH_CAP mode, the TMRx pin is an input pin (see Figure 16-16). The internally clocked timer is used to determine the period and pulse width of externally applied rectangular waveforms. Setting the TMODE field to b#10 in the \text{TIMERx_CONFIG} (timer configuration register) enables this mode.

When enabled in this mode, the timer resets the count in the \text{TIMERx_COUNTER} register to 0x0000 0001 and does not start counting until it detects a leading edge on the TMRx pin.
Using the Timer

When the timer detects the first leading edge, it starts incrementing. When it detects a trailing edge of a waveform, the timer captures the current 32-bit value of the \textit{TIMERx_COUNTER} register into the width buffer register. At the next leading edge, the timer transfers the current 32-bit value of the \textit{TIMERx_COUNTER} register into the period buffer register. The count register is reset to 0x0000 0001 again, and the timer continues counting and capturing until it is disabled.

In this mode, software can measure both the pulse width and the pulse period of a waveform. To control the definition of leading edge and trailing edge of the \textit{TMRx} pin, the \textit{PULSE_HI} bit in the \textit{TIMERx_CONFIG} register is
set or cleared. If the PULSE_HI bit is cleared, the measurement is initiated by a falling edge, the timer counter register is captured to the timer pulse width buffer register on the rising edge, and the timer period is captured on the next falling edge. When the PULSE_HI bit is set, the measurement is initiated by a rising edge, the timer counter register is captured to the timer pulse width buffer register on the falling edge, and the timer period is captured on the next rising edge.

In WDTH_CAP mode, these three events always occur at the same time as one unit:

1. The TIMERx_PERIOD register is updated from the period buffer register.
2. The TIMERx_WIDTH register is updated from the width buffer register.
3. The timer interrupt latch bit (TIMILx) gets set (if enabled) but does not generate an error.

The PERIOD_CNT bit in the TIMERx_CONFIG register controls the point in time at which this set of transactions is executed. Taken together, these three events are called a measurement report. The timer counter overflow error latch bit (TOVF_ERRx) does not get set at a measurement report. A measurement report occurs at most once per input signal period.

The current timer counter value is always copied to the width buffer and period buffer registers at the trailing and leading edges of the input signal, respectively, but these values are not visible to software. A measurement report event samples the captured values into visible registers and sets the timer interrupt to signal that TIMERx_PERIOD and TIMERx_WIDTH are ready to be read. When the PERIOD_CNT bit is set, the measurement report occurs just after the period buffer register captures its value (at a leading edge). When the PERIOD_CNT bit is cleared, the measurement report occurs just after the width buffer register captures its value (at a trailing edge).
Using the Timer

If the `PERIOD_CNT` bit is set and a leading edge occurred (see Figure 16-17), then the `TIMERx_PERIOD` and `TIMERx_WIDTH` registers report the pulse period and pulse width measured in the period that just ended. If the `PERIOD_CNT` bit is cleared and a trailing edge occurred (see Figure 16-18), then the `TIMERx_WIDTH` register reports the pulse width measured in the pulse that just ended, but the `TIMERx_PERIOD` register reports the pulse period measured at the end of the previous period.

If the `PERIOD_CNT` bit is cleared and the first trailing edge occurred, then the first period value has not yet been measured at the first measurement report, so the period value is not valid. Reading the `TIMERx_PERIOD` value in this case returns 0, as shown in Figure 16-18. To measure the pulse width of a waveform that has only one leading edge and one trailing edge, set `PERIOD_CNT = 0`. If `PERIOD_CNT = 1` for this case, no period value is captured in the period buffer register. Instead, an error report interrupt is generated (if enabled) when the counter range is exceeded and the counter wraps around. In this case, both `TIMERx_WIDTH` and `TIMERx_PERIOD` read 0 (because no measurement report occurred to copy the value captured in the width buffer register to `TIMERx_WIDTH`). See the first interrupt in Figure 16-19.

When using the `PERIOD_CNT = 0` mode described above to measure the width of a single pulse, it is recommended to disable the timer after taking the interrupt that ends the measurement interval. If desired, the timer can then be re-enabled as appropriate in preparation for another measurement. This procedure prevents the timer from free-running after the width measurement and logging errors generated by the timer count overflowing.

A timer interrupt (if enabled) is generated if the timer counter register wraps around from 0xFFFF FFFF to 0 in the absence of a leading edge. At that point, the `TOVF_ERRx` bit in the `TIMER_STATUS` register and the `ERR_TYP` bits in the `TIMERx_CONFIG` register are set, indicating a count overflow due to a period greater than the counter’s range. This is called an error report. When a timer generates an interrupt in `WDTH_CAP` mode, either an error has
occurred (an error report) or a new measurement is ready to be read (a measurement report), but never both at the same time. The `TIMERx_PERIOD` and `TIMERx_WIDTH` registers are never updated at the time an error is signaled. Refer to Figure 16-19 and Figure 16-20 for more information.

Figure 16-17. Example of Period Capture Measurement Report Timing (WDTH_CAP mode, PERIOD_CNT = 1)
Using the Timer

Figure 16-18. Example of Width Capture Measurement Report Timing (WDTH_CAP mode, PERIOD_CNT = 0)
Both \texttt{TIMILx} and \texttt{TOVF_ERRx} are sticky bits, and software has to explicitly clear them. If the timer overflowed and \texttt{PERIOD_CNT} = 1, neither the \texttt{TIMERx_PERIOD} nor the \texttt{TIMERx_WIDTH} register were updated. If the timer overflowed and \texttt{PERIOD_CNT} = 1, neither the \texttt{TIMERx_PERIOD} nor the \texttt{TIMERx_WIDTH} register were updated. If the timer overflowed and \texttt{PERIOD_CNT} = 1, neither the \texttt{TIMERx_PERIOD} nor the \texttt{TIMERx_WIDTH} register were updated. If the timer overflowed and \texttt{PERIOD_CNT} = 1, neither the \texttt{TIMERx_PERIOD} nor the \texttt{TIMERx_WIDTH} register were updated.
overflowed and \texttt{PERIOD_CNT = 0}, the \texttt{TIMERx_PERIOD} and \texttt{TIMERx_WIDTH} registers were updated only if a trailing edge was detected at a previous measurement report.

Software can count the number of error report interrupts between measurement report interrupts to measure input signal periods longer than 0xFFFF FFFF. Each error report interrupt adds a full $2^{32}$ \texttt{SCLK} counts to the total for the period, but the width is ambiguous. For example, in Figure 16-19 the period is 0x1 0000 0004 but the pulse width could be either 0x0 0000 0002 or 0x1 0000 0002.

The waveform applied to the \texttt{TMRx} pin is not required to have a 50% duty cycle, but the minimum \texttt{TMRx} low time is one \texttt{SCLK} period and the minimum \texttt{TMRx} high time is one \texttt{SCLK} period. This implies the maximum \texttt{TMRx} input frequency is \texttt{SCLK}/2 with a 50% duty cycle. Under these conditions, the \texttt{WDTH_CAP} mode timer would measure \texttt{Period = 2} and \texttt{Pulse Width = 1}.
Figure 16-20. Example Timing for Width Capture Followed by Period Overflow (WDTH_CAP mode, PERIOD_CNT = 0)
Autobaud Mode

Any one of the three timers may provide autobaud detection for the universal asynchronous receiver/transmitter (UART0). The timer input select (TIN_SEL) bit in the TIMERx_CONFIG register causes the timer to sample the UART0 port receive data (RX0) pin instead of the TMRx pin when enabled for WDTH_CAP mode.

Do not enable the UART0 until after autobaud detection is complete.

A software routine can detect the pulse widths of serial stream bit cells. Because the sample base of the timers is synchronous with UART0 operation—all derived from the phase-locked loop (PLL) clock—the pulse widths can be used to calculate the baud rate divider for UART0.

$$DIVISOR = \frac{\text{TIMERx WIDTH}}{(16 \times \text{Number of captured UART0 bits})}$$

In order to increase the number of timer counts and therefore the resolution of the captured signal, it is recommended not to measure just the pulse width of a single bit, but to enlarge the pulse of interest over more bits. Typically a NULL character (ASCII 0x00) is used in autobaud detection, as shown in Figure 16-21.

![Figure 16-21. Autobaud Detection Character 0x00](image)

Because the example frame in Figure 16-21 encloses 8 data bits and 1 start bit, apply the formula:

$$DIVISOR = \text{TIMERx WIDTH}/(16 \times 9)$$
Real UART0 RX signals often have asymmetrical falling and rising edges, and the sampling logic level is not exactly in the middle of the signal voltage range. At higher bit rates, such pulse width-based autobaud detection might not return adequate results without additional analog signal conditioning. Measuring signal periods works around this issue and is strongly recommended.

For example, predefine ASCII character “@” (40h) as an autobaud detection byte and measure the period between two subsequent falling edges. As shown in Figure 16-22, measure the period between the falling edge of the start bit and the falling edge after bit 6. Since this period encloses 8 bits, apply the formula:

$$\text{DIVISOR} = \frac{\text{TIMERx_PERIOD}}{(16 \times 8)}$$

![Figure 16-22. Autobaud Detection Character 0x40](image)

**External Event (EXT_CLK) Mode**

In `EXT_CLK` mode, the TMRx pin is an input (see Figure 16-23). The timer works as a counter clocked by an external source, which can also be asynchronous to the system clock. The current count in `TIMERx_COUNTER` represents the number of leading edge events detected. Setting the `TMODE` field to b#11 in the `TIMERx_CONFIG` register enables this mode. The `TIMERx_PERIOD` register is programmed with the value of the maximum timer external count.
Using the Timer

The waveform applied to the TMRx pin is not required to have a 50% duty cycle, but the minimum TMRx low time is one SCLK period, and the minimum TMRx high time is one SCLK period. This implies the maximum TMRx input frequency is SCLK/2.

Period may be programmed to any value from 1 to \(2^{32} - 1\), inclusive.

![Figure 16-23. Timer Flow Diagram, EXT_CLK Mode](image)

After the timer has been enabled, it resets the timer counter register to 0x0 and then waits for the first leading edge on the TMRx pin. This edge causes the timer counter register to increment to the value 0x1. Every subsequent leading edge increments the count register. After reaching the period value, the TIMILx bit is set, and an interrupt is generated. The next leading edge reloads the timer counter register again with 0x1. The timer continues counting until it is disabled. The PULSE_HI bit determines whether the leading edge is rising (PULSE_HI set) or falling (PULSE_HI cleared).
The configuration bits, TIN_SEL and PERIOD_CNT, have no effect in this mode. The TOVF_ERRx and ERR_TYP bits are set if the timer counter register wraps around from 0xFFFF FFFF to 0 or if Period = 0 at startup or when the timer counter register rolls over (from Count = Period to Count = 0x1). The TIMERx_WIDTH register is unused.

Using the Timers With the PPI

Up to two timers are used to generate frame sync signals for certain PPI modes. For detailed instructions on how to configure the timers for use with the PPI, refer to “Frame Synchronization in GP Modes” on page 11-28 of the PPI chapter.

Interrupts

Each of the three timers can generate a single interrupt. The three resulting interrupt signals are routed to the system interrupt controllers block for prioritization and masking. The timer status (TIMER_STATUS) register latches the timer interrupts to provide a means for software to determine the interrupt source. These bits are W1C and must be cleared prior to a RTI to assure that the interrupt is not reissued.

To enable interrupt generation, set the IRQ_ENA bit and unmask the interrupt source in the system interrupt mask register (SIC_IMASKx). To poll the TIMILx bit without interrupt generation, set IRQ_ENA but leave the interrupt masked. If enabled by IRQ_ENA, interrupt requests are also generated by error conditions. See Figure 16-24.

The system interrupt controllers enable flexible interrupt handling. All timers may or may not share the same interrupt channel, so that a single interrupt routine services more than one timer. In PWM mode, more timers may run with the same period settings and issue their interrupt requests simultaneously. In this case, the service routine might clear all TIMILx latch bits at once by writing 0x07 to the TIMER_STATUS register.
Using the Timer

If interrupts are enabled, make sure that the interrupt service routine (ISR) clears the TIMILx bit in the TIMERx_STATUS register before the RTI instruction executes. This ensures that the interrupt is not reissued. Remember that writes to system registers are delayed. If only a few instructions separate the TIMILx clear command from the RTI instruction, an extra SSYNC instruction may be inserted. In EXT_CLK mode, reset the TIMILx bit in the TIMERx_STATUS register at the very beginning of the interrupt service routine (ISR) to avoid missing any timer events.

Illegal States

For Table 16-1, these definitions are used:

- **Startup.** The first clock period during which the timer counter is running after the timer is enabled by writing TIMER_ENABLE.

- **Rollover.** The time when the current count matches the value in TIMERx_PERIOD and the counter is reloaded with the value 1.

- **Overflow.** The timer counter was incremented instead of doing a rollover when it was holding the maximum possible count value of 0xFFFF FFFF. The counter does not have a large enough range to express the next greater value and so erroneously loads a new value of 0x0000 0000.

- **Unchanged.** No new error.
  
  - When ERR_TYP is unchanged, it displays the previously reported error code or 00 if there has been no error since this timer was enabled.
  
  - When TOVF_ERR is unchanged, it reads 0 if there has been no error since this timer was enabled, or if software has performed a W1C to clear any previous error. If a previous error has not been acknowledged by software, TOVF_ERR reads 1.
Figure 16-24. Timers Interrupt Structure
Using the Timer

Software should read TOVF_ERR to check for an error. If TOVF_ERR is set, software can then read ERR_TYP for more information. Once detected, software should write 1 to clear TOVF_ERR to acknowledge the error.

The following table can be read as: “In mode __ at event __, if TIMERx_PERIOD is ___ and TIMERx_WIDTH is __, then ERR_TYP is ___ and TOVF_ERR is __.”

⚠️ Startup error conditions do not prevent the timer from starting. Similarly, overflow and rollover error conditions do not stop the timer. Illegal cases may cause unwanted behavior of the TMRx pin.

Table 16-1. Overview of Illegal States

<table>
<thead>
<tr>
<th>Mode</th>
<th>Event</th>
<th>TIMERx_PERIOD</th>
<th>TIMERx_WIDTH</th>
<th>ERR_TYP</th>
<th>TOVF_ERR</th>
</tr>
</thead>
<tbody>
<tr>
<td>PWM_OUT, PERIOD_CNT = 1</td>
<td>Startup</td>
<td>== 0</td>
<td>Anything</td>
<td>b#10</td>
<td>Set</td>
</tr>
<tr>
<td></td>
<td>(No boundary condition tests performed on TIMERx_WIDTH)</td>
<td>== 1</td>
<td>Anything</td>
<td>b#10</td>
<td>Set</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt;= 2</td>
<td>Anything</td>
<td>Unchanged</td>
<td>Unchanged</td>
</tr>
<tr>
<td>Rollover</td>
<td>== 0</td>
<td>Anything</td>
<td>b#10</td>
<td>Set</td>
<td></td>
</tr>
<tr>
<td></td>
<td>== 1</td>
<td>Anything</td>
<td>b#11</td>
<td>Set</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&gt;= 2</td>
<td>&gt;= 0</td>
<td>b#11</td>
<td>Set</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&gt;= 2</td>
<td>&lt; TIMERx_PERIOD</td>
<td>Unchanged</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&gt;= 2</td>
<td>&gt;= TIMERx_PERIOD</td>
<td>b#11</td>
<td>Set</td>
<td></td>
</tr>
<tr>
<td>Overflow, not possible unless there is also another error, such as TIMERx_PERIOD == 0.</td>
<td>Anything</td>
<td>Anything</td>
<td>b#01</td>
<td>Set</td>
<td></td>
</tr>
</tbody>
</table>
### Table 16-1. Overview of Illegal States (Cont'd)

<table>
<thead>
<tr>
<th>Mode</th>
<th>Event</th>
<th>TIMERx_PERIOD</th>
<th>TIMERx_WIDTH</th>
<th>ERR_TYP</th>
<th>TOVF_ERR</th>
</tr>
</thead>
<tbody>
<tr>
<td>PWM_OUT, PERIOD_CNT = 0</td>
<td>Startup</td>
<td>Anything</td>
<td>== 0</td>
<td>b#01</td>
<td>Set</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Anything &gt; = 1</td>
<td>Unchanged</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Rollover</td>
<td>Rollover is not possible in this mode.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WDTH_CAP</td>
<td>Startup</td>
<td>TIMERx_PERIOD and TIMERx_WIDTH are read-only in this mode, no error possible.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Rollover</td>
<td>TIMERx_PERIOD and TIMERx_WIDTH are read-only in this mode, no error possible.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Overflow</td>
<td>Anything</td>
<td>Anything</td>
<td>b#01</td>
<td>Set</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXT_CLK</td>
<td>Startup</td>
<td>== 0</td>
<td>Anything</td>
<td>b#10</td>
<td>Set</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt;= 1</td>
<td>Anything</td>
<td>Unchanged</td>
<td>Unchanged</td>
</tr>
<tr>
<td></td>
<td>Rollover</td>
<td>== 0</td>
<td>Anything</td>
<td>b#10</td>
<td>Set</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt;= 1</td>
<td>Anything</td>
<td>Unchanged</td>
<td>Unchanged</td>
</tr>
<tr>
<td></td>
<td>Overflow, not possible unless there is also another error, such as TIMERx_WIDTH == 0.</td>
<td>Anything</td>
<td>Anything</td>
<td>b#01</td>
<td>Set</td>
</tr>
</tbody>
</table>
Summary

Table 16-2 summarizes control bit and register usage in each timer mode.

Table 16-2. Control Bit and Register Usage Chart

<table>
<thead>
<tr>
<th>Bit/Register</th>
<th>PWM_OUT Mode</th>
<th>WDTH_CAP Mode</th>
<th>EXT_CLK Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMER_ENABLE</td>
<td>1 - Enable timer</td>
<td>1 - Enable timer</td>
<td>1 - Enable timer</td>
</tr>
<tr>
<td></td>
<td>0 - No effect</td>
<td>0 - No effect</td>
<td>0 - No effect</td>
</tr>
<tr>
<td>TIMER_DISABLE</td>
<td>1 - Disable timer at end of period</td>
<td>1 - Disable timer</td>
<td>1 - Disable timer</td>
</tr>
<tr>
<td></td>
<td>0 - No effect</td>
<td>0 - No effect</td>
<td>0 - No effect</td>
</tr>
<tr>
<td>TMODE</td>
<td>b#01</td>
<td>b#10</td>
<td>b#11</td>
</tr>
<tr>
<td>PULSE_HI</td>
<td>1 - Generate high width</td>
<td>1 - Measure high width</td>
<td>1 - Count rising edges</td>
</tr>
<tr>
<td></td>
<td>0 - Generate low width</td>
<td>0 - Measure low width</td>
<td>0 - Count falling edges</td>
</tr>
<tr>
<td>PERIOD_CNT</td>
<td>1 - Generate PWM</td>
<td>1 - Interrupt after measuring period</td>
<td>Unused</td>
</tr>
<tr>
<td></td>
<td>0 - Single width pulse</td>
<td>0 - Interrupt after measuring width</td>
<td></td>
</tr>
<tr>
<td>IRQ_ENA</td>
<td>1 - Enable interrupt</td>
<td>1 - Enable interrupt</td>
<td>1 - Enable interrupt</td>
</tr>
<tr>
<td></td>
<td>0 - Disable interrupt</td>
<td>0 - Disable interrupt</td>
<td>0 - Disable interrupt</td>
</tr>
<tr>
<td>TIN_SEL</td>
<td>Depends on CLK_SEL:</td>
<td>1 - Select RX input</td>
<td>Unused</td>
</tr>
<tr>
<td></td>
<td>If CLK_SEL = 1,</td>
<td>0 - Select TMRx input</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 - Count PPI_CLKs</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 - Count PF1 clocks</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>If CLK_SEL = 0,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Unused</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUT_DIS</td>
<td>1 - Disable TMRx pin</td>
<td>Unused</td>
<td>Unused</td>
</tr>
<tr>
<td></td>
<td>0 - Enable TMRx pin</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLK_SEL</td>
<td>1 - PWM_CLK clocks timer</td>
<td>Unused</td>
<td>Unused</td>
</tr>
<tr>
<td></td>
<td>0 - SCLK clocks timer</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 16-2. Control Bit and Register Usage Chart (Cont’d)

<table>
<thead>
<tr>
<th>Bit/Register</th>
<th>PWM_OUT Mode</th>
<th>WDTH_CAP Mode</th>
<th>EXT_CLK Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOGGLE_HI</td>
<td>1 - One waveform period every two counter periods 0 - One waveform period every one counter period</td>
<td>Unused</td>
<td>Unused</td>
</tr>
<tr>
<td>ERR_TYP</td>
<td>Reports b#00, b#01, b#10, or b#11, as appropriate</td>
<td>Reports b#00 or b#01, as appropriate</td>
<td>Reports b#00, b#01, or b#10, as appropriate</td>
</tr>
<tr>
<td>EMU_RUN</td>
<td>0 - Halt during emulation 1 - Count during emulation</td>
<td>0 - Halt during emulation 1 - Count during emulation</td>
<td>0 - Halt during emulation 1 - Count during emulation</td>
</tr>
<tr>
<td>TMR Pin</td>
<td>Depends on OUT_DIS: 1 - Three-state 0 - Output</td>
<td>Depends on TIN_SEL: 1 - Unused 0 - Input</td>
<td>Input</td>
</tr>
<tr>
<td>Period</td>
<td>R/W: Period value</td>
<td>RO: Period value</td>
<td>R/W: Period value</td>
</tr>
<tr>
<td>Width</td>
<td>R/W: Width value</td>
<td>RO: Width value</td>
<td>Unused</td>
</tr>
<tr>
<td>Counter</td>
<td>RO: Counts up on SCLK or PWM_CLK</td>
<td>RO: Counts up on SCLK</td>
<td>RO: Counts up on event</td>
</tr>
<tr>
<td>TRUNx</td>
<td>Read: Timer slave enable status Write: 1 - Stop timer if disabled 0 - No effect</td>
<td>Read: Timer slave enable status Write: 1 - No effect 0 - No effect</td>
<td>Read: Timer slave enable status Write: 1 - No effect 0 - No effect</td>
</tr>
</tbody>
</table>
The core timer is a programmable interval timer which can generate periodic interrupts. The core timer runs at the core clock (CCLK) rate. The timer includes four core memory mapped registers (MMRs), the timer control register (TCNTL), the timer count register (TCOUNT), the timer period register (TPERIOD), and the timer scale register (TSCALE).

<table>
<thead>
<tr>
<th>Bit/Register</th>
<th>PWM_OUT Mode</th>
<th>WDTH_CAP Mode</th>
<th>EXT_CLK Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOVF_ERR</td>
<td>Set at startup or rollover if period = 0 or 1 Set at rollover if width &gt;= Period Set if counter wraps</td>
<td>Set if counter wraps</td>
<td>Set if counter wraps or set at startup or rollover if period = 0</td>
</tr>
<tr>
<td>IRQ</td>
<td>Depends on IRQ_ENA: 1 - Set when TOVF_ERR set or when counter equals period and PERIOD_CNT = 1 or when counter equals width and PERIOD_CNT = 0 0 - Not set</td>
<td>Depends on IRQ_ENA: 1 - Set when TOVF_ERR set or when counter captures period and PERIOD_CNT = 1 or when counter captures width and PERIOD_CNT = 0 0 - Not set</td>
<td>Depends on IRQ_ENA: 1 - Set when counter equals period or TOVF_ERR set 0 - Not set</td>
</tr>
</tbody>
</table>
Figure 16-25 provides a block diagram of the core timer.

![Core Timer Block Diagram](image)

**TCNTL Register**

When the timer is enabled by setting the TMREN bit in the core timer control register (TCNTL), the TCOUNT register is decremented once every TSCALE + 1 number of clock cycles. When the value of the TCOUNT register reaches 0, an interrupt is generated and the TINT bit is set in the TCNTL register. If the TAUTORLD bit in the TCNTL register is set, then the TCOUNT register is reloaded with the contents of the TPERIOD register and the count begins again. (See Figure 16-26.)

The TINT bit in the TCNTL register indicates that an interrupt has been generated. Note that this is not a W1C bit. Write a 0 to clear it. However, the write is optional. It is not required to clear interrupt requests. The core timer module does not provide any further interrupt enable bit. When the timer is enabled, interrupts can be masked in the CEC controller.
The core timer can be put into low power mode by clearing the \texttt{TMPWR} bit in the \texttt{TCNTL} register. Before using the timer, set the \texttt{TMPWR} bit. This restores clocks to the timer unit. When \texttt{TMPWR} is set, the core timer may then be enabled by setting the \texttt{TMREN} bit in the \texttt{TCNTL} register.

\textbf{Hardware behavior is undefined if \texttt{TMREN} is set when \texttt{TMPWR} = 0.}

\textbf{Core Timer Control Register (TCNTL)}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{core_timer_control_register}
\caption{Core Timer Control Register}
\end{figure}

\begin{enumerate}
\item \texttt{TINT} \\
\begin{itemize}
\item Sticky status bit
\item 0 - Timer has not generated an interrupt
\item 1 - Timer has generated an interrupt
\end{itemize}
\item \texttt{TAUTORLD} \\
\begin{itemize}
\item 0 - Disable auto-reload feature. When \texttt{TCOUNT} reaches zero, the timer generates an interrupt and halts
\item 1 - Enable auto-reload feature. When \texttt{TCOUNT} reaches zero and the timer generates an interrupt, \texttt{TCOUNT} is automatically reloaded with the contents of \texttt{TPERIOD} and the timer continues to count
\end{itemize}
\item \texttt{TMPWR} \\
\begin{itemize}
\item 0 - Puts the timer in low power mode
\item 1 - Active state. Timer can be enabled using the \texttt{TMREN} bit
\end{itemize}
\item \texttt{TMREN} \\
\begin{itemize}
\item Meaningful only when \texttt{TMPWR} = 1
\item 0 - Disable timer
\item 1 - Enable timer
\end{itemize}
\end{enumerate}

\textbf{TCOUNT Register}

The core timer count register (\texttt{TCOUNT}) decrements once every \texttt{TSCALE} + 1 clock cycles. When the value of \texttt{TCOUNT} reaches 0, an interrupt is generated and the \texttt{TINT} bit of the \texttt{TCNTL} register is set. (See Figure 16-27.)
When auto-reload is enabled, the TCOUNT register (Figure 16-28) is reloaded with the value of the core timer period register (TPERIOD) whenever TCOUNT reaches 0.

To ensure that there is valid data in the TPERIOD register, the TPERIOD and TCOUNT registers are initialized simultaneously on the first write to either register. If a different value is desired for the first count period, write the data to TCOUNT after writing to TPERIOD.
TSSCALE Register

The core timer scale register (TSSCALE), shown in Figure 16-29, stores the scaling value that is one less than the number of cycles between decrements of TCOUNT. For example, if the value in the TSSCALE register is 0, the counter register decrements once every clock cycle. If TSSCALE is 1, the counter decrements once every two cycles.

Watchdog Timer

The processor includes a 32-bit timer that can be used to implement a software watchdog function. A software watchdog can improve system reliability by generating an event to the processor core if the timer expires before being updated by software. Depending on how the watchdog timer is programmed, the event that is generated may be a reset, a non-maskable interrupt, or a general-purpose interrupt. The watchdog timer is clocked by the system clock (SCLK).
Watchdog Timer Operation

To use the watchdog timer:

1. Set the count value for the watchdog timer by writing the count value into the watchdog count register (WDOG_CNT). Note that loading the WDOG_CNT register while the watchdog timer is not enabled will also pre-load the WDOG_STAT register.

2. In the watchdog control register (WDOG_CTL), select the event to be generated upon time-out.

3. Enable the watchdog timer in WDOG_CTL. The watchdog timer then begins counting down, decrementing the value in the WDOG_STAT register. When the WDOG_STAT reaches 0, the programmed event is generated. To prevent the event from being generated, software must reload the count value from WDOG_CNT to WDOG_STAT by executing a write (of any value) to WDOG_STAT, or must disable the watchdog timer in WDOG_CTL before the watchdog timer expires.

WDOG_CNT Register

The watchdog count register (WDOG_CNT) holds the 32-bit unsigned count value. The WDOG_CNT register must be accessed with 32-bit read/writes only. See Figure 16-30.

The WDOG_CNT register holds the programmable count value. A valid write to the WDOG_CNT register also pre-loads the watchdog counter. For added safety, the WDOG_CNT register can be updated only when the watchdog timer is disabled. A write to the watchdog count register while the timer is enabled does not modify the contents of this register.
Watchdog Timer

Watchdog Count Register (WDOG_CNT)

WDOG_STAT Register

The 32-bit watchdog status register (WDOG_STAT) contains the current count value of the watchdog timer. See Figure 16-31. Reads to WDOG_STAT return the current count value. When the watchdog timer is enabled, WDOG_STAT is decremented by 1 on each SCLK cycle. When WDOG_STAT reaches 0, the watchdog timer stops counting and the event selected in the watchdog control register (WDOG_CTL) is generated.

Watchdog Status Register (WDOG_STAT)
Values cannot be stored directly in `WDOG_STAT`, but are instead copied from `WDOG_CNT`. This can happen in two ways.

- While the watchdog timer is disabled, writing the `WDOG_CNT` register pre-loads the `WDOG_STAT` register.
- While the watchdog timer is enabled, writing the `WDOG_STAT` register loads it with the value in `WDOG_CNT`.

When the processor executes a write (of an arbitrary value) to `WDOG_STAT`, the value in `WDOG_CNT` is copied into `WDOG_STAT`. Typically, software sets the value of `WDOG_CNT` at initialization, then periodically writes to `WDOG_STAT` before the watchdog timer expires. This reloads the watchdog timer with the value from `WDOG_CNT` and prevents generation of the selected event.

The `WDOG_STAT` register is a 32-bit unsigned system memory-mapped register that must be accessed with 32-bit reads and writes.

If the user does not reload the counter before `SCLK * Count` register cycles, a watchdog interrupt or reset is generated and the `TRO` bit in the watchdog control register is set. When this happens the counter stops decrementing and remains at zero.

If the counter is enabled with a zero loaded to the counter, the `TRO` bit of the watchdog control register is set immediately and the counter remains at zero and does not decrement.
Watchdog Timer

WDMOG_CTL Register

The watchdog control register (WDOG_CTL) is a 16-bit system memory-mapped register used to control the watchdog timer (see Figure 16-32).

Watchdog Control Register (WDOG_CTL)

![Watchdog Control Register](image)

Figure 16-32. Watchdog Control Register

The **ICTL[1:0]** field is used to select the event that is generated when the watchdog timer expires. Note that if the general-purpose interrupt option is selected, the system interrupt mask register (SIC_IMASKx) should be appropriately configured to unmask that interrupt. If the generation of watchdog events is disabled, the watchdog timer operates as described, except that no event is generated when the watchdog timer expires.

The **TMR_EN[7:0]** field is used to enable and disable the watchdog timer. Writing any value other than the disable value into this field enables the watchdog timer. This multibit disable key minimizes the chance of inadvertently disabling the watchdog timer.
Software can determine whether the timer has rolled over by interrogating the $TRO$ status bit of the watchdog control register. This is a sticky bit that is set whenever the watchdog timer count reaches 0 and cleared only by disabling the watchdog timer and then writing a 1 to the bit.

Note that when the processor is in emulation mode, the watchdog timer counter will not decrement even if it is enabled.
Watchdog Timer
17 REAL-TIME CLOCK

The real-time clock (RTC) provides a set of digital watch features to the processor, including time of day, alarm, and stopwatch countdown. It is typically used to implement either a real-time watch or a life counter.

The RTC watch features are clocked by a 32.768 kHz crystal external to the processor. The RTC uses dedicated power supply pins and is independent of any reset, which enables it to maintain functionality even when the rest of the processor is powered down.

The RTC input clock is divided down to a 1 Hz signal by a prescaler, which can be bypassed. When bypassed, the RTC is clocked at the 32.768 kHz crystal rate. In normal operation, the prescaler is enabled.

The primary function of the RTC is to maintain an accurate day count and time of day. The RTC accomplishes this by means of four counters:

- 60-second counter
- 60-minute counter
- 24-hour counter
- 32768-day counter

The RTC increments the 60-second counter once per second and increments the other three counters when appropriate. The 32768-day counter is incremented each day at midnight (0 hours, 0 minutes, 0 seconds). Interrupts can be issued periodically, either every second, every minute, every hour, or every day. Each of these interrupts can be independently controlled.
The RTC provides two alarm features, programmed with the RTC alarm register (RTC_ALARM). The first is a time of day alarm (hour, minute, and second). When the alarm interrupt is enabled, the RTC generates an interrupt each day at the time specified. The second alarm feature allows the application to specify a day as well as a time. When the day alarm interrupt is enabled, the RTC generates an interrupt on the day and time specified. The alarm interrupt and day alarm interrupt can be enabled or disabled independently.

The RTC provides a stopwatch function that acts as a countdown timer. The application can program a second count into the RTC stopwatch count register (RTC_SWCNT). When the stopwatch interrupt is enabled and the specified number of seconds have elapsed, the RTC generates an interrupt.

**Interfaces**

The RTC external interface consists of two clock pins, which together with the external components form the reference clock circuit for the RTC. The RTC interfaces internally to the processor system through the peripheral access bus (PAB), and through the interrupt interface to the system interrupt controllers (SICx).

The RTC has dedicated power supply pins that power the clock functions at all times, including when the core power supply is turned off.

**RTC Clock Requirements**

The RTC timer, shown in Figure 17-1, is clocked by a 32.768 kHz crystal external to the processor. The RTC system memory-mapped registers (MMRs) are clocked by this crystal. When the prescaler is disabled, the RTC MMRs are clocked at the 32.768 kHz crystal frequency. When the prescaler is enabled, the RTC MMRs are clocked at the 1 Hz rate.
There is no way to disable the RTC counters from software. If a given system does not require the RTC functionality, then it may be disabled with hardware tie-offs. Tie the \texttt{RTXI} pin to \texttt{EGND}, tie the \texttt{RTCVDD} pin to \texttt{EVDD}, and leave the \texttt{RTXO} pin unconnected.

Figure 17-1. RTC Block Diagram
RTC Programming Model

The RTC programming model consists of a set of system MMRs. Software can configure the RTC and can determine the status of the RTC through reads and writes to these registers. The RTC interrupt control register (RTC_ICTL) and the RTC interrupt status register (RTC_ISTAT) provide RTC interrupt management capability.

Note that software cannot disable the RTC counting function. However, all RTC interrupts can be disabled, or masked. At reset, all interrupts are disabled. The RTC state can be read via the system MMR status registers at any time.

The primary real-time clock functionality, shown in Figure 17-1, consists of registers and counters that are powered by an independent RTC V<sub>dd</sub> supply. This logic is never reset; it comes up in an unknown state when RTC V<sub>dd</sub> is first powered on.

The RTC also contains logic powered by the same internal V<sub>dd</sub> as the processor core and other peripherals. This logic contains some control functionality, holding registers for PAB write data, and prefetched PAB read data shadow registers for each of the five RTC V<sub>dd</sub>-powered registers. This logic is reset by the same system reset and clocked by the same SCLK as the other peripherals.

Figure 17-2 shows the connections between the RTC V<sub>dd</sub>-powered RTC MMRs and their corresponding internal V<sub>dd</sub>-powered write holding registers and read shadow registers. In the figure, “REG” means each of the RTC_STAT, RTC_ALARM, RTC_SWCNT, RTC_ICTL, and RTC_PREN registers. The RTC_ISTAT register connects only to the PAB.

The rising edge of the 1 Hz RTC clock is the “1 Hz tick”. Software can synchronize to the 1 Hz tick by waiting for the seconds event flag to set or by waiting for the seconds interrupt (if enabled).
Register Writes

Writes to all RTC MMRs, except the RTC interrupt status register (`RTC_ISTAT`), are saved in write holding registers and then are synchronized to the RTC 1 Hz clock. The write pending status bit in `RTC_ISTAT` indicates the progress of the write. The write pending status bit is set when a write is initiated and is cleared when all writes are complete. The falling edge of the write pending status bit causes the write complete flag in `RTC_ISTAT` to be set. This flag can be configured in `RTC_ICTL` to cause an interrupt. Software does not have to wait for writes to one RTC MMR to

Figure 17-2. RTC Register Architecture
RTC Programming Model

complete before writing to another RTC MMR. The write pending status bit is set if any writes are in progress, and the write complete flag is set only when all writes are complete.

- Any writes in progress when peripherals are reset will be aborted. Do not stop SCLK (enter deep sleep mode) or remove internal \( V_{dd} \) power until all RTC writes have completed.

- Do not attempt another write to the same register without waiting for the previous write to complete. Subsequent writes to the same register are ignored if the previous write is not complete.

- Reading a register that has been written before the write complete flag is set will return the old value. Always check the write pending status bit before attempting a read or write.

Write Latency

Writes to the RTC MMRs are synchronized to the 1 Hz RTC clock. When setting the time of day, do not factor in the delay when writing to the RTC MMRs. The most accurate method of setting the real-time clock is to monitor the seconds (1 Hz) event flag or to program an interrupt for this event and then write the current time to the RTC status register (RTC_STAT) in the interrupt service routine (ISR). The new value is inserted ahead of the incremenet. Hardware adds one second to the written value (with appropriate carries into minutes, hours and days) and loads the incremented value at the next 1 Hz tick, when it represents the then-current time.

Writes posted at any time are properly synchronized to the 1 Hz clock. Writes complete at the rising edge of the 1 Hz clock. A write posted just before the 1 Hz tick may not be completed until the 1 Hz tick one second later. Any write posted in the first 990 ms after a 1 Hz tick will complete on the next 1 Hz tick, but the simplest, most predictable and
Real-Time Clock

The recommended technique is to only post writes to RTC_STAT, RTC_ALARM, RTC_SWCNT, RTC_ICTL, or RTC_PREN immediately after a seconds interrupt or event. All five registers may be written in the same second.

W1C bits in the RTC_ISTAT register take effect immediately.

Register Reads

There is no latency when reading RTC MMRs, as the values come from the read shadow registers. The shadows are updated and ready for reading by the time any RTC interrupts or event flags for that second are asserted. Once the internal Vdd logic completes its initialization sequence after SCLK starts, there is no point in time when it is unsafe to read the RTC MMRs for synchronization reasons. They always return coherent values, although the values may be unknown.

Deep Sleep

When the dynamic power management controller (DPMC) state is deep sleep, all clocks in the system (except RTXI and the RTC 1 Hz tick) are stopped. In this state, the RTC Vdd counters continue to increment. The internal Vdd shadow registers are not updated, but neither can they be read.

During deep sleep state, all bits in RTC_ISTAT are cleared. Events that occur during deep sleep are not recorded in RTC_ISTAT. The internal Vdd RTC control logic generates a virtual 1 Hz tick within one RTXI period (30.52 μs) after SCLK restarts. This loads all shadow registers with up-to-date values and sets the seconds event flag. Other event flags may also be set. When the system wakes up from deep sleep, whether by an RTC event or a hardware reset, all of the RTC events that occurred during that second (and only that second) are reported in RTC_ISTAT.
When the system wakes up from deep sleep state, software does not need to W1C the bits in RTC_ISTAT. All “write-1-to-clear” bits are already cleared by hardware. The seconds event flag is set when the RTC internal Vdd logic has completed its restart sequence. Software should wait until the seconds event flag is set and then may begin reading or writing any RTC register.

**Prescaler Enable**

The single active bit of the RTC prescaler enable register (RTC_PREN) is written using a synchronization path. Clearing of the bit is synchronized to the 32.768 kHz clock. This faster synchronization allows the module to be put into high-speed mode (bypassing the prescaler) without waiting the full 1 second for the write to complete that would be necessary if the module were already running with the prescaler enabled.

When setting the RTC_PREN bit, the first positive edge of the 1 Hz clock occurs 1 to 2 cycles of the 32.768 kHz clock after the prescaler is enabled. The write complete status/interrupt works as usual when enabling or disabling the prescale counter. The new RTC clock rate is in effect before the write complete status is set.

**Event Flags**

The unknown values in the registers at powerup can cause event flags to set before the correct value is written into each of the registers. By catching the 1 Hz clock edge, the write to RTC_STAT can occur a full second before the write to RTC_ALARM. This would cause an extra second of delay between the validity of RTC_STAT and RTC_ALARM, if the value of the RTC_ALARM out of reset is the same as the value written to RTC_STAT. Wait for the writes to complete on these registers before using the flags and interrupts associated with their values.
Real-Time Clock

The following is a list of flags along with the conditions under which they are valid:

- **Seconds (1 Hz) event flag**
  
  Always set on the positive edge of the 1 Hz clock and after shadow registers have updated after waking from deep sleep. This is valid as long as the RTC 1 Hz clock is running. Use this flag or interrupt to validate the other flags.

- **Write complete**
  
  Always valid.

- **Write pending status**
  
  Always valid.

- **Minutes event flag**
  
  Valid only after the second field in `RTC_STAT` is valid. Use the write complete and write pending status flags or interrupts to validate the `RTC_STAT` value before using this flag value or enabling the interrupt.

- **Hours event flag**
  
  Valid only after the minute field in `RTC_STAT` is valid. Use the write complete and write pending status flags or interrupts to validate the `RTC_STAT` value before using this flag value or enabling the interrupt.
RTC Programming Model

• 24 Hours event flag

Valid only after the hour field in RTC_STAT is valid. Use the write complete and write pending status flags or interrupts to validate the RTC_STAT value before using this flag value or enabling the interrupt.

• Stopwatch event flag

Valid only after the RTC_SWCNT register is valid. Use the write complete and write pending status flags or interrupts to validate the RTC_SWCNT value before using this flag value or enabling the interrupt.

• Alarm event flag

Valid only after the RTC_STAT and RTC_ALARM registers are valid. Use the write complete and write pending status flags or interrupts to validate the RTC_STAT and RTC_ALARM values before using this flag value or enabling its interrupt.

• Day alarm event flag

Same as Alarm.

Writes posted together at the beginning of the same second take effect together at the next 1 Hz tick. The following sequence is safe and does not result in any spurious interrupts from a previous state.

1. Wait for 1 Hz tick.

2. Write 1s to clear the RTC_ISTAT flags for alarm, day alarm, stopwatch, and/or per-interval.

3. Write new values for RTC_STAT, RTC_ALARM, and/or RTC_SWCNT.
4. Write new value for RTC_ICTL with alarm, day alarm, stopwatch, and/or per-interval interrupts enabled.

5. Wait for 1 Hz tick.

6. New values have now taken effect simultaneously.

**Interrupts**

The RTC can provide interrupts at several programmable intervals, including:

- Per second
- Per minute
- Per hour
- Per day
- On countdown from a programmable value
- Daily at a specific time
- On a specific day and time

The RTC can be programmed to provide an interrupt at the completion of all pending writes to any of the 1 Hz registers (RTC_STAT, RTC_ALARM, RTC_SWCNT, RTC_ICTL, and RTC_PREN). Interrupts can be individually enabled or disabled using the RTC interrupt control register (RTC_ICTL). Interrupt status can be determined by reading the RTC interrupt status register (RTC_ISTAT).

The RTC interrupt is set whenever an event latched into the RTC_ISTAT register is enabled in the RTC_ICTL register. The pending RTC interrupt is cleared whenever all enabled and set bits in RTC_ISTAT are cleared, or when all bits in RTC_ICTL corresponding to pending events are cleared.
As shown in Figure 17-3, the RTC generates an interrupt request (IRQ) to the processor core for event handling and wake up from a sleep state. The RTC generates a separate signal for wake up from a deep sleep or from an internal $V_{dd}$ power-off state. The deep sleep wake-up signal is asserted at the 1 Hz tick when any RTC interval event enabled in $\text{RTC}_{-}\text{ICTL}$ occurs. The assertion of the deep sleep wake-up signal causes the processor core clock ($\text{CCLK}$) and the system clock ($\text{SCLK}$) to restart. Any enabled event that asserts the RTC deep sleep wake-up signal also causes the RTC IRQ to assert once $\text{SCLK}$ restarts.

Figure 17-3. RTC Interrupt Structure
RTC Status (RTC_STAT) Register

The RTC status register (RTC_STAT), shown in Figure 17-4, is used to read or write the current time. Reads return a 32-bit value that always reflects the current state of the days, hours, minutes, and seconds counters. Reads and writes must be 32-bit transactions; attempted 16-bit transactions result in an MMR error. Reads always return a coherent 32-bit value. The hours, minutes, and seconds fields are usually set to match the real time of day. The day counter value is incremented every day at midnight to record how many days have elapsed since it was last modified. Its value does not correspond to a particular calendar day. The 15-bit day counter provides a range of 89 years, 260 or 261 days (depending on leap years) before it overflows.

After the 1 Hz tick, program RTC_STAT with the current time. At the next 1 Hz tick, RTC_STAT takes on the new, incremented value. For example:

1. Wait for 1 Hz tick.
2. Read RTC_STAT, get 10:45:30.
4. Read RTC_STAT, still get old time 10:45:30.
5. Wait for 1 Hz tick.
6. Read RTC_STAT, get new current time, 13:11:00.
RTC Interrupt Control (RTC_ICTL) Register

The eight RTC interrupt events (see Figure 17-5) can be individually masked or enabled by the RTC interrupt control register (RTC_ICTL). The seconds interrupt is generated on each 1 Hz clock tick, if enabled. The minutes interrupt is generated at the 1 Hz clock tick that advances the seconds counter from 59 to 0. The hour interrupt is generated at the 1 Hz clock tick that advances the minute counter from 59 to 0. The 24-hour interrupt occurs once per 24-hour period at the 1 Hz clock tick that advances the time to midnight (00:00:00). Any of these interrupts can generate a wake-up request to the processor, if enabled. All implemented bits are read/write.

This register is only partially cleared at reset, so some events may appear to be enabled initially. However, the RTC interrupt and the RTC wake up to the PLL are handled specially and are masked.

Figure 17-4. RTC Status Register

RTC Interrupt Control (RTC_ICTL) Register

The eight RTC interrupt events (see Figure 17-5) can be individually masked or enabled by the RTC interrupt control register (RTC_ICTL). The seconds interrupt is generated on each 1 Hz clock tick, if enabled. The minutes interrupt is generated at the 1 Hz clock tick that advances the seconds counter from 59 to 0. The hour interrupt is generated at the 1 Hz clock tick that advances the minute counter from 59 to 0. The 24-hour interrupt occurs once per 24-hour period at the 1 Hz clock tick that advances the time to midnight (00:00:00). Any of these interrupts can generate a wake-up request to the processor, if enabled. All implemented bits are read/write.

This register is only partially cleared at reset, so some events may appear to be enabled initially. However, the RTC interrupt and the RTC wake up to the PLL are handled specially and are masked.
Real-Time Clock

(forced low) until after the first write to the `RTC_ICTL` register is complete. Therefore, all interrupts act as if they were disabled at system reset (as if all bits of `RTC_ICTL` were zero), even though some bits of `RTC_ICTL` may read as nonzero. If no RTC interrupts are needed immediately after reset, it is recommended to write `RTC_ICTL` to 0x0000 so that later read-modify-write accesses will function as intended.

**RTC Interrupt Status (RTC_ISTAT) Register**

The RTC interrupt status register (RTC_ISTAT) provides the status of all RTC interrupts (see Figure 17-6). These bits are sticky. Once set by the corresponding event, each bit remains set until cleared by a software write to this register. Event flags are always set; they are not masked by the interrupt enable bits in `RTC_ICTL`. Values are cleared by writing a 1 to the respective bit location, except for the write pending status bit, which is read-only. Writes of 0 to any bit of the register have no effect. This register is cleared at reset and during deep sleep.

Figure 17-5. RTC Interrupt Control Register

---

ADSP-BF538/ADSP-BF538F Blackfin Processor Hardware Reference 17-15
RTC Stopwatch Count (RTC_SWCNT) Register

The RTC stopwatch count register (RTC_SWCNT) contains the countdown value for the stopwatch (see Figure 17-7). The stopwatch counts down seconds from the programmed value and generates an interrupt (if enabled) when the count reaches 0. The counter stops counting at this point and does not resume counting until a new value is written to RTC_SWCNT. Once running, the counter may be overwritten with a new value. This allows the stopwatch to be used as a watchdog timer with a precision of one second. Writing the running stopwatch to 0 forces it to stop and interrupt early. The stopwatch event flag is set at the 1 Hz tick at which any of these occur:

Figure 17-6. RTC Interrupt Status Register
• The stopwatch counter decrements to 0x0000

• A write of 0x0000 to RTC_SWCNT completes and the stopwatch was running (current stopwatch count was greater than 0)

• A write of 0x0000 to RTC_SWCNT completes and the stopwatch was stopped (current stopwatch count was equal to 0)

The register can be programmed to any value between 0 and \((2^{16} - 1)\) seconds. This is a range of 18 hours, 12 minutes, and 15 seconds.

Typically, software should wait for a 1 Hz tick, then write RTC_SWCNT. One second later, RTC_SWCNT changes to the new value and begins decrementing. Because the register write occupies nearly one second, the time from writing a value of \(N\) until the stopwatch interrupt is nearly \(N + 1\) seconds. To produce an exact delay, software can compensate by writing \(N - 1\) to get a delay of nearly \(N\) seconds. This implies that you cannot achieve a delay of 1 second with the stopwatch. Writing a value of 1 immediately after a 1 Hz tick results in a stopwatch interrupt nearly two seconds later. To wait one second, software should just wait for the next 1 Hz tick.

The RTC stopwatch count register is not reset. After initial powerup, it may be running. When the stopwatch is not used, writing it to 0 to force it to stop saves a small amount of power.

**RTC Stopwatch Count Register (RTC_SWCNT)**

![RTC Stopwatch Count Register](image)

Figure 17-7. RTC Stopwatch Count Register
RTC Alarm \textbf{(RTC\_ALARM)} Register

The RTC alarm register (\texttt{RTC\_ALARM}) is programmed by software for the time (in hours, minutes, and seconds) the alarm interrupt occurs (see Figure 17-8). Reads and writes can occur at any time. The alarm interrupt occurs whenever the hour, minute, and second fields first match those of the RTC status register. The day interrupt occurs whenever the day, hour, minute, and second fields first match those of the RTC status register.

![RTC Alarm Register (RTC\_ALARM)](image)

Figure 17-8. RTC Alarm Register

\section*{RTC Prescaler Enable (RTC\_PREN) Register}

The RTC prescaler enable register (\texttt{RTC\_PREN}) has one active bit (see Figure 17-9). When this bit is set, the prescaler is enabled, and the RTC runs at a frequency of 1 Hz. When this bit is cleared, the prescaler is disabled, and the RTC runs at the 32.768 kHz crystal frequency.
In order for the RTC to operate at the proper rate, software must set the prescaler enable bit after initial powerup. Write \texttt{RTC\_PREN} and then wait for the write complete event before programming the other registers. It is safe to write \texttt{RTC\_PREN} to 1 every time the processor boots. The first time sets the bit, and subsequent writes will have no effect, as no state is changed.

Do not disable the prescaler by clearing the bit in \texttt{RTC\_PREN} without making sure that there are no writes to RTC MMRs in progress. Do not switch between fast and slow mode during normal operation by setting and clearing this bit, as this disrupts the accurate tracking of real time by the counters. To avoid these potential errors, initialize the RTC during startup via \texttt{RTC\_PREN} and do not dynamically alter the state of the prescaler during normal operation.

Running without the prescaler enabled is provided primarily as a test mode. All functionality works, just 32,768 times as fast. Typical software should never program \texttt{RTC\_PREN} to 0. The only reason to do so is to synchronize the 1 Hz tick to a more precise external event, as the 1 Hz tick predictably occurs a few \texttt{RTXI} cycles after a 0 → 1 transition of \texttt{RTC\_PREN}. Use the following sequence to achieve synchronization to within 100 μs.

1. Write \texttt{RTC\_PREN} to 0.
2. Wait for the write to complete.
3. Wait for the external event.

Figure 17-9. Prescaler Enable Register
4. Write RTC_PREN to 1.

5. Wait for the write to complete.

6. Reprogram the time into RTC_STAT.

State Transitions Summary

Table 17-1 shows how each RTC MMR is affected by the system states. The phase-locked loop (PLL) states (reset, full on, active, sleep, and deep sleep) are defined in Chapter 8, “Dynamic Power Management”. “No Power” means none of the processor power supply pins are connected to a source of energy. “Off” means the processor core, peripherals, and memory are not powered (Internal V_{dd} is off), while the RTC is still powered and running. External V_{dd} may still be powered. Registers described as “As written” are holding the last value software wrote to the register. If the register has not been written since RTC V_{dd} power was applied, then the state is unknown (for all bits of RTC_STAT, RTC_ALARM, and RTC_SWCNT, and for some bits of RTC_ISTAT, RTC_PREN, and RTC_ICTL).

Table 17-1. Effect of States on RTC MMRs

<table>
<thead>
<tr>
<th>RTC V_{dd}</th>
<th>IV_{dd}</th>
<th>System State</th>
<th>RTC_ICTL</th>
<th>RTC_ISTAT</th>
<th>RTC_STAT</th>
<th>RTC_ALARM</th>
<th>RTC_PREN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Off</td>
<td>Off</td>
<td>No Power</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>On</td>
<td>On</td>
<td>Reset</td>
<td>As written</td>
<td>0</td>
<td>Counting</td>
<td>As written</td>
<td></td>
</tr>
<tr>
<td>On</td>
<td>On</td>
<td>Full On</td>
<td>As written</td>
<td>Events</td>
<td>Counting</td>
<td>As written</td>
<td></td>
</tr>
<tr>
<td>On</td>
<td>On</td>
<td>Sleep</td>
<td>As written</td>
<td>Events</td>
<td>Counting</td>
<td>As written</td>
<td></td>
</tr>
<tr>
<td>On</td>
<td>On</td>
<td>Active</td>
<td>As written</td>
<td>Events</td>
<td>Counting</td>
<td>As written</td>
<td></td>
</tr>
<tr>
<td>On</td>
<td>On</td>
<td>Deep Sleep</td>
<td>As written</td>
<td>0</td>
<td>Counting</td>
<td>As written</td>
<td></td>
</tr>
<tr>
<td>On</td>
<td>Off</td>
<td>Off</td>
<td>As written</td>
<td>X</td>
<td>Counting</td>
<td>As written</td>
<td></td>
</tr>
</tbody>
</table>
Table 17-2 summarizes software’s responsibilities with respect to the RTC at various system state transition events.

Table 17-2. RTC System State Transition Events

<table>
<thead>
<tr>
<th>At This Event:</th>
<th>Execute This Sequence:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power On from No Power</td>
<td>Write RTC_PREN = 1.</td>
</tr>
<tr>
<td></td>
<td>Wait for Write Complete.</td>
</tr>
<tr>
<td></td>
<td>Write RTC_STAT to current time.</td>
</tr>
<tr>
<td></td>
<td>Write RTC_ALARM, if needed.</td>
</tr>
<tr>
<td></td>
<td>Write RTC_SWCNT.</td>
</tr>
<tr>
<td></td>
<td>Write RTC_ISTAT to clear any pending RTC events.</td>
</tr>
<tr>
<td></td>
<td>Write RTC_ICTL to enable any desired RTC interrupts or to disable all RTC interrupts.</td>
</tr>
<tr>
<td>Full On after Reset or Full On after Power On from Off</td>
<td>Wait for Seconds Event, or write RTC_PREN = 1 and wait for Write Complete.</td>
</tr>
<tr>
<td></td>
<td>Write RTC_ISTAT to clear any pending RTC events.</td>
</tr>
<tr>
<td></td>
<td>Write RTC_ICTL to enable any desired RTC interrupts or to disable all RTC interrupts.</td>
</tr>
<tr>
<td></td>
<td>Read RTC MMRs as required.</td>
</tr>
<tr>
<td>Wake from Deep Sleep</td>
<td>Wait for Seconds Event flag to set.</td>
</tr>
<tr>
<td></td>
<td>Write RTC_ISTAT to acknowledge RTC Deep Sleep wake-up.</td>
</tr>
<tr>
<td></td>
<td>Read RTC MMRs as required.</td>
</tr>
<tr>
<td></td>
<td>The PLL state is now Active. Transition to Full On as needed.</td>
</tr>
<tr>
<td>Wake from Sleep</td>
<td>If wake-up came from RTC, Seconds Event flag will be set. In this case, write RTC_ISTAT to acknowledge RTC wake-up IRQ. Always, read RTC MMRs as required.</td>
</tr>
<tr>
<td>Before Going to Sleep</td>
<td>If wake-up by RTC is desired:</td>
</tr>
<tr>
<td></td>
<td>Write RTC_ALARM and/or RTC_SWCNT as needed to schedule a wake-up event.</td>
</tr>
<tr>
<td></td>
<td>Write RTC_ICTL to enable the desired RTC interrupt sources for wake-up.</td>
</tr>
<tr>
<td></td>
<td>Wait for Write Complete.</td>
</tr>
<tr>
<td></td>
<td>Enable RTC for wake-up in the System interrupt</td>
</tr>
<tr>
<td></td>
<td>Wake-up Enable register (SIC_IWRx).</td>
</tr>
</tbody>
</table>
### Table 17-2. RTC System State Transition Events (Cont’d)

<table>
<thead>
<tr>
<th>At This Event</th>
<th>Execute This Sequence:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before Going to Deep Sleep</td>
<td>Write RTC_ALARM and/or RTC_SWCNT as needed to schedule a wake-up event. Write RTC_ICTL to enable the desired RTC event sources for Deep Sleep wake-up. Wait for Write Complete.</td>
</tr>
<tr>
<td>Before Going to Off</td>
<td>Write RTC_ALARM and/or RTC_SWCNT as needed to schedule a wake-up event. Write RTC_ICTL to enable any desired RTC event sources for powerup wake-up. Wait for Write Complete. Set the Wake bit in the Voltage Regulator control register (VR_CTL).</td>
</tr>
</tbody>
</table>
18 EXTERNAL BUS INTERFACE UNIT

The external bus interface unit (EBIU) provides glueless interfaces to external memories. The processor supports synchronous DRAM (SDRAM) and is compliant with the PC100 and PC133 SDRAM standards. The EBIU also supports asynchronous interfaces such as SRAM, ROM, FIFOs, flash memory, and ASIC/FPGA designs.

Overview

The EBIU services requests for external memory from the core or from a DMA channel. The priority of the requests is determined by the external bus controller. The address of the request determines whether the request is serviced by the EBIU SDRAM controller or the EBIU asynchronous memory controller.

The EBIU is clocked by the system clock ($\text{SCLK}$). All synchronous memories interfaced to the processor operate at the $\text{SCLK}$ frequency. The ratio between core frequency and $\text{SCLK}$ frequency is programmable using a phase-locked loop (PLL) system memory-mapped register (MMR). For more information, see “Core Clock/System Clock Ratio Control” on page 8-4.

The external memory space is shown in Figure 18-1. One memory region is dedicated to SDRAM support. SDRAM interface timing and the size of the SDRAM region are programmable. The SDRAM memory space can range in size from 16 to 512M byte.
Overview

For information on how to connect to SDRAMs smaller than 16M byte, see “Using SDRAMs Smaller Than 16M Byte” on page 21-8.

The start address of the SDRAM memory space is 0x0000 0000. The area from the end of the SDRAM memory space up to address 0x2000 0000 is reserved.

The next four regions are dedicated to supporting asynchronous memories. Each asynchronous memory region can be independently programmed to support different memory device characteristics. Each region has its own memory select output pin from the EBIU.

The next region is reserved memory space. References to this region do not generate external bus transactions. Writes have no effect on external memory values, and reads return undefined values. The EBIU generates an error response on the internal bus, which will generate a hardware exception for a core access or will optionally generate an interrupt from a DMA channel.

Block Diagram

Figure 18-2 is a conceptual block diagram of the EBIU and its interfaces. Signal names shown with an overbar are active low signals.

Since only one external memory device can be accessed at a time, control, address, and data pins for each memory type are multiplexed together at the pins of the device. The asynchronous memory controller (AMC) and the SDRAM controller (SDC) effectively arbitrate for the shared pin resources.
Figure 18-1. External Memory Map
Internal Memory Interfaces

The EBIU functions as a slave on three buses internal to the processor:

- External access bus (EAB), mastered by the core memory management unit on behalf of external bus requests from the core
- DMA external bus (DEB), mastered by the DMA controller on behalf of external bus requests from any DMA channel
- Peripheral access bus (PAB), mastered by the core on behalf of system MMR requests from the core

These are synchronous interfaces, clocked by \( SCLK \), as are the EBIU and pads registers. The EAB provides access to both asynchronous external memory and synchronous DRAM external memory. The external access is controlled by either the asynchronous memory controller (AMC) or the SDRAM controller (SDC), depending on the internal address used to
access the EBIU. Since the AMC and SDC share the same interface to the external pins, access is sequential and must be arbitrated based on requests from the EAB.

The third bus (PAB) is used only to access the memory-mapped control and status registers of the EBIU. The PAB connects separately to the AMC and SDC; it does not need to arbitrate with or take access cycles from the EAB bus.

The external bus controller (EBC) logic must arbitrate access requests for external memory coming from the EAB and DEB buses. The EBC logic routes read and write requests to the appropriate memory controller based on the bus selects. The AMC and SDC compete for access to the shared resources in the pads logic. This competition is resolved in a pipelined fashion, in the order dictated by the EBC arbiter. Transactions from the core have priority over DMA accesses in most circumstances. However, if the DMA controller detects an excessive backup of transactions, it can request its priority to be temporarily raised above the core.

**External Memory Interfaces**

Both the AMC and the SDC share the external interface address and data pins, as well as some of the control signals. These pins are shared:

- ADDR[19:1], address bus
- DATA[15:0], data bus
- ABE[1:0]/SDQM[1:0], AMC byte enables/SDC data masks
- BR, BG, BGH, external bus access control signals

No other signals are multiplexed between the two controllers.
Table 18-1 and Table 18-2 describe the signals associated with each interface.

### Table 18-1. Asynchronous Memory Interface Signals

<table>
<thead>
<tr>
<th>Pad</th>
<th>Pin Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA[15:0]</td>
<td>I/O</td>
<td>External data bus</td>
</tr>
<tr>
<td>ADDR[19:1]</td>
<td>O</td>
<td>External address bus</td>
</tr>
<tr>
<td>AMS[3:0]</td>
<td>O</td>
<td>Asynchronous memory selects</td>
</tr>
<tr>
<td>AWE</td>
<td>O</td>
<td>Asynchronous memory write enable</td>
</tr>
<tr>
<td>ARE</td>
<td>O</td>
<td>Asynchronous memory read enable</td>
</tr>
<tr>
<td>AOE</td>
<td>O</td>
<td>Asynchronous memory output enable</td>
</tr>
<tr>
<td>ARDY</td>
<td>I</td>
<td>Asynchronous memory ready response</td>
</tr>
<tr>
<td>ABE[1:0]/SDQM[1:0]</td>
<td>O</td>
<td>Byte enables</td>
</tr>
</tbody>
</table>

1 Pin Types: I = Input, O = Output

In most cases, the AOE pin should be connected to the OE pin of an external memory-mapped asynchronous device. Refer to ADSP-BF538/ADSP-BF538F Embedded Processor Data Sheet for specific timing information between the AOE and ARE signals to determine which interface signal should be used in your system.

### Table 18-2. SDRAM Interface Signals

<table>
<thead>
<tr>
<th>Pad</th>
<th>Pin Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA[15:0]</td>
<td>I/O</td>
<td>External data bus</td>
</tr>
<tr>
<td>ADDR[19:18]</td>
<td>O</td>
<td>External address bus</td>
</tr>
<tr>
<td>ADDR[16:1]</td>
<td></td>
<td>Connect to SDRAM Address pins. bank address is output on ADDR[19:18] and should be connected to SDRAM BA[1:0] pins.</td>
</tr>
<tr>
<td>SRAS</td>
<td>O</td>
<td>SDRAM row address strobe pin Connect to the SDRAM RAS pin.</td>
</tr>
</tbody>
</table>
Table 18-2. SDRAM Interface Signals (Cont’d)

<table>
<thead>
<tr>
<th>Pad</th>
<th>Pin Type 1</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCAS</td>
<td>O</td>
<td>SDRAM column address strobe pin Connect to the SDRAM CAS pin.</td>
</tr>
<tr>
<td>SWE</td>
<td>O</td>
<td>SDRAM write enable pin Connect to the SDRAM WE pin.</td>
</tr>
<tr>
<td>ABE[1:0]/SDQM[1:0]</td>
<td>O</td>
<td>SDRAM data mask pins Connect to the SDRAM DQM pins.</td>
</tr>
<tr>
<td>SMS</td>
<td>O</td>
<td>Memory select pin of external memory bank configured for SDRAM Connect to the SDRAM CS (chip select) pin. Active low.</td>
</tr>
<tr>
<td>SA10</td>
<td>O</td>
<td>SDRAM A10 pin SDRAM interface uses this pin to be able to do refreshes while the AMC is using the bus. Connect to the SDRAM A[10] pin.</td>
</tr>
<tr>
<td>SCKE</td>
<td>O</td>
<td>SDRAM clock enable pin Connect to the SDRAM CKE pin.</td>
</tr>
<tr>
<td>CLKOUT</td>
<td>O</td>
<td>SDRAM clock output pin Switches at system clock frequency. Connect to the SDRAM CLK pin.</td>
</tr>
</tbody>
</table>

1 Pin Types: I = Input, O = Output

**EBIU Programming Model**

This section describes the programming model of the EBIU. This model is based on system memory-mapped registers used to program the EBIU.

There are six control registers and one status register in the EBIU. They are:

- Asynchronous memory global control register *(EBIU_AMGCTL)*
- Asynchronous memory bank control 0 register *(EBIU_AMBCTLO)*
- Asynchronous memory bank control 1 register *(EBIU_AMBCTL1)*
Overview

- SDRAM memory global control register (EBIU_SDGCTL)
- SDRAM memory bank control register (EBIU_SDBCTL)
- SDRAM refresh rate control register (EBIU_SDRRC)
- SDRAM control status register (EBIU_SDSTAT)

Each of these registers is described in detail in the AMC and SDC sections later in this chapter.

Error Detection

The EBIU responds to any bus operation which addresses the range of 0x0000 0000 – 0xEEFF FFFF, even if that bus operation addresses reserved or disabled memory or functions. It responds by completing the bus operation (asserting the appropriate number of acknowledges as specified by the bus master) and by asserting the bus error signal for these error conditions:

- Any access to reserved off-chip memory space
- Any access to a disabled external memory bank
- Any access to an unpopulated area of an SDRAM memory bank

If the core requested the faulting bus operation, the bus error response from the EBIU is gated into the HWE interrupt internal to the core (this interrupt can be masked off in the core). If a DMA master requested the faulting bus operation, then the bus error is captured in that controller and can optionally generate an interrupt to the core.
Asynchronous Memory Interface

The asynchronous memory interface allows a glueless interface to a variety of memory and peripheral types. These include SRAM, ROM, EPROM, flash memory, and FPGA/ASIC designs. Four asynchronous memory regions are supported. Each has a unique memory select associated with it, shown in Table 18-3.

Table 18-3. Asynchronous Memory Bank Address Range

<table>
<thead>
<tr>
<th>Memory Bank Select</th>
<th>Address Start</th>
<th>Address End</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMS[3]</td>
<td>2030 0000</td>
<td>203F FFFF</td>
</tr>
<tr>
<td>AMS[1]</td>
<td>2010 0000</td>
<td>201F FFFF</td>
</tr>
<tr>
<td>AMS[0]</td>
<td>2000 0000</td>
<td>200F FFFF</td>
</tr>
</tbody>
</table>

Asynchronous Memory Address Decode

The address range allocated to each asynchronous memory bank is fixed at 1M byte; however, not all of an enabled memory bank need be populated. Unlike the SDRAM memory, which may need to support very large memory structures spanning multiple memory banks, it should be relatively easy to constrain code and data structures to fit within one of the supported asynchronous memory banks, because of the nature of the types of code or data that is stored here.

Note accesses to unpopulated memory of partially populated AMC banks do not result in a bus error and will alias to valid AMC addresses.

The asynchronous memory signals are defined in Table 18-1. The timing of these pins is programmable to allow a flexible interface to devices of different speeds. For example interfaces, see Chapter 21, “System Design”.

---

ADSP-BF538/ADSP-BF538F Blackfin Processor Hardware Reference 18-9
Asynchronous Memory Interface

EBIU_AMGCTL Register

The asynchronous memory global control register (EBIU_AMGCTL), shown in Figure 18-3, configures global aspects of the controller. It contains bank enables and other information as described in this section. This register should not be programmed while the AMC is in use. The EBIU_AMGCTL register should be the last control register written to when configuring the processor to access external memory-mapped asynchronous devices.

For external devices that need a clock, CLKOUT can be enabled by setting the AMCKEN bit in the EBIU_AMGCTL register. In systems that do not use CLKOUT, set the AMCKEN bit to 0.

Figure 18-3. Asynchronous Memory Global Control Register

| AMBEN[2:0] | Enable asynchronous memory banks |
| 000 | All banks disabled |
| 001 | Bank0 enabled |
| 010 | Bank0 and Bank1 enabled |
| 011 | Bank0, Bank1, and Bank2 enabled |
| 1xx | All banks (Bank0, Bank1, Bank2, Bank3) enabled |

| AMCKEN | 0 - Disable CLKOUT for asynchronous memory region accesses |
| 1 - Enable CLKOUT for asynchronous memory region accesses |

| CDPRI0 | 0 - Core has priority over DMA for external accesses |
| 1 - DMA has priority over core for external accesses |

Reset = 0x00F2
EBIU_AMBCTL0 and EBIU_AMBCTL1 Registers

The EBIU asynchronous memory controller has two asynchronous memory bank control registers (`EBIU_AMBCTL0` and `EBIU_AMBCTL1`). (See Figure 18-4 through Figure 18-7.) They contain bits for counters for setup, strobe, and hold time; bits to determine memory type and size; and bits to configure use of `ARDY`. These registers should not be programmed while the AMC is in use.

The timing characteristics of the AMC can be programmed using these four parameters:

- **Setup**: the time between the beginning of a memory cycle (`AMS[x]` low) and the read-enable assertion (`ARE` low) or write-enable assertion (`AWE` low)
- **Read access**: the time between read-enable assertion (`ARE` low) and deassertion (`ARE` high)
- **Write access**: the time between write-enable assertion (`AWE` low) and deassertion (`AWE` high)
- **Hold**: the time between read-enable deassertion (`ARE` high) or write-enable deassertion (`AWE` high) and the end of the memory cycle (`AMS[x]` high)

Each of these parameters can be programmed in terms of EBIU clock cycles. In addition, there are minimum values for these parameters:

- Setup $\geq 1$ cycle
- Read access $\geq 1$ cycle
- Write access $\geq 1$ cycle
- Hold $\geq 0$ cycles
Asynchronous Memory Interface

**Asynchronous Memory Bank Control 0 Register (EBIU_AMBCTL0)**

<table>
<thead>
<tr>
<th>Bit 31–16</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>Not supported</td>
<td></td>
</tr>
<tr>
<td>0001 to 1111</td>
<td>1 to 15 cycles</td>
<td></td>
</tr>
</tbody>
</table>

**B1WAT[3:0]**
bank 1 write access time (number of cycles AWE is held asserted)
0000 - Not supported
0001 to 1111 - 1 to 15 cycles

**B1RAT[3:0]**
bank 1 read access time (number of cycles ARE is held asserted)
0000 - Not supported
0001 to 1111 - 1 to 15 cycles

**B1HT[1:0]**
bank 1 hold time (number of cycles between AWE or ARE deasserted, and AOE deasserted)
00 - 0 cycles
01 - 1 cycle
10 - 2 cycles
11 - 3 cycles

**B1ST[1:0]**
bank 1 setup time (number of cycles after AOE asserted, before AWE or ARE asserted)
00 - 4 cycles
01 - 1 cycle
10 - 2 cycles
11 - 3 cycles

**B1RDYEN**
bank 1 ARDY enable
0 - Ignore ARDY for accesses to this memory bank
1 - After access time countdown, use state of ARDY to determine completion of access

**B1RDYPOL**
bank 1 ARDY polarity
0 - Transaction completes if ARDY sampled low
1 - Transition completes if ARDY sampled high

**B1TT[1:0]**
bank 1 memory transition time (number of cycles inserted after a read access to this bank, and before a write access to this bank or a read access to another bank)
00 - 4 cycles for bank transition
01 - 1 cycle for bank transition
10 - 2 cycles for bank transition
11 - 3 cycles for bank transition

Reset = 0xFFC2 FFC20xFFC0 0A04

Figure 18-4. Asynchronous Memory Bank Control 0 Register (Bits 31–16)
Asynchronous Memory Bank Control 0 Register (EBIU_AMBCTL0)

**0xFFC0 0A04**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>B0RDYPOL</td>
</tr>
<tr>
<td>14</td>
<td>B0RT[1:0]</td>
</tr>
<tr>
<td>13</td>
<td>B0ST[1:0]</td>
</tr>
<tr>
<td>12</td>
<td>B0WAT[3:0]</td>
</tr>
<tr>
<td>11</td>
<td>B0RDYEN</td>
</tr>
<tr>
<td>10</td>
<td>B0HT[1:0]</td>
</tr>
<tr>
<td>9</td>
<td>Bank 0 write access time (number of cycles AWE is held asserted)</td>
</tr>
<tr>
<td>8</td>
<td>Bank 0 read access time (number of cycles ARE is held asserted)</td>
</tr>
<tr>
<td>7</td>
<td>Bank 0 hold time (number of cycles between AWE or ARE deasserted, and AOE deasserted)</td>
</tr>
<tr>
<td>6</td>
<td>Bank 0 setup time (number of cycles after AOE asserted, before AWE or ARE asserted)</td>
</tr>
<tr>
<td>5</td>
<td>Bank 0 memory transition time (number of cycles inserted after a read access to this bank, and before a write access to this bank or a read access to another bank)</td>
</tr>
</tbody>
</table>

**Reset = 0xFFC2 FFC2**

- **B0RDYPOL**: bank 0 ARDY polarity
  - 0: Transaction completes if ARDY sampled low
  - 1: Transition completes if ARDY sampled high

- **B0RT[1:0]**: bank 0 ARDY enable
  - 0: Ignore ARDY for accesses to this memory bank
  - 1: After access time countdown, use state of ARDY to determine completion of access

- **B0ST[1:0]**: bank 0 ARDY enable
  - 00: ARDY is not supported
  - 01 to 1111: 1 to 15 cycles

- **B0WAT[3:0]**: bank 0 write access time (number of cycles AWE is held asserted)
  - 0000: ARDY is not supported
  - 0001 to 1111: 1 to 15 cycles

- **B0RDYEN**: bank 0 ARDY enable
  - 0: Ignore ARDY for accesses to this memory bank
  - 1: After access time countdown, use state of ARDY to determine completion of access

- **B0HT[1:0]**: bank 0 hold time (number of cycles between AWE or ARE deasserted, and AOE deasserted)
  - 00: 0 cycles
  - 01: 1 cycle
  - 10: 2 cycles
  - 11: 3 cycles

- **B0ST[1:0]**: bank 0 setup time (number of cycles after AOE asserted, before AWE or ARE asserted)
  - 00: 4 cycles
  - 01: 1 cycle
  - 10: 2 cycles
  - 11: 3 cycles

Figure 18-5. Asynchronous Memory Bank Control 0 Register (Bits 15–0)
**Asynchronous Memory Bank Control 1 Register (EBIU_AMBCTL1)**

<table>
<thead>
<tr>
<th>Bit 31</th>
<th>Bit 30</th>
<th>Bit 29</th>
<th>Bit 28</th>
<th>Bit 27</th>
<th>Bit 26</th>
<th>Bit 25</th>
<th>Bit 24</th>
<th>Bit 23</th>
<th>Bit 22</th>
<th>Bit 21</th>
<th>Bit 20</th>
<th>Bit 19</th>
<th>Bit 18</th>
<th>Bit 17</th>
<th>Bit 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Reset = 0xFFC2 FFC2**

**B3WAT[3:0]**
- Bank 3 write access time (number of cycles AWE is held asserted)
- 0000 - Not supported
- 0001 to 1111 - 1 to 15 cycles

**B3RAT[3:0]**
- Bank 3 read access time (number of cycles ARE is held asserted)
- 0000 - Not supported
- 0001 to 1111 - 1 to 15 cycles

**B3HT[1:0]**
- Bank 3 hold time (number of cycles between AWE or ARE deasserted, and AOE deasserted)
- 00 - 0 cycles
- 01 - 1 cycle
- 10 - 2 cycles
- 11 - 3 cycles

**B3ST[1:0]**
- Bank 3 setup time (number of cycles after AOE asserted, before AWE or ARE asserted)
- 00 - 4 cycles
- 01 - 1 cycle
- 10 - 2 cycles
- 11 - 3 cycles

**B3RDYEN**
- Bank 3 ARDY enable
  - 0 - Ignore ARDY for accesses to this memory bank
  - 1 - After access time countdown, use state of ARDY to determine completion of access

**B3RDYPOL**
- Bank 3 ARDY polarity
  - 0 - Transaction completes if ARDY sampled low
  - 1 - Transition completes if ARDY sampled high

**B3TT[1:0]**
- Bank 3 memory transition time (number of cycles inserted after a read access to this bank, and before a write access to this bank or a read access to another bank)
  - 00 - 4 cycles for bank transition
  - 01 - 1 cycle for bank transition
  - 10 - 2 cycles for bank transition
  - 11 - 3 cycles for bank transition

---

**Figure 18-6. Asynchronous Memory Bank Control 1 Register (Bits 31–16)**
Avoiding Bus Contention

Because the three-stated data bus is shared by multiple devices in a system, be careful to avoid contention. Contention causes excessive power dissipation and can lead to device failure. Contention occurs during the time one device is getting off the bus and another is getting on. If the first device is slow to three-state and the second device is quick to drive, the devices contend.

There are two cases where contention can occur. The first case is a read followed by a write to the same memory space. In this case, the data bus drivers can potentially contend with those of the memory device addressed.
by the read. The second case is back-to-back reads from two different memory spaces. In this case, the two memory devices addressed by the two reads could potentially contend at the transition between the two read operations.

To avoid contention, program the turnaround time (bank transition time) appropriately in the asynchronous memory bank control registers. This feature allows software to set the number of clock cycles between these types of accesses on a bank-by-bank basis. Minimally, the EBIU provides one cycle for the transition to occur.

**ARDY Input Control**

Each bank can be programmed to sample the ARDY input after the read or write access timer has counted down or to ignore this input signal. If enabled and disabled at the sample window, ARDY can be used to extend the access time as required. Note ARDY is synchronously sampled, therefore:

- Assertion and deassertion of ARDY to the processor must meet the data sheet setup and hold times. Failure to meet these synchronous specifications could result in metastable behavior internally. The processor’s CLKOUT signal should be used to ensure synchronous transitions of ARDY.

- The ARDY pin must be stable (either asserted or deasserted) at the external interface on the cycle before the internal bank counter reaches 0; that is, more than one CLKOUT cycle before the scheduled rising edge of AWE or ARE. This will determine whether the access is extended or not.

- Once the transaction has been extended as a result of ARDY being sampled in the “busy” state, the transaction will then complete in the cycle after ARDY is subsequently sampled in the “ready” state.
The polarity of ARDY is programmable on a per-bank basis. Since ARDY is not sampled until an access is in progress to a bank in which the ARDY enable is asserted, ARDY does not need to be driven by default. For more information, see “Adding Additional Wait States” on page 18-21.

**Programmable Timing Characteristics**

This section describes the programmable timing characteristics for the EBIU. Timing relationships depend on the programming of the AMC, whether initiation is from the core or from memory DMA (MemDMA), and the sequence of transactions (read followed by read, read followed by write, and so on).

**Asynchronous Accesses by Core Instructions**

Some external memory accesses are caused by core instructions of the type:

\[
\text{RO.L} = \text{W}[P0++] ; /* \text{Read from external memory, where P0 points to a location in external memory */}
\]

or:

\[
\text{W}[P0++] = \text{RO.L} ; /* \text{Write to external memory */}
\]

**Asynchronous Reads**

*Figure 18-8* shows an asynchronous read bus cycle with timing programmed as setup = 2 cycles, read access = 2 cycles, hold = 1 cycle, and transition time = 1 cycle.

Asynchronous read bus cycles proceed as follows.

1. At the start of the setup period, \text{AMS}[x], the address bus, and \text{ABE}[1:0] become valid, and \text{AOE} asserts.

2. At the beginning of the read access period and after the 2 setup cycles, \text{ARE} asserts.
3. At the beginning of the hold period, read data is sampled on the rising edge of the EBIU clock. The \text{ARE} pin deasserts after this rising edge.

4. At the end of the hold period, \text{AOE} deasserts unless this bus cycle is followed by another asynchronous read to the same memory space. Also, \text{AMS}[x] deasserts unless the next cycle is to the same memory bank.

5. Unless another read of the same memory bank is queued internally, the AMC appends the programmed number of memory transition time cycles.

Figure 18-8. Asynchronous Read Bus Cycles
Read access is completed with the AMSx and AOE signals getting deasserted. There are a few idle cycles before the next read operation starts. The number of idle cycles is a function of the CCLK/SCLK ratio. The number of idle cycles is 6 for a CCLK/SCLK ratio of 3, 4 for a CCLK/SCLK ratio of 5, and 3 for a CCLK/SCLK ratio of 10.

Asynchronous Writes

Figure 18-9 shows an asynchronous write bus cycle followed by an asynchronous read cycle to the same bank, with timing programmed as setup = 2 cycles, write access = 2 cycles, read access = 3 cycles, hold = 1 cycle, and transition time = 1 cycle.

Asynchronous write bus cycles proceed as follows.

1. At the start of the setup period, AMS[x], the address bus, data buses, and ABE[1:0] become valid.
2. At the beginning of the write access period, AWE asserts.
3. At the beginning of the hold period, AWE deasserts.

Asynchronous read bus cycles proceed as follows.

1. At the start of the setup period, AMS[x], the address bus, and ABE[1:0] become valid, and AOE asserts.
2. At the beginning of the read access period, ARE asserts.
3. At the beginning of the hold period, read data is sampled on the rising edge of the EBIU clock. The ARE signal deasserts after this rising edge.
4. At the end of the hold period, \( AOE \) deasserts unless this bus cycle is followed by another asynchronous read to the same memory space. Also, \( AMS[x] \) deasserts unless the next cycle is to the same memory bank.

5. Unless another read of the same memory bank is queued internally, the AMC appends the programmed number of memory transition time cycles.

Figure 18-9. Asynchronous Write and Read Bus Cycles
Adding Additional Wait States

The ARDY pin is used to insert extra wait states. The input is sampled synchronously with the EBIU internal clock. The EBIU starts sampling ARDY on the clock cycle before the end of the programmed strobe period. If ARDY is sampled as deasserted, the access period is extended. The ARDY pin is then sampled on each subsequent clock edge. Read data is latched on the clock edge after ARDY is sampled as asserted. The read- or write-enable remains asserted for one clock cycle after ARDY is sampled as asserted. An example of this behavior is shown in Figure 18-10, where setup = 2 cycles, read access = 4 cycles, and hold = 1 cycle. Note the read access period must be programmed to a minimum of two cycles to make use of the ARDY input.

Byte Enables

The $\text{ABE}[1:0]$ pins are both low during all asynchronous reads and 16-bit asynchronous writes. When an asynchronous write is made to the upper byte of a 16-bit memory, $\text{ABE}_1 = 0$ and $\text{ABE}_0 = 1$. When an asynchronous write is made to the lower byte of a 16-bit memory, $\text{ABE}_1 = 1$ and $\text{ABE}_0 = 0$.

On-Chip Flash Memory

The ADSP-BF538F4 and ADSP-BF538F8 processors provide on-chip flash memory options. This flash memory is a separate die inside the package, and it can be mapped to any of the four asynchronous memory banks by connecting the $\text{FCE}$ pin to the appropriate $\text{AMS}_x$ pin. If the $\text{FCE}$ pin is connected to $\text{AMS}_0$ pin, the processor will boot from the on-chip flash memory.
The SDRAM controller (SDC) enables the processor to transfer data to and from synchronous DRAM (SDRAM) with a maximum frequency specified in *ADSP-BF538/ADSP-BF538F Embedded Processor Data Sheet*.
The processor supports a glueless interface with one external bank of standard SDRAMs of 64 Mbit to 512 Mbit, with configurations x4, x8, and x16, up to a maximum total capacity of 512M bytes of SDRAM. This bank is controlled by the SMS memory select pin. The interface includes timing options to support additional buffers between the processor and SDRAM, to handle the capacitive loads of large memory arrays.

All inputs are sampled and all outputs are valid on the rising edge of the SDRAM clock output CLKOUT.

The EBIU SDC provides a glueless interface with standard SDRAMs. The SDRAM controller:

- Supports SDRAMs of 64M bit, 128M bit, 256M bit, and 512M bit with configurations of x4, x8, and x16
- Supports up to 512M bytes of SDRAM in external SDRAM
- Supports SDRAM page sizes of 512 bytes, 1K byte, 2K byte, and 4K byte
- Supports four internal banks within the SDRAM
- Uses a programmable refresh counter to coordinate between varying clock frequencies and the refresh rate required by the SDRAM.
- Provides multiple timing options to support additional buffers between the processor and SDRAM
- Uses a separate pin (SA10) that enables the SDC to pre-charge SDRAM before issuing an auto-refresh or self-refresh command while the asynchronous memory controller has control of the EBIU port
- Supports self-refresh for standard SDRAMs and partial array self-refresh for mobile SDRAMs
SDRAM Controller (SDC)

- Provides two SDRAM power-up options
- Supports interleaved SDRAM bank accesses

**Definition of Terms**

The following are definitions used in the remainder of this chapter.

**Bank Activate Command**

The bank activate command causes the SDRAM to open an internal bank (specified by the bank address) in a row (specified by the row address). When the bank activate command is issued to the SDRAM, the SDRAM opens a new row address in the dedicated bank. The memory in the open internal bank and row is referred to as the open page. The bank activate command must be applied before a read or write command.

**Burst Length**

The burst length determines the number of words that the SDRAM device stores or delivers after detecting a single write or read command, respectively. The burst length is selected by writing certain bits to the SDRAM mode register during the SDRAM power-up sequence.

Although the SDC supports only burst length = 1 mode, during a burst to SDRAM, the SDC applies the read or write command every cycle and keeps accessing the data. Therefore, the effective burst length is much greater than 1. In other words, setting burst length = 1 does not reduce the performance throughput.
Burst Stop Command

The burst stop command is one of several ways to terminate or interrupt a burst read or write operation.

Since the SDRAM burst length is always hard wired to be 1, the SDC does not support the burst stop command.

Burst Type

The burst type determines the address order in which the SDRAM delivers burst data after detecting a read command or stores burst data after detecting a write command. The burst type is programmed in the SDRAM during the SDRAM power-up sequence.

Since the SDRAM burst length is always programmed to be 1, the burst type does not matter. However, the SDC always sets the burst type to sequential-accesses-only during the SDRAM power-up sequence.

CAS Latency (CL)

The column address strobe (CAS) latency is the delay in clock cycles between when the SDRAM detects the read command and when it provides the data at its output pins. The CAS latency is programmed in the SDRAM mode register during the power-up sequence.

The speed grade of the SDRAM and the application’s clock frequency determine the value of the CAS latency. The SDC can support CAS latency of two or three clock cycles. The selected CAS latency value must be programmed into the SDRAM memory global control register (EBIU_SDGCTL) before the SDRAM power-up sequence. See “EBIU_SDGCTL Register” on page 18-33.
**SDRAM Controller (SDC)**

**CBR (CAS Before RAS) Refresh or Auto-Refresh**

When the SDC refresh counter times out, the SDC pre-charges all four banks of SDRAM and then issues an auto-refresh command to them. This causes the SDRAMs to generate an internal CBR refresh cycle. When the internal refresh completes, all four internal SDRAM banks are pre-charged.

**DQM Pin Mask Function**

The $SDQM[1:0]$ pins provide a byte-masking capability on 8-bit writes to SDRAM. The $DQM$ pins are used to block the input buffer of the SDRAM during partial write operations. The $SDQM[1:0]$ pins are not used to mask data on partial read cycles. For write cycles, the data masks have a latency of zero cycles, permitting data writes when the corresponding $SDQM[x]$ pin is sampled low and blocking data writes when the $SDQM[x]$ pin is sampled high on a byte-by-byte basis.

**Internal Bank**

There are several internal memory banks on a given SDRAM. The SDC supports interleaved accesses among the internal banks. The bank address can be thought of as part of the row address. The SDC assumes that all SDRAMs to which it interfaces have four internal banks and allows each activated bank to have a unique row address.

**Mode Register**

SDRAM devices contain an internal configuration register which allows specification of the SDRAM device’s functionality. After power-up and before executing a read or write to the SDRAM memory space, the application must trigger the SDC to write to the SDRAM mode register. The write to the SDRAM mode register is triggered by writing a 1 to the $PSSE$ bit in the SDRAM memory global control register ($EBIU_SDGCTL$) and then issuing a read or write transfer to the SDRAM address space. The
initial read or write triggers the SDRAM power-up sequence to be run, which programs the SDRAM mode register with the CAS latency from the EBIU_SDGCTL register. This initial read or write to SDRAM takes many cycles to complete.

Note for most applications, the SDRAM power-up sequence and writing of the mode register needs to be done only once. Once the power-up sequence has completed, the PSSE bit should not be set again unless a change to the mode register is desired. In this case, refer to “Managing SDRAM Refresh During PLL Transitions” on page 21-8.

Low power SDRAM devices may also contain an extended mode register. The EBIU enables programming of the extended mode register during power-up via the EMREN bit in the EBIU_SDGCTL register.

Page Size

Page size is the amount of memory which has the same row address and can be accessed with successive read or write commands without needing to activate another row. The page size can be calculated for 16-bit SDRAM banks with this formula:

- 16-bit SDRAM banks: page size = $2^{(CAW + 1)}$

where CAW is the column address width of the SDRAM, plus 1 because the SDRAM bank is 16 bits wide (1 address bit = 2 bytes).

Pre-Charge Command

The pre-charge command closes a specific internal bank in the active page or all internal banks in the page.
SDRAM Bank

The SDRAM bank is a region of memory that can be configured to 16M byte, 32M byte, 64M byte, 128M byte, 256M byte, or 512M byte and is selected by the SMS pin.

Do not confuse the “SDRAM internal banks” which are internal to the SDRAM and are selected with the bank address, with the “SDRAM bank” or “external bank” that is enabled by the SMS pin.

Self-Refresh

When the SDRAM is in self-refresh mode, the SDRAM internal timer initiates auto-refresh cycles periodically, without external control input. The SDC must issue a series of commands including the self-refresh command to put the SDRAM into this low power mode, and it must issue another series of commands to exit self-refresh mode. Entering self-refresh mode is programmable in the SDRAM memory global control register (EBIU_SDGCTL) and any access to the SDRAM address space causes the SDC to exit the SDRAM from self-refresh mode. See “Entering and Exiting Self-Refresh Mode (SRFS)” on page 18-38.

\[t_{\text{RAS}}\]

This is the required delay between issuing a bank activate command and issuing a pre-charge command, and between the self-refresh command and the exit from self-refresh. The \[t_{\text{RAS}}\] bit field in the SDRAM memory global control register (EBIU_SDGCTL) is 4 bits wide and can be programmed to be 1 to 15 clock cycles long. “Selecting the Bank Activate Command Delay (TRAS)” on page 18-41.
$t_{RC}$

This is the required delay between issuing successive bank activate commands to the same SDRAM internal bank. This delay is not directly programmable. The $t_{RC}$ delay must be satisfied by programming the $T_{RAS}$ and $T_{RP}$ fields to ensure that $t_{RAS} + t_{RP} \geq t_{RC}$.

$t_{RCD}$

This is the required delay between a bank activate command and the start of the first read or write command. The $T_{RCD}$ bit field in the SDRAM memory global control register ($E_{BIU\_SDGCTL}$) is 3 bits wide and can be programmed to be from 1 to 7 clock cycles long.

$t_{RFC}$

This is the required delay between issuing an auto-refresh command and a bank activate command and between issuing successive auto-refresh commands. This delay is not directly programmable and is assumed to be equal to $t_{RC}$. The $t_{RC}$ delay must be satisfied by programming the $T_{RAS}$ and $T_{RP}$ fields to ensure that $t_{RAS} + t_{RP} \geq t_{RC}$.

$t_{RP}$

This is the required delay between issuing a pre-charge command and:

- issuing a bank activate command
- issuing an auto-refresh command
- issuing a self-refresh command

The $T_{RP}$ bit field in the SDRAM memory global control register ($E_{BIU\_SDGCTL}$) is 3 bits wide and can be programmed to be 1 to 7 clock cycles long. “Selecting the Pre-Charge Delay (TRP)” on page 18-42.
SDRAM Controller (SDC)

$t_{RRD}$

This is the required delay between issuing a bank A activate command and a bank B activate command. This delay is not directly programmable and is assumed to be $t_{RCD} + 1$.

$t_{WR}$

This is the required delay between a write command (driving write data) and a pre-charge command. The TWR bit field in the SDRAM memory global control register (EBIU_SDGCTL) is 2 bits wide and can be programmed to be from 1 to 3 clock cycles long.

$t_{XSR}$

This is the required delay between exiting self-refresh mode and issuing the auto-refresh command. This delay is not directly programmable and is assumed to be equal to $t_{RC}$. The $t_{RC}$ delay must be satisfied by programming the TRAS and TRP fields to ensure that $t_{RAS} + t_{RP} \geq t_{RC}$.

SDRAM Configurations Supported

Table 18-4 shows all possible bank sizes, bank widths and SDRAM discrete component configurations that can be gluelessly interfaced to the SDC.
Table 18-4. SDRAM Discrete Component Configurations Supported

<table>
<thead>
<tr>
<th>System Size (M byte)</th>
<th>System Size (M bit)</th>
<th>SDRAM Configuration</th>
<th>Number of Chips</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>4M x 16</td>
<td>4M x 4</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>4M x 16</td>
<td>4M x 16</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>8M x 16</td>
<td>8M x 8</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>8M x 16</td>
<td>8M x 16</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td>16M x 16</td>
<td>16M x 4</td>
<td>4</td>
</tr>
<tr>
<td>32</td>
<td>16M x 16</td>
<td>16M x 8</td>
<td>2</td>
</tr>
<tr>
<td>32</td>
<td>16M x 16</td>
<td>16M x 16</td>
<td>1</td>
</tr>
<tr>
<td>64</td>
<td>32M x 16</td>
<td>32M x 4</td>
<td>4</td>
</tr>
<tr>
<td>64</td>
<td>32M x 16</td>
<td>32M x 8</td>
<td>2</td>
</tr>
<tr>
<td>64</td>
<td>32M x 16</td>
<td>32M x 16</td>
<td>1</td>
</tr>
<tr>
<td>128</td>
<td>64M x 16</td>
<td>64M x 4</td>
<td>4</td>
</tr>
<tr>
<td>128</td>
<td>64M x 16</td>
<td>64M x 8</td>
<td>2</td>
</tr>
<tr>
<td>128</td>
<td>64M x 16</td>
<td>64M x 16</td>
<td>1</td>
</tr>
</tbody>
</table>

Example SDRAM System Block Diagrams

Figure 18-11 shows a block diagram of the SDRAM interface. In this example, the SDRAM interface connects to two 64M bit (x8) SDRAM devices to form one external bank of 16M bytes of memory. The same address and control bus feeds both SDRAM devices.

The SDC includes a separate address pin (SA10) to enable the execution of auto-refresh commands in parallel with any asynchronous memory access. This separate pin allows the SDC to issue a pre-charge command to the SDRAM before it issues an auto-refresh command.
In addition, the SA10 pin allows the SDC to enter and exit self-refresh mode in parallel with any asynchronous memory access. The SA10 pin (instead of the ADDR[11] pin) should be directly connected to the SDRAM A10 pin. During the pre-charge command, SA10 is used to indicate that a pre-charge all should be done. During a bank activate command, SA10 outputs the internal row address bit, which should be multiplexed to the A10 SDRAM input. During read and write commands, SA10 is used to disable the auto-pre-charge function of SDRAMs.


![Figure 18-11. 16M Byte SDRAM System Example](image-url)
External Bus Interface Unit

Executing a Parallel Refresh Command

The SDC includes a separate address pin (SA10) to enable the execution of auto-refresh commands in parallel with any asynchronous memory access. This separate pin allows the SDC to issue a pre-charge command to the SDRAM before it issues an auto-refresh command. In addition, the SA10 pin allows the SDC to enter and exit self-refresh mode in parallel with any asynchronous memory access.

The SA10 pin should be directly connected to the A10 pin of the SDRAM (instead of to the ADDR[10] pin). During the pre-charge command, SA10 is used to indicate that a pre-charge all should be done. During a bank activate command, SA10 outputs the internal row address bit, which should be multiplexed to the A10 SDRAM input. During read and write commands, SA10 is used to disable the auto-pre-charge function of SDRAMs.

EBIU_SDGCTL Register

The SDRAM memory global control register (EBIU_SDGCTL) includes all programmable parameters associated with the SDRAM access timing and configuration. Figure 18-12 shows the EBIU_SDGCTL register bit definitions.

The SCTLE bit is used to enable or disable the SDC. If SCTLE is disabled, any access to SDRAM address space generates an internal bus error, and the access does not occur externally. For more information, see “Error Detection” on page 18-8. When SCTLE is disabled, all SDC control pins are in their inactive states and the SDRAM clock is not running. The SCTLE bit must be enabled for SDC operation and is enabled by default at reset.
Figure 18-12. SDRAM Memory Global Control Register
The CAS latency (CL), SDRAM t_{RAS} timing (TRAS), SDRAM t_{RP} timing (TRP), SDRAM t_{RCD} timing (TRCD), and SDRAM t_{WR} timing (TWR) bits should be programmed based on the system clock frequency and the timing specifications of the SDRAM used.

The user must ensure that \( t_{RAS} + t_{RP} \geq \max(t_{RC}, t_{RFC}, t_{XSR}) \).

The PSM and PSSE bits work together to specify and trigger an SDRAM power-up (initialization) sequence. If the PSM bit is set to 1, the SDC does a pre-charge all command, followed by a load mode register command, and then does eight auto-refresh cycles. If the PSM bit is cleared, the SDC does a pre-charge all command, followed by eight auto-refresh cycles, and then a load mode register command. Two events must occur before the SDC does the SDRAM power-up sequence:

- The PSSE bit must be set to 1 to enable the SDRAM power-up sequence.
- A read or write access must be done to enabled SDRAM address space in order to have the external bus granted to the SDC so that the SDRAM power-up sequence may occur.

The SDRAM power-up sequence occurs and is followed immediately by the read or write transfer to SDRAM that was used to trigger the SDRAM power-up sequence. Note there is a latency for this first access to SDRAM because the SDRAM power-up sequence takes many cycles to complete.

Before executing the SDC power-up sequence, ensure that the SDRAM receives stable power and is clocked for the proper amount of time, as specified by the SDRAM specification.

The power-up start delay bit (PUPSD) optionally delays the power-up start sequence for 15 SCLK cycles. This is useful for multiprocessor systems sharing an external SDRAM. If the bus has been previously granted to the other processor before power-up and self-refresh mode is used when switching bus ownership, then the PUPSD bit can be used to guarantee a
sufficient period of inactivity from self-refresh to the first pre-charge command in the power-up sequence in order to meet the exit self-refresh time ($t_{XSR}$) of the SDRAM.

When the SRFS bit is set to 1, self-refresh mode is triggered. Once the SDC completes any active transfers, the SDC executes the sequence of commands to put the SDRAM into self-refresh mode. The next access to an enabled SDRAM bank causes the SDC to execute the commands to exit the SDRAM from self-refresh and execute the access. See “Entering and Exiting Self-Refresh Mode (SRFS)” on page 18-38 for more information about the SRFS bit.

The EBUFE bit is used to enable or disable external buffer timing. When buffered SDRAM modules or discrete register-buffers are used to drive the SDRAM control inputs, EBUFE should be set to 1. Using this setting adds a cycle of data buffering to read and write accesses. See “Setting the SDRAM Buffering Timing Option (EBUFE)” on page 18-39 for more information about the EBUFE bit.

The FBBRW bit enables an SDRAM read followed by write to occur on consecutive cycles. In many systems, this is not possible because the turn-off time of the SDRAM data pins is too long, leading to bus contention with the succeeding write from the processor. When this bit is 0, a clock cycle is inserted between read accesses followed immediately by write accesses.

The EMREN bit enables programming of the extended mode register during startup. The extended mode register is used to control SDRAM power consumption in certain mobile low power SDRAMs. If the EMREN bit is enabled, then the TCSR and PASR[1:0] bits control the value written to the extended mode register. The PASR bits determine how many SDRAM internal banks are refreshed during self-refresh. The TCSR bit signals to the SDRAM the worst case temperature range for the system, and thus how often the SDRAM internal banks need to be refreshed during self-refresh.
The $\text{CDDBG}$ bit is used to enable or disable the SDRAM control signals when the external memory interface is granted to an external controller. If this bit is set to a 1, then the control signals are three-stated when bus grant is active. Otherwise, these signals continue to be driven during grant. If the bit is set and the external bus is granted, all SDRAM internal banks are assumed to have been changed by the external controller. This means a pre-charge is required on each bank prior to use after control of the external bus is re-established. The control signals affected by this pin are $\text{SRAS, SCAS, SWE, SMS, SA10, SCKE, and CLKOUT}$.

Note all reserved bits in this register must always be written with 0s.

### Setting the SDRAM Clock Enable (SCTLE)

The $\text{SCTLE}$ bit allows software to disable all SDRAM control pins. These pins are $\text{SDQM[3:0], SCAS, SRAS, SWE, SCKE, and CLKOUT}$.

- $\text{SCTLE} = 0$
  
  Disable all SDRAM control pins (control pins negated, $\text{CLKOUT}$ low)

- $\text{SCTLE} = 1$
  
  Enable all SDRAM control pins ($\text{CLKOUT}$ toggles)

Note the $\text{CLKOUT}$ function is also shared with the AMC. Even if $\text{SCTLE}$ is disabled, $\text{CLKOUT}$ can be enabled independently by the $\text{CLKOUT}$ enable in the AMC ($\text{AMCKEN}$ in the $\text{EBIU_AMGCTL}$ register).

If the system does not use SDRAM, $\text{SCTLE}$ should be set to 0.

If an access occurs to the SDRAM address space while $\text{SCTLE}$ is 0, the access generates an internal bus error and the access does not occur externally. For more information, see “Error Detection” on page 18-8. With careful software control, the $\text{SCTLE}$ bit can be used in conjunction with
self-refresh mode to further lower power consumption. However, SCTLE must remain enabled at all times when the SDC is needed to generate auto-refresh commands to SDRAM.

**Entering and Exiting Self-Refresh Mode (SRFS)**

The SDC supports SDRAM self-refresh mode. In self-refresh mode, the SDRAM performs refresh operations internally—without external control—reducing the SDRAM power consumption.

The **SRFS** bit in **EBIU_SDGCTL** enables the start of self-refresh mode:

- **SRFS = 0**
  
  Disable self-refresh mode

- **SRFS = 1**

  Enable self-refresh mode

When **SRFS** is set to 1, once the SDC enters an idle state it issues a pre-charge command if necessary, and then issues a self-refresh command. If an internal access is pending, the SDC delays issuing the self-refresh command until it completes the pending SDRAM access and any subsequent pending access requests. Refer to “SDC Commands” on page 18-55 for more information.

Once the SDRAM device enters into self-refresh mode, the SDRAM controller asserts the **SDSRA** bit in the SDRAM control status register (**EBIU_SDSTAT**).
The SDRAM device exits self-refresh mode only when the SDC receives a core or DMA access request. In conjunction with the SRFS bit, 2 possibilities are given to exit the self-refresh mode:

- If SRFS bit is set before the request, the SDC exits self-refresh and remains in auto-refresh mode.
- If SRFS bit is cleared before the request, the SDC exits self-refresh only for a single request and returns back to self-refresh mode until a new request is coming.

Note once the SRFS bit is set to 1, the SDC enters self-refresh mode when it finishes pending accesses. There is no way to cancel the entry to self-refresh mode.

**Setting the SDRAM Buffering Timing Option (EBUFE)**

To meet overall system timing requirements, systems that employ several SDRAM devices connected in parallel may require buffering between the processor and multiple SDRAM devices. This buffering generally consists of a register and driver.

To meet such timing requirements and to allow intermediary registration, the SDC supports pipelining of SDRAM address and control signals.

The EBUFE bit in the EBIU_SDGCTL register enables this mode:

- **EBUFE = 0**
  
  Disable external buffering timing

- **EBUFE = 1**
  
  Enable external buffering timing
When $EBUF = 1$, the SDRAM controller delays the data in write accesses by one cycle, enabling external buffer registers to latch the address and controls. In read accesses, the SDRAM controller samples data one cycle later to account for the one-cycle delay added by the external buffer registers. When external buffering timing is enabled, the latency of all accesses is increased by one cycle.

**Selecting the CAS Latency Value (CL)**

The CAS latency value defines the delay, in number of clock cycles, between the time the SDRAM detects the read command and the time it provides the data at its output pins.

CAS latency does not apply to write cycles.

The $CL$ bits in the SDRAM memory global control register ($EBIU_SDGCTL$) select the CAS latency value:

- $CL = 00$
  
  Reserved
- $CL = 01$
  
  Reserved
- $CL = 10$
  
  2 clock cycles
- $CL = 11$
  
  3 clock cycles

Generally, the frequency of operation determines the value of the CAS latency. For specific information about setting this value, consult the SDRAM device documentation.
Selecting the Bank Activate Command Delay (TRAS)

The $t_{RAS}$ value (bank activate command delay) defines the required delay, in number of clock cycles, between the time the SDC issues a bank activate command and the time it issues a pre-charge command. The SDRAM must also remain in self-refresh mode for at least the time period specified by $t_{RAS}$. The $t_{RP}$ and $t_{RAS}$ values define the $t_{RFC}$, $t_{RC}$, and $t_{XSR}$ values. See definition on page 18-28 for more information.

The $t_{RAS}$ parameter allows the processor to adapt to the timing requirements of the system’s SDRAM devices.

The $TRAS$ bits in the SDRAM memory global control register ($EBIU_{SDGCTL}$) select the $t_{RAS}$ value. Any value between 1 and 15 clock cycles can be selected. For example:

- $TRAS = 0000$  
  No effect
- $TRAS = 0001$  
  1 clock cycle
- $TRAS = 0010$  
  2 clock cycles
- $TRAS = 1111$  
  15 clock cycles

For specific information on setting this value, consult the SDRAM device documentation.
Selecting the RAS to CAS Delay (TRCD)

The $t_{RCD}$ value (RAS to CAS delay) defines the delay for the first read or write command after a row activate command, in number of clock cycles. The $t_{RCD}$ parameter allows the processor to adapt to the timing requirements of the system’s SDRAM devices.

The $t_{RCD}$ bits in the SDRAM memory global control register (EBIU_SDGCTL) select the $t_{RCD}$ value. Any value between 1 and 7 clock cycles may be selected. For example:

- $TRCD = \text{reserved}$
  
  No effect

- $TRCD = 001$
  
  1 clock cycle

- $TRCD = 010$
  
  2 clock cycles

- $TRCD = 111$
  
  7 clock cycles

Selecting the Pre-Charge Delay (TRP)

The $t_{RP}$ value (pre-charge delay) defines the required delay, in number of clock cycles, between the time the SDC issues a pre-charge command and the time it issues a bank activate command. The $t_{RP}$ also specifies the time required between pre-charge and auto-refresh, and between pre-charge and self-refresh. The $t_{RP}$ and $t_{RAS}$ values define the $t_{RFC}$, $t_{RC}$, and $t_{XSR}$ values.
This parameter enables the application to accommodate the SDRAM timing requirements.

The TRP bits in the SDRAM memory global control register (EBIU_SDGCTL) select the tRP value. Any value between 1 and 7 clock cycles may be selected. For example:

- **TRP = 000**
  
  No effect
- **TRP = 001**
  
  1 clock cycle
- **TRP = 010**
  
  2 clock cycles
- **TRP = 111**
  
  7 clock cycles

**Selecting the Write to Pre-Charge Delay (TWR)**

The tWR value defines the required delay, in number of clock cycles, between the time the SDC issues a write command (drives write data) and a pre-charge command.

This parameter enables the application to accommodate the SDRAM timing requirements.
The **TWR** bits in the SDRAM memory global control register (**EBIU_SDGCTL**) select the **tWR** value. Any value between 1 and 3 clock cycles may be selected. For example:

- **TWR = 00**
  
  Reserved
- **TWR = 01**
  
  1 clock cycle
- **TWR = 10**
  
  2 clock cycles
- **TWR = 11**
  
  3 clock cycles

**EBIU_SDBCTL Register**

The SDRAM memory bank control register (**EBIU_SDBCTL**), shown in Figure 18-13, includes external bank-specific programmable parameters. It allows software to control some parameters of the SDRAM. The external bank can be configured for a different size of SDRAM. It uses the access timing parameters defined in the SDRAM memory global control register (**EBIU_SDGCTL**). The **EBIU_SDBCTL** register should be programmed before power-up and should be changed only when the SDC is idle.
External Bus Interface Unit

The EBIU_SDBCTL register stores the configuration information for the SDRAM bank interface. The EBIU supports 64M bit, 128M bit, 256M bit, and 512M bit SDRAM devices with x4, x8, x16 configurations. Table 18-4 maps SDRAM density and I/O width to the supported EBSZ encoding. See “SDRAM External Memory Size” on page 18-50 for more information on bank starting address decodes.

The SDC determines the internal SDRAM page size from the EBCAW parameters. Page sizes of 512B, 1K byte, 2K byte, and 4K byte are supported. Table 18-5 shows the page size and breakdown of the internal address (IA[31:0], as seen from the core or DMA) into the row, bank, column, and byte address. The column address and the byte address together make up the address inside the page.

The EBE bit in the EBIU_SDBCTL register is used to enable or disable the external SDRAM bank. If the SDRAM is disabled, any access to the SDRAM address space generates an internal bus error, and the access does not occur externally. For more information, see “Error Detection” on page 18-8.

Figure 18-13. SDRAM Memory Bank Control Register

The EBIU_SDBCTL register stores the configuration information for the SDRAM bank interface. The EBIU supports 64M bit, 128M bit, 256M bit, and 512M bit SDRAM devices with x4, x8, x16 configurations. Table 18-4 maps SDRAM density and I/O width to the supported EBSZ encoding. See “SDRAM External Memory Size” on page 18-50 for more information on bank starting address decodes.

The SDC determines the internal SDRAM page size from the EBCAW parameters. Page sizes of 512B, 1K byte, 2K byte, and 4K byte are supported. Table 18-5 shows the page size and breakdown of the internal address (IA[31:0], as seen from the core or DMA) into the row, bank, column, and byte address. The column address and the byte address together make up the address inside the page.

The EBE bit in the EBIU_SDBCTL register is used to enable or disable the external SDRAM bank. If the SDRAM is disabled, any access to the SDRAM address space generates an internal bus error, and the access does not occur externally. For more information, see “Error Detection” on page 18-8.
Table 18-5. Internal Address Mapping

<table>
<thead>
<tr>
<th>Bank Size (M byte)</th>
<th>Bank Address</th>
<th>Page Size (K byte)</th>
<th>Row Address</th>
<th>Column Address</th>
<th>Byte Address</th>
</tr>
</thead>
</table>

For information on how to connect to SDRAMs smaller than 16M byte, see “Using SDRAMs Smaller Than 16M Byte” on page 21-8.
EBIU_SDSTAT Register

The SDRAM control status register (EBIU_SDSTAT), shown in Figure 18-14, provides information on the state of the SDC. This information can be used to determine when it is safe to alter SDC control parameters or it can be used as a debug aid. The SDEASE bit of this register is sticky. Once it has been set, software must explicitly write a 1 to the bit to clear it. Writes have no effect on the other status bits, which are updated by the SDC only. This SDC MMR is 16 bits wide.

SDRAM Control Status Register (EBIU_SDSTAT)

<table>
<thead>
<tr>
<th>Bit 15</th>
<th>Bit 14</th>
<th>Bit 13</th>
<th>Bit 12</th>
<th>Bit 11</th>
<th>Bit 10</th>
<th>Bit 9</th>
<th>Bit 8</th>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- **BGSTAT**: Bus grant status
  - 0: Bus not granted
  - 1: Bus granted

- **SDEASE**: SDRAM EAB sticky error status. Write 1 to this bit to clear it.
  - 0: No error detected
  - 1: EAB access generated an error

- **SDRS**: Will not power up on next SDRAM access (SDRAM already powered up)
  - 0: SDRAM controller idle
  - 1: SDRAM self-refresh active
  - 0: SDRAMs not in self-refresh mode
  - 1: SDRAMs in self-refresh mode

- **SDPUA**: SDRAM power-up active
  - 0: SDC not in power-up sequence
  - 1: SDC in power-up sequence

Figure 18-14. SDRAM Control Status Register

EBIU_SDRRC Register

The SDRAM refresh rate control register (EBIU_SDRRC), shown in Figure 18-15, provides a flexible mechanism for specifying the auto-refresh timing. Since the clock supplied to the SDRAM can vary, the SDC provides a programmable refresh counter, which has a period based
on the value programmed into the RDIV field of this register. This counter coordinates the supplied clock rate with the SDRAM device’s required refresh rate.

The desired delay (in number of SDRAM clock cycles) between consecutive refresh counter time-outs must be written to the RDIV field. A refresh counter time-out triggers an auto-refresh command to all external SDRAM devices. Write the RDIV value to the EBIU_SDRRC register before the SDRAM power-up sequence is triggered. Change this value only when the SDC is idle.

To calculate the value that should be written to the EBIU_SDRRC register, use the following equation:

\[
RDIV = \left( \frac{f_{SCLK} \times t_{REF}}{NRA} \right) - (t_{RAS} + t_{RP})
\]

Where:

- \( f_{SCLK} \) = SDRAM clock frequency (system clock frequency)
- \( t_{REF} \) = SDRAM refresh period
- \( NRA \) = number of row addresses in SDRAM (refresh cycles to refresh whole SDRAM)
- \( t_{RAS} \) = active to pre-charge time (\( TRAS \) in the SDRAM memory global control register) in number of clock cycles
- \( t_{RP} \) = RAS to pre-charge time (\( TRP \) in the SDRAM memory global control register) in number of clock cycles
This equation calculates the number of clock cycles between required refreshes and subtracts the required delay between bank activate commands to the same internal bank \((t_{RC} = t_{RAS} + t_{RP})\). The \(t_{RC}\) value is subtracted, so that in the case where a refresh time-out occurs while an SDRAM cycle is active, the SDRAM refresh rate specification is guaranteed to be met. The result from the equation should always be rounded down to an integer.

Below is an example of the calculation of \(RDIV\) for a typical SDRAM in a system with a 133 MHz clock:

- \(f_{SCLK} = 133\) MHz
- \(t_{REF} = 64\) ms
- \(NRA = 4096\) row addresses
- \(t_{RAS} = 2\)
- \(t_{RP} = 2\)

The equation for \(RDIV\) yields:

\[
RDIV = ( (133 \times 106 \times 64 \times 10^{-3})/4096) - (2 + 2) = 2074\) clock cycles

This means \(RDIV\) is 0x81A (hex) and the SDRAM refresh rate control register should be written with 0x081A.

Note that \(RDIV\) must be programmed to a nonzero value if the SDRAM controller is enabled. When \(RDIV = 0\), operation of the SDRAM controller is not supported and can produce undesirable behavior. Values for \(RDIV\) can range from 0x001 to 0xFFF.

Refer to “Managing SDRAM Refresh During PLL Transitions” on page 21-8 for a detailed discussion of the process for changing the PLL frequency when using SDRAM.
SDRAM External Memory Size

The total amount of external SDRAM memory addressed by the processor is controlled by the EBSZ bits of the EBIU_SDBCTL register (see Table 18-6). Accesses above the range shown for a specialized EBSZ value results in an internal bus error and the access does not occur. For more information, see “Error Detection” on page 18-8.

Table 18-6. External Bank Size Encodings

<table>
<thead>
<tr>
<th>EBSZ</th>
<th>Bank Size (M byte)</th>
<th>Valid SDRAM Addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>b#00</td>
<td>16</td>
<td>0x0000 0000 – 0x00FF FFFF</td>
</tr>
<tr>
<td>b#01</td>
<td>32</td>
<td>0x0000 0000 – 0x01FF FFFF</td>
</tr>
<tr>
<td>b#10</td>
<td>64</td>
<td>0x0000 0000 – 0x03FF FFFF</td>
</tr>
<tr>
<td>b#11</td>
<td>128</td>
<td>0x0000 0000 – 0x07FF FFFF</td>
</tr>
</tbody>
</table>

SDRAM Address Mapping

To access SDRAM, the SDC multiplexes the internal 32-bit non-multiplexed address into a row address, a column address, a bank address, and the byte data masks for the SDRAM device. See Figure 18-16. The lowest bit is mapped to byte data masks, the next bits are mapped into the column address, the next bits are mapped into the row address, and the final two bits are mapped into the bank address. This mapping is based on the EBSZ and EBCAW parameters programmed into the SDRAM memory bank control register.

Figure 18-16. Multiplexed SDRAM Addressing Scheme
16-Bit Wide SDRAM Address Muxing

Table 18-7 shows the connection of the address pins with the SDRAM device pins.

Table 18-7. SDRAM Address Connections for 16-Bit Banks

<table>
<thead>
<tr>
<th>External Address Pin</th>
<th>SDRAM Address Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDR[18]</td>
<td>BA[0]</td>
</tr>
<tr>
<td>ADDR[1]</td>
<td>A[0]</td>
</tr>
</tbody>
</table>
Data Mask (SDQM[1:0]) Encoding

During write transfers to SDRAM, the SDQM[1:0] pins are used to mask writes to bytes that are not accessed. Table 18-8 shows the SDQM[1:0] encoding for 16-bit wide SDRAM based on the internal transfer address bit IA[0] and the transfer size.

During read transfers to SDRAM banks, reads are always done of all bytes in the bank regardless of the transfer size. This means for 16-bit SDRAM banks, SDQM[1:0] are all 0s.

The only time that the SDQM[1:0] pins are high is when bytes are masked during write transfers to the SDRAM. At all other times, the SDQM[1:0] pins are held low.

Table 18-8. SDQM[1:0] Encoding During Writes

<table>
<thead>
<tr>
<th>Internal Address IA[0]</th>
<th>Internal Transfer Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>byte</td>
</tr>
<tr>
<td></td>
<td>SDQM[0] = 0</td>
</tr>
<tr>
<td>1</td>
<td>SDQM[1] = 0</td>
</tr>
</tbody>
</table>

SDC Operation

The SDC uses a burst length = 1 for read and write operations. Whenever a page miss occurs, the SDC executes a pre-charge command followed by a bank activate command before executing the read or write command. If there is a page hit, the read or write command can be given immediately without requiring the pre-charge command.
For SDRAM read commands, there is a latency from the start of the read command to the availability of data from the SDRAM, equal to the CAS latency. This latency is always present for any single read transfer. Subsequent reads do not have latency.

A programmable refresh counter is provided. It can be programmed to generate background auto-refresh cycles at the required refresh rate based on the clock frequency used. The refresh counter period is specified with the \texttt{RDIV} field in the SDRAM refresh rate control register.

To allow auto-refresh commands to execute in parallel with any AMC access, a separate \texttt{A10} pin (\texttt{SA10}) is provided. All the SDRAM internal banks are pre-charged before issuing an auto-refresh command.

The internal 32-bit non-multiplexed address is multiplexed into a row address, a column address, a bank select address, and data masks. Bit0 for 16-bit wide SDRAMs is used to generate the data masks. The next lowest bits are mapped into the column address, next bits are mapped into the row address, and the final two bits are mapped into the internal bank address. This mapping is based on the \texttt{EBCAW} and \texttt{EBSZ} values programmed into the SDRAM memory bank control register.

**SDC Configuration**

After a processor’s hardware or software reset, the SDC clocks are enabled; however, the SDC must be configured and initialized. Before programming the SDC and executing the power-up sequence, ensure the clock to the SDRAM is enabled after the power has stabilized for the proper amount of time (as specified by the SDRAM). In order to set up the SDC and start the SDRAM power-up sequence for the SDRAMs, the SDRAM refresh rate control register (\texttt{EBIU_SDRRC}), the SDRAM memory bank control register (\texttt{EBIU_SDBCTL}), and SDRAM memory global control register (\texttt{EBIU_SDGCTL}) must be written, and a transfer must be started to SDRAM.
address space. The SDRS bit of the SDRAM control status register can be checked to determine the current state of the SDC. If this bit is set, the SDRAM power-up sequence has not been initiated.

The RDIV field of the EBIU_SDRRC register should be written to set the SDRAM refresh rate.

The EBIU_SDBCTL register should be written to describe the sizes and SDRAM memory configuration used (EBSZ and EBCAW) and to enable the external bank (EBE). Note until the SDRAM power-up sequence has been started, any access to SDRAM address space, regardless of the state of the EBE bit, generates an internal bus error, and the access does not occur externally. For more information, see “Error Detection” on page 18-8. After the SDRAM power-up sequence has completed, if the external bank is disabled, any transfer to it results in a hardware error interrupt, and the SDRAM transfer does not occur.

The EBIU_SDGCTL register is written to:

- set the SDRAM cycle timing options (CL, TRAS, TRP, TRCD, TWR, EBUFE)
- enable the SDRAM clock (SCTLE)
- select and enable the start of the SDRAM power-up sequence (PSM, PSSE)

Note if SCTLE is disabled, any access to SDRAM address space generates an internal bus error and the access does not occur externally. For more information, see “Error Detection” on page 18-8.

Once the PSSE bit in the EBIU_SDGCTL register is set to 1, and a transfer occurs to enabled SDRAM address space, the SDC initiates the SDRAM power-up sequence. The exact sequence is determined by the PSM bit in the EBIU_SDGCTL register. The transfer used to trigger the SDRAM power-up sequence can be either a read or a write. This transfer occurs
when the SDRAM power-up sequence has completed. This initial transfer takes many cycles to complete since the SDRAM power-up sequence must take place.

**SDC Commands**

This section provides a description of each of the commands that the SDC uses to manage the SDRAM interface. These commands are initiated automatically upon a memory read or memory write. A summary of the various commands used by the on-chip controller for the SDRAM interface is as follows.

- Pre-charge all: pre-charges all banks
- Single pre-charge: pre-charges a single bank
- Bank activate: activates a page in the required SDRAM internal bank
- Load mode register: initializes the SDRAM operation parameters during the power-up sequence
- Load extended mode register: initializes mobile SDRAM operation parameters during the power-up sequence
- Read/write
- Auto-refresh: causes the SDRAM to execute a CAS before RAS refresh
- Self-refresh: places the SDRAM in self-refresh mode, in which the SDRAM powers down and controls its refresh operations internally
- NOP/command inhibit: no operation
Table 18-9 shows the SDRAM pin state during SDC commands.

Table 18-9. Pin State During SDC Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>SMS</th>
<th>SCAS</th>
<th>SRAS</th>
<th>SWE</th>
<th>SCKE</th>
<th>SA10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-charge All</td>
<td>low</td>
<td>high</td>
<td>low</td>
<td>low</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>Single Pre-charge</td>
<td>low</td>
<td>high</td>
<td>low</td>
<td>low</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td>Bank Activate</td>
<td>low</td>
<td>high</td>
<td>low</td>
<td>high</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td>Load Mode Register</td>
<td>low</td>
<td>low</td>
<td>low</td>
<td>low</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>Load Extended Mode Register</td>
<td>low</td>
<td>low</td>
<td>low</td>
<td>low</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td>Read</td>
<td>low</td>
<td>low</td>
<td>high</td>
<td>high</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td>Write</td>
<td>low</td>
<td>low</td>
<td>high</td>
<td>low</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td>Auto-Refresh</td>
<td>low</td>
<td>low</td>
<td>low</td>
<td>high</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>Self-Refresh</td>
<td>low</td>
<td>low</td>
<td>low</td>
<td>high</td>
<td>low</td>
<td>low</td>
</tr>
<tr>
<td>NOP</td>
<td>low</td>
<td>high</td>
<td>high</td>
<td>high</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>Command Inhibit</td>
<td>high</td>
<td>high</td>
<td>high</td>
<td>high</td>
<td>high</td>
<td>high</td>
</tr>
</tbody>
</table>

**Pre-Charge Commands**

The pre-charge all command is given to pre-charge all internal banks at the same time before executing an auto-refresh. For a page miss during reads or writes in a specific internal SDRAM bank, the SDC uses the single pre-charge command to that bank.
Bank Activate Command

The bank activate command is required if the next data access is in a different page. The SDC executes the pre-charge command, followed by a bank activate command, to activate the page in the desired SDRAM internal bank.

- The SDC supports bank interleaving (opening up to 4 internal SDRAM banks at a time). This results in an effective size of 4 pages. The address mapping indicates the start address of each internal bank.

- Bank interleaving is accomplished by switching between 4 internal SDRAM banks without any stalls between the pages.

Load Mode Register Command

The load mode register command initializes SDRAM operation parameters. This command is a part of the SDRAM power-up sequence. The load mode register command uses the address bus of the SDRAM as data input. The power-up sequence is initiated by writing 1 to the PSSE bit in the SDRAM memory global control register (EBIU_SDGCTL) and then writing or reading from any enabled address within the SDRAM address space to trigger the power-up sequence. The exact order of the power-up sequence is determined by the PSM bit of the EBIU_SDGCTL register.

The load mode register command initializes these parameters:

- Burst length = 1, bits 2–0, always 0
- Wrap type = sequential, bit 3, always 0
- Ltmode = latency mode (CAS latency), bits 6–4, programmable in the EBIU_SDGCTL register
- Bits 14–7, always 0
SDRAM Controller (SDC)

While executing the load mode register command, the unused address pins are set to 0. During the two clock cycles following the load mode register command, the SDC issues only NOP commands.

For low power mobile SDRAMs that include an extended mode register, this register is programmed during power-up sequence if the EMREN bit is set in the EBIU_SDGCTL register.

The extended mode register is initialized with these parameters:

- Partial array self-refresh, bits 2–0, bit 2 always 0, bits 1–0 programmable in EBIU_SDGCTL
- Temperature compensated self-refresh, bits 4–3, bit 3 always 1, bit 4 programmable in EBIU_SDGCTL
- Bits 12–5, always 0, and bit 13 always 1

Read/Write Command

A read/write command is executed if the next read/write access is in the present active page. During the read command, the SDRAM latches the column address. The delay between activate and read commands is determined by the $t_{RCD}$ parameter. Data is available from the SDRAM after the CAS latency has been met.

In the write command, the SDRAM latches the column address. The write data is also valid in the same cycle. The delay between activate and write commands is determined by the $t_{RCD}$ parameter.

The SDC does not use the auto-pre-charge function of SDRAMs, which is enabled by asserting $SA_{10}$ high during a read or write command.
Auto-Refresh Command

The SDRAM internally increments the refresh address counter and causes a CAS before RAS (CBR) refresh to occur internally for that address when the auto-refresh command is given. The SDC generates an auto-refresh command after the SDC refresh counter times out. The RDIV value in the SDRAM refresh rate control register must be set so that all addresses are refreshed within the tREF period specified in the SDRAM timing specifications. This command is issued to the external bank whether or not it is enabled (EBE in the SDRAM memory global control register). Before executing the auto-refresh command, the SDC executes a pre-charge all command to the external bank. The next activate command is not given until the tRFC specification (tRFC = tRAS + tRP) is met.

Auto-refresh commands are also issued by the SDC as part of the power-up sequence and also after exiting self-refresh mode.

Self-Refresh Command

The self-refresh command causes refresh operations to be performed internally by the SDRAM, without any external control. This means that the SDC does not generate any auto-refresh cycles while the SDRAM is in self-refresh mode. Before executing the self-refresh command, all internal banks are pre-charged. Self-refresh mode is enabled by writing a 1 to the SRFS bit of the SDRAM memory global control register (EBIU_SDGCTL). After issuing the self-refresh command, the SDC drives SCKE low. This puts the SDRAM into a power down mode (SCKE = 0, SRAS/SMS/SCAS/SWE = T) Before exiting self-refresh mode, the SDC asserts SCKE. The SDRAM remains in self-refresh mode for at least tRAS and until an internal access to SDRAM space occurs. When an internal access occurs causing the SDC to exit the SDRAM from self-refresh mode, the SDC waits to meet the tXSR specification (tXSR = tRAS + tRP) and then issues an auto-refresh command. After the auto-refresh command, the SDC waits for the tRFC specification (tRFC = tRAS + tRP) to be met before executing the activate command for the transfer that caused the SDRAM
to exit self-refresh mode. Therefore, the latency from when a transfer is received by the SDC while in self-refresh mode, until the activate command occurs for that transfer, is \(2 \times (t_{\text{RAS}} + t_{\text{RP}})\).

Note \(\text{CLKOUT}\) is not disabled by the SDC during self-refresh mode. However, software may disable the clock by clearing the \(\text{SCTLE}\) bit in the \(\text{EBIU_SDGCTL}\) register. The application software should ensure that all applicable clock timing specifications are met before the transfer to SDRAM address space which causes the controller to exit self-refresh mode. If a transfer occurs to SDRAM address space when the \(\text{SCTLE}\) bit is cleared, an internal bus error is generated, and the access does not occur externally, leaving the SDRAM in self-refresh mode. For more information, see “Error Detection” on page 18-8.

**No Operation/Command Inhibit Commands**

The no operation (NOP) command to the SDRAM has no effect on operations currently in progress. The command inhibit command is the same as a NOP command; however, the SDRAM is not chip-selected. When the SDC is actively accessing the SDRAM but needs to insert additional commands with no effect, the NOP command is given. When the SDC is not accessing the SDRAM, the command inhibit command is given.

**SDRAM Timing Specifications**

To support key timing requirements and power-up sequences for different SDRAM vendors, the SDC provides programmability for \(t_{\text{RAS}}, t_{\text{RP}}, t_{\text{RCD}}, t_{\text{WR}}\), and the power-up sequence mode. (For more information, see “EBIU_SDGCTL Register” on page 18-33.) CAS latency should be programmed in the \(\text{EBIU_SDGCTL}\) register based on the frequency of operation. (Refer to the SDRAM vendor’s data sheet for more information.)
For other parameters, the SDC assumes:

- Bank cycle time: $t_{RC} = t_{RAS} + t_{RP}$
- Refresh cycle time: $t_{RFC} = t_{RAS} + t_{RP}$
- Exit self-refresh time: $t_{XSR} = t_{RAS} + t_{RP}$
- Load mode register to activate time: $t_{MRD}$ or $t_{RSC} = 3$ clock cycles
- Page-miss penalty = $t_{RP} + t_{RCD}$
- Row (bank A) to row (bank B) active time: $t_{RRD} = t_{RCD} + 1$

**SDRAM Performance**

Table 7-3 lists the data throughput rates for the core or DMA read/write accesses to 16-bit wide SDRAM. For this example, assume all cycles are $SCLK$ cycles and the following $SCLK$ frequency and SDRAM parameters are used:

- $SCLK$ frequency = 133 MHz
- CAS latency = 2 cycles ($CL = 2$)
- No SDRAM buffering ($EBUFE = 0$)
- RAS pre-charge ($t_{RP}$) = 2 cycles ($TRP = 2$)
- RAS to CAS delay ($t_{RCD}$) = 2 cycles ($TRCD = 2$)
- Active command time ($t_{RAS}$) = 5 cycles ($TRAS = 5$)

When the external buffer timing ($EBUFE = 1$ in the SDRAM memory global control register) and/or CAS latency of 3 ($CL = 11$ in the SDRAM memory global control register) is used, all accesses take one extra cycle for each feature selected.
Bus Request and Grant

The processor can relinquish control of the data and address buses to an external device. The processor three-states its memory interface to allow an external controller to access either external asynchronous or synchronous memory parts.

Operation

When the external device requires access to the bus, it asserts the bus request (BR) signal. The BR signal is arbitrated with EAB requests. If no internal request is pending, the external bus request will be granted. The processor initiates a bus grant by:

- Three-stating the data and address buses and the asynchronous memory control signals. The synchronous memory control signals can optionally be three-stated.

- Asserting the bus grant (BG) signal.

The processor may halt program execution if the bus is granted to an external device and an instruction fetch or data read/write request is made to external memory. When the external device releases BR, the processor deasserts BG and continues execution from the point at which it stopped.

The processor asserts the BGH pin when it is ready to start another external port access, but is held off because the bus was previously granted.

When the bus has been granted, the BGSTAT bit in the SDSTAT register is set. This bit can be used by the processor to check the bus status to avoid initiating a transaction that would be delayed by the external bus grant.
19 CONTROLLER AREA NETWORK (CAN) MODULE

This chapter describes the controller area network (CAN) module which is available on the ADSP-BF538/BF538F4/BF538F8 processors. Familiarity with the CAN standard is assumed. Refer to Version 2.0 of CAN Specification from Robert Bosch GmbH. The CAN pins, \texttt{CANTX} and \texttt{CANRX}, can be freed as GPIO if the CAN interface is not used. Refer to Chapter 15, “General-Purpose Input/Output Ports C, D, E” for details.

Overview

Key features of the CAN module are:

- Conforms to the CAN 2.0B (active) standard.
- Supports both standard (11-bit) and extended (29-bit) identifiers
- Supports data rates of up to 1M bit/s
- 32 mailboxes (8 transmit, 8 receive, 16 configurable)
- Dedicated acceptance mask for each mailbox
- Data filtering (first 2 bytes) can be used for acceptance filtering (DeviceNet™ mode)
- Error status and warning registers
- Universal counter module
- Readable receive and transmit pin values
The CAN module is a low bit rate serial interface intended for use in applications where bit rates are typically up to 1M bit/s. The CAN protocol incorporates a data CRC check, message error tracking and fault node confinement as means to improve network reliability to the level required for control applications.

The interface to the CAN bus is a simple two-wire line. See Figure 19-1 for a symbolic representation of the CAN transceiver interconnection. The Blackfin processor’s `CANTX` output and `CANRX` input pins are connected to an external CAN transceiver’s TX and RX pins (respectively). The `CANTX` and `CANRX` pins operate with TTL levels and are appropriate for operation with CAN bus transceivers according to ISO/DIS 11898 or with a modified RS-485 interface. See Table 19-1.

![Figure 19-1. Representation of CAN Transceiver Interconnection](image)

The Blackfin processor’s CAN module can interface to either 3V or 5V CAN transceivers. The `CANRX` pin is 5V tolerant. See the product data sheet for information on using 5V transceivers.

The default state of the `CANTX` output is recessive (logic 1). Note that `CANTX` is multiplexed with a SPORT transmit signal pin. The output state may be low if the SPORT is selected instead of the CAN, as is the default case after reset. The input value on the `CANRX` pin is ignored.

The CAN module architecture is based around a 32-entry mailbox RAM. The mailbox is accessed sequentially by the CAN serial interface or the host CPU. Each mailbox consists of eight 16-bit registers. The data is
Controller Area Network (CAN) Module

divided into fields, which includes a message identifier, a time stamp, a byte count, up to 8 bytes of data, and several control bits. Each node monitors the messages being passed on the network. If the identifier in the transmitted message matches an identifier in one of its mailboxes, then the module knows that the message was meant for it, passes the data into its appropriate mailbox, and signals the host of its arrival with an interrupt.

Table 19-1. Input and Output Values for RX and TX

<table>
<thead>
<tr>
<th>Pin</th>
<th>Value at Pin</th>
<th>Value on CAN Bus Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>RX</td>
<td>Low (GND)</td>
<td>Dominant</td>
</tr>
<tr>
<td></td>
<td>High (VCC)</td>
<td>Recessive</td>
</tr>
<tr>
<td>TX</td>
<td>Low (GND)</td>
<td>Dominant</td>
</tr>
<tr>
<td></td>
<td>High (VCC)</td>
<td>Recessive</td>
</tr>
</tbody>
</table>

The CAN network itself is a single, differential pair line. All nodes continuously monitor this line. The asynchronous interface does not require a separate clock wire. Messages are passed in one of 4 standard message types or frames. Synchronization is achieved by an elaborate sync scheme performed in each CAN receiver. Message arbitration is accomplished 1 bit at a time. A dominant polarity is established for the network. All nodes are allowed to start transmitting at the same time following a frame sync pulse.

As each node transmits a bit, it checks to see if the bus is the same state that it transmitted. If it is, it continues to transmit. If not, then another node has transmitted a dominant bit so the first node knows it has lost the arbitration and it stops transmitting. The arbitration continues, bit by bit until only 1 node is left transmitting.

The electrical characteristics of each network connection are very stringent so the CAN interface is typically divided into 2 parts: a controller and a transceiver. This allows a single controller to support different drivers and
Overview

CAN networks. The CAN module represents only the controller part of the interface. This module's network I/O is a single transmit line and a single receive line, which communicate to a line transceiver.

The CAN protocol, standards and recommendations are not repeated in this chapter. This chapter covers only those sections which are of immediate need to understand the implementation.

Low Power Features

The Blackfin processor provides a low power hibernate state, and the CAN module includes a built-in sleep mode. The behavior of the CAN module in these two modes is described in the following sections.

CAN Wake-Up From Hibernate State

The Blackfin processor provides a hibernate state, where the internal voltage regulator shuts off the internal power supply to the chip, turning off the core and system clocks in the process. In this mode, the only power drawn (roughly 50 μA) is that used by the regulator circuitry awaiting any of the possible hibernate wake-up events. One such event is a wake-up due to CAN bus activity. After hibernation, the CAN module must be re-initialized.

For low power designs, the external CAN bus transceiver is typically put into standby mode via one of the Blackfin processor’s general-purpose I/O pins. While in standby mode, the CAN transceiver continually drives the recessive logic 1 level onto the CANRX pin. If the transceiver then senses CAN bus activity it drives the CANRX pin to the dominant logic 0 level. This signals to the Blackfin processor that CAN bus activity has been detected. If the internal voltage regulator is programmed to recognize CAN bus activity as an event to exit hibernate state, the part responds appropriately. Otherwise, the activity on the CANRX pin has no effect on the processor state.
Controller Area Network (CAN) Module

To enable this functionality, the voltage control register (VR_CTL) must be programmed with the CAN wake-up enable bit set. The typical sequence of events to use the CAN wake-up feature is:

1. Use a general-purpose I/O pin to put the external transceiver into standby mode.

2. Program VR_CTL with the CAN wake-up enable CANWE bit set and the FREQ field set to 00.

CAN Built-In Sleep Mode

The CAN module has a built-in sleep mode. This mode is entered by setting the SMR bit in the CAN_CONTROL register. Once this mode is entered, many of the internal CAN module clocks are shut off, reducing power consumption. When the CAN module is in sleep mode, all register reads return the contents of CAN_INTR instead of the usual contents. All register writes, except to CAN_INTR, are ignored in sleep mode.

A small part of the module is clocked continuously to allow for wake-up out of sleep mode. A write to the CAN_INTR register ends sleep mode. If the WBA bit in the CAN_CONTROL register is set before entering sleep mode, a dominant bit on the CANRX pin ends sleep mode.
CAN Module Control and Configuration Registers

Some global command bits are implemented in the master control register (CAN_CONTROL). The global status register (CAN_STATUS) represents a set of internal status signals. The CAN bit timing parameter and the special modes of the CAN module are defined in the CAN_CLOCK and CAN_TIMING registers.

All bit timing values can only be changed when the CAN core module is in its configuration mode. The software reset does not change the values of CAN_CLOCK and CAN_TIMING. Thus, an ongoing transfer via the CAN bus cannot be corrupted by changing the bit timing parameter or initiating the software reset (SRS = 1 in CAN_CONTROL). The registers CAN_CLOCK and CAN_TIMING are locked if CCA = 0 in the CAN_STATUS register.

Additionally, the CAN module contains test mode features that aid in the debugging of the CAN software and system, available via the CAN debug register (CAN_DEBUG).

CAN Control (CAN_CONTROL) Register

Some global command bits are implemented in the master control register (CAN_CONTROL), shown in Figure 19-2. After a power-up reset or software reset, the CCR bit is set and all other bits are cleared. During write access, all reserved bits must be 0.
Controller Area Network (CAN) Module

Master Control Register (CAN_CONTROL)

0xFFC0 2AA0

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Reset = 0x0080

- **SRS (Software Reset)**
  - 0 - No effect
  - 1 - Reset

- **DNM (DeviceNet Mode)**
  - 0 - Disable
  - 1 - Enable

- **ABO (Auto Bus On)**
  - 0 - Configuration mode
  - 1 - Enable

- **WBA (Wake Up on CAN Bus Activity)**
  - 0 - Stays in Sleep mode
  - 1 - CAN leave Sleep mode

- **CCR (CAN Configuration Mode Request)**
  - 0 - Cancelled
  - 1 - Requested

- **CSR (CAN Suspend Mode Request)**
  - 0 - Cancelled
  - 1 - Requested

- **SMR (Sleep Mode Request)**
  - 0 - Not requested
  - 1 - Enters Sleep mode

Figure 19-2. Master Control Register

Additional information for the CAN_CONTROL register bits includes:

- **CAN Configuration Mode Request (CCR).**
  
  If the **TSEG1** field of the CAN_TIMING register is programmed to ‘0,’ the module does not leave configuration mode.

  If the CAN module’s transmit error count is greater than or equal to 256, the module enters the “bus-off” state. For more information, see “CAN Error Counter (CAN_CEC) Register” on page 19-85. In this state, the unit is not allowed to have any influence on the bus, that is, its output drivers are switched off.

  During the bus-off recovery sequence, the configuration mode request bit is set by the internal logic (CCR = 1), thus the CAN core module does not automatically go bus on. The CCR bit cannot be reset until the bus-off recovery sequence is finished.
The \texttt{CCR} bit works in an interlock with the configuration mode acknowledge bit (\texttt{CCA} in the \texttt{CAN\_STATUS} register). The \texttt{CCR} bit cannot be cleared if it is set and \texttt{CCA} is clear, and the \texttt{CCR} bit cannot be set if it is clear and \texttt{CCA} is set.

\[1\] The configuration mode is requested. After power-up, this mode is active (\texttt{CCR} = 1 and \texttt{CCA} = 1). The bit timing parameters must be defined before this mode is exited. The write access to the bit timing parameter is locked in normal operating mode (\texttt{CCA} is inactive low). If the CAN core module is currently processing a message on the CAN bus line, this operation is finished before the configuration mode is acknowledged (\texttt{CCA} is active high). Thus, the user must wait until \texttt{CCA} is set before the access to the bit timing parameters (\texttt{CAN\_CLOCK} and \texttt{CAN\_TIMING}) is allowed. During configuration mode, the module is not active on the CAN bus line. The \texttt{CANTX} output pin remains recessive and the module does not receive/transmit messages or error frames. After leaving the configuration mode, all CAN core internal registers and the CAN error counters are set to their initial values.

\[0\] The configuration mode request is cancelled.

- \textbf{CAN Suspend Mode Request (CSR)}.

The \texttt{CSR} bit works in an interlock with the suspend mode acknowledge bit (\texttt{CSA} in the \texttt{CAN\_STATUS} register). The \texttt{CSR} bit cannot be cleared if it is set and \texttt{CSA} is clear, and the \texttt{CSR} bit cannot be set if it is clear and \texttt{CSA} is set.
Controller Area Network (CAN) Module

1—The suspend mode is requested. If the CAN core module is currently processing a message on the CAN bus line, this operation is finished before the suspend mode is acknowledged (CSA is active high). Thus, the user must wait until CSA is set. During suspend mode, the module is not active on the CAN bus line. The CANTX output pin remains recessive and the module does not receive/transmit messages or error frames. The content of the CAN error counters remains unchanged.

0—The suspend mode request is cancelled.

• **Sleep Mode Request** (SMR).

A sleep mode, which gates off the clock to most of the CAN registers and results in lower power use, is provided. The module can wake up on CAN bus activity or on writing to the CAN_INTR register. The SMACK bit in the CAN_INTR register indicates when sleep mode is active.

For greatest power savings, use the hibernate state and set the voltage regulator to wake up on CAN bus activity. For more information, see “CAN Wake-Up From Hibernate State” on page 19-4.

1—The module enters the sleep mode after the current operation of the CAN bus is finished.

0—No sleep mode requested.

• **Wake-Up on CAN Bus Activity** (WBA).

1—The sleep mode is left automatically if there is any activity on the CAN bus line detected.

0—The module stays in sleep mode independent of the CAN bus line status until SMR is reset.
• **Auto Bus On** (ABO).

  1—After completing the bus-off recovery procedure, the bus active state is automatically entered.
  
  0—After completing the bus-off recovery procedure, the configuration mode is entered.

• **DeviceNet Mode** (DNM).

  If enabled, the acceptance filtering run starts after reception of the first CRC bit, else after the first received DLC bit.

  1—DeviceNet mode (filtering on data bytes) is enabled. The FDF and FMD bits in the CAN_AMxxH register determine the DeviceNet mode functionality. See “CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40.

  0—Standard acceptance filtering is only used on the identifier.

• **Software Reset** (SRS).

  This register bit is always read as 0.

  1—Initiates a software reset. All relevant register bits are set to their initial values unless otherwise noted in the corresponding register description.

  0—Has no effect.
Controller Area Network (CAN) Module

CAN Status (CAN_STATUS) Register

The global status register (CAN_STATUS), shown in Figure 19-3, represents a set of internal status signals. This register is read only. A write access to the CAN_STATUS register has no effect.

Figure 19-3. Global Status Register

Additional information for the CAN_STATUS register bits includes:

- **Receive Mode** (REC).
  
  1—CAN protocol kernel is in receive mode.
  
  0—CAN protocol kernel is not in receive mode.
CAN Module Control and Configuration Registers

- **Transmit Mode** (TRM).
  
  1—CAN protocol kernel is in transmit mode.
  
  0—CAN protocol kernel is not in transmit mode.

- **Mail Box Pointer** (MBPTR).
  
  Represents the mailbox number of the current transmit message. After a successful transmission, these bits remain unchanged.
  
  11111—The message of mailbox 31 is currently being processed.
  
  …
  
  …
  
  …
  
  00000—The message of mailbox 0 is currently being processed.

- **CAN Configuration Mode Acknowledge** (CCA).
  
  1—CAN protocol kernel is in configuration mode.
  
  0—CAN protocol kernel is not in configuration mode.

- **CAN Suspend Mode Acknowledge** (CSA).
  
  1—CAN protocol kernel is in suspend mode.
  
  0—CAN protocol kernel is not in suspend mode.
Controller Area Network (CAN) Module

- **Sleep Mode Acknowledge (SMA).**
  
  The SMA bit is always read as 0. If the CAN module is in sleep mode, reading the CAN_STATUS register returns the contents of the CAN_INTR register.

  1—Module is in sleep mode. All clocks are switched off.
  0—Module is not in sleep mode.

- **CAN Error Bus-Off Mode (EBO).**
  
  1—Transmit error counter has reached the bus-off limit of 256.
  0—Value of the transmit error counter TXECNT is below 256.

- **CAN Error Passive Mode (EP).**
  
  1—At least one error counter reached the error passive level of 128.
  0—Values of both error counters (RXECNT and TXECNT) are below 128.

- **CAN Receive Warning Flag (WR).**
  
  1—Value of the receive error counter reached the warning limit.
  0—Value of the receive counter RXECNT is below the warning limit.

- **CAN Transmit Warning Flag (WT).**
  
  1—Value of the transmit error counter reached the warning limit.
  0—Value of the transmit counter TXECNT is below the warning limit.
CAN Clock (CAN_CLOCK) Register

The CAN_CLOCK register (Figure 19-4) does not generate the effective CAN bit clock. It derives the time quantum (TQ) from the system clock (SCLK). Multiple time quanta make up a CAN bit as controlled by the CAN_TIMING register.

The time quantum is derived from this formula: \[ TQ = \frac{BRP + 1}{SCLK} \]

Although the BRP field can be set to any value, it is recommended that the value be greater than or equal to 4. Restrictions apply to the bit timing configuration when BRP is less than 4.

All nodes on a CAN bus should use the same nominal bit rate.

Do not modify this register during normal operation. Always enter configuration mode. Writes to this register have no effect if not in configuration mode.

A software reset has no effect on this register (all values are unchanged).
CAN Timing (CAN_TIMING) Register

Based on the time quantum (TQ) generated by the BRP prescaler in the CAN_CLOCK register, CAN_TIMING (see Figure 19-5) controls the nominal bit time and the sample point of the individual bits in a CAN protocol.

Figure 19-5. CAN Timing Register

Figure 19-6 shows the three phases of a CAN bit: the synchronization segment, the segment before the sample point, and the segment after the sample point.

Figure 19-6. Three Phases of a CAN Bit

The synchronization segment is fixed to one TQ. It is required to synchronize the nodes on the bus. All signal edges are expected to occur within this segment.
The \text{TSEG1} and \text{TSEG2} fields control how many TQs the CAN bits consist of, resulting in the CAN bit rate. The nominal bit time is given by this formula: $t_{\text{BIT}} = \text{TQ} \times (1 + (1 + \text{TSEG1}) + (1 + \text{TSEG2}))$

For safe receive operation on given physical networks, the sample point is programmable by the \text{TSEG1} field. The \text{TSEG2} field holds the number of TQs needed to complete the bit time. Often, best sample reliability is achieved with sample points in the high 80\% range of the bit time. Never use sample points lower than 50\%. Thus, \text{TSEG1} should always be greater than or equal to \text{TSEG2}.

The Blackfin CAN module does not distinguish between the propagation segment and the phase segment 1 as defined by the standard. The \text{TSEG1} value is intended to cover both of them. The \text{TSEG2} value represents the phase segment 2.

If the CAN module detects a recessive-to-dominant edge outside the synchronization segment, it can automatically move the sampling point such that the CAN bit is still handled properly. The synchronization jump width (\text{SJW}) field specifies the maximum number of TQs allowed for such a re synchronization attempt. The \text{SJW} value should not exceed \text{TSEG2} or \text{TSEG1}.

\text{SJW} \leq \text{TSEG2} \leq \text{TSEG1}

In addition to this fundamental rule, \text{TSEG2} must also be greater than or equal to the information processing time (IPT). This is the time required by the logic to sample \text{CANRX}. On the Blackfin CAN module, this is 3 \text{SCLK} cycles. Because of this, \text{TSEG2} must not be 0. If the clock prescaler is set to 0, \text{TSEG2} must be greater than or equal to 3. If the prescaler is set to 1, the minimum \text{TSEG2} is 2.

With the \text{SAM} bit cleared, \text{CANRX} is sampled once after \text{TSEG1} expires. If the \text{SAM} bit is set, the input signal is oversampled three times at \text{SCLK} rate. The resulting value is generated by a majority decision of the three sample values. Always keep the \text{SAM} bit cleared if the \text{BRP} value is less than 4.
Do not modify this register during normal operation. Always enter configuration mode. Writes to this register have no effect if not in configuration mode.

A software reset has no effect to this register (all values are unchanged).

**CAN Debug (CAN_DEBUG) Register**

The CAN module contains test mode features that aid in the debugging of the CAN software and system. **Listing 19-1** provides an example of enabling CAN debug features.

When these features are used, the CAN module may not be compliant to the CAN specification. All test modes should be enabled or disabled only when the module is in configuration mode (CCA = 1 in the **CAN_STATUS** register) or in suspend mode (CSA = 1 in **CAN_STATUS**).

The CDE bit is used to gain access to all of the debug features. This bit must be set to enable the test mode, and must be written first before subsequent writes to the **CAN_DEBUG** register. When the CDE bit is cleared, all debug features are disabled.

**Listing 19-1. Enabling CAN Debug Features in C**

```c
#include <cdefBF538.h>
/* Enable debug mode, CDE must be set before other flags can be changed in register */
*pCAN_DEBUG |= CDE ;

/* Set debug flags */
*pCAN_DEBUG &= ~DTO ;
*pCAN_DEBUG |= MRB | MAA | DIL ;
```


/* Run test code */

/* Disable debug mode */
*pCAN_DEBUG &= ~CDE ;

When the CDE bit is set, it enables writes to the other bits of the CAN_DEBUG register. It also enables these features, which are not compliant with the CAN standard:

- Bit timing registers can be changed anytime, not only during configuration mode. This includes the CAN_CLOCK and CAN_TIMING registers.

- Allows write access to the read-only transmit/receive error counter register CAN_CEC.

The mode read back bit (MRB) is used to enable the read back mode. In this mode, a message transmitted on the CAN bus (or via an internal loop back mode) is received back directly to the internal receive buffer. After a correct transmission, the internal logic treats this as a normal receive message. This feature allows the user to test most of the CAN features without an external device.

The mode auto acknowledge bit (MAA) allows the CAN module to generate its own acknowledge during the ACK slot of the CAN frame. No external devices or connections are necessary to read back a transmit message. In this mode, the message that is sent is automatically stored in the internal receive buffer. In auto acknowledge mode, the module itself transmits the acknowledge. This acknowledge can be programmed to appear on the CANTX pin if DIL=1 and DTO=0. If the acknowledge is only going to be used internally, then these test mode bits should be set to DIL=0 and DTO=1.

The disable internal loop bit (DIL) is used to internally enable the transmit output to be routed back to the receive input.

The disable transmit output bit (DTO) is used to disable the CANTX output pin. When this bit is set, the CANTX pin continuously drives recessive bits.
The disable receive input bit (DRI) is used to disable the CANRX input. When set, the internal logic receives recessive bits or receives the internally generated transmit value in the case of the internal loop enabled (DIL=0). In either case, the value on the CANRX input pin is ignored.

The disable error counters bit (DEC) is used to disable the transmit and receive error counters in the CAN_CEC register. When this bit is set, the CAN_CEC holds its current contents and is not allowed to increment or decrement the error counters. This mode does not conform to the CAN specification.

Writes to the error counters should be in debug mode only. Write access during reception may lead to undefined values. The maximum value which can be written into the error counters is 255. Thus, the error counter value of 256 which forces the module into the bus-off state can not be written into the error counters.

Table 19-2 shows several common combinations of test mode bits.

Table 19-2. CAN Test Modes

<table>
<thead>
<tr>
<th>MRB</th>
<th>MAA</th>
<th>DIL</th>
<th>DTO</th>
<th>DRI</th>
<th>CDE</th>
<th>Functional Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>Normal mode, not debug mode.</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>No read back of transmit message.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Normal transmission on CAN bus line. Read back. External acknowledge from external device required.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Normal transmission on CAN bus line. Read back. No external acknowledge required. Transmit message and acknowledge are transmitted on CAN bus line. CANRX input is enabled.</td>
</tr>
</tbody>
</table>
### Table 19-2. CAN Test Modes (Cont’d)

<table>
<thead>
<tr>
<th>MRB</th>
<th>MAA</th>
<th>DIL</th>
<th>DTO</th>
<th>DRI</th>
<th>CDE</th>
<th>Functional Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Normal transmission on CAN bus line. Read back. No external acknowledge required. Transmit message and acknowledge are transmitted on CAN bus line. CANRX input and internal loop are enabled (internal OR of TX and RX).</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Normal transmission on CAN bus line. Read back. No external acknowledge required. Transmit message and acknowledge are transmitted on CAN bus line. CANRX input is ignored. Internal loop is enabled.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>No transmission on CAN bus line. Read back. No external acknowledge required. Neither transmit message nor acknowledge are transmitted on CANTX. CANRX input is ignored. Internal loop is enabled.</td>
</tr>
</tbody>
</table>

---

19-20  ADSP-BF538/ADSP-BF538F Blackfin Processor Hardware Reference
Data Storage

All CAN-related data is stored in a mailbox RAM. There are 8 words of 16 bits for each of the 32 mailboxes.

The values of the identifier (base part and extended part), the identifier extension bit (IDE), the remote transmission request bit (RTR), the data length code (DLC), and the data field of each message can be programmed in the mailbox area (see Figure 19-7). The substitute remote request (SRR, always sent as recessive) bit and the reserved bits $R_0$ and $R_1$ (always sent as dominant) are generated automatically by the internal logic.

Figure 19-7. CAN Message Formats
Mailbox Identifier Word Registers

Each mailbox consists of 8 words and includes:

- The 29 bit identifier (base part plus extended part)
- The acceptance mask enable bit (AME)
- The remote transmission request bit (RTR)
- The identifier extension bit (IDE)
- The data length code (DLC)
- Up to eight bytes for the data field
- Two bytes for the time stamp value (TSV)

The upper 12 bits of word 4 of each mailbox are marked as reserved. These 12 bits should always be set to 0.

If the filtering on data field option is enabled (DNM = 1 in CAN_CONTROL register and FDF = 1 in corresponding acceptance mask), the bits [15:0] of word 6 (ExtId) are reused as acceptance code (DFC) for the data field filtering.
Controller Area Network (CAN) Module

CAN Mailbox Identifier 1 (CAN_MBxx_ID1) Registers

The mailboxes are implemented as RAM, and have no reset value. Each mailbox must be reset manually before it is enabled by setting the corresponding CAN_MCx register bit.

Mailbox Identifier Word 7 (CAN_MBxx_ID1)

For memory-mapped addresses, see Table 19-3.

Reset = 0xXXXX

Table 19-3. Mailbox Identifier Word 7 Register Addresses

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN_MB00_ID1</td>
<td>0xFFC0 2C1C</td>
<td>CAN_MB16_ID1</td>
<td>0xFFC0 2E1C</td>
</tr>
<tr>
<td>CAN_MB01_ID1</td>
<td>0xFFC0 2C3C</td>
<td>CAN_MB17_ID1</td>
<td>0xFFC0 2E3C</td>
</tr>
<tr>
<td>CAN_MB02_ID1</td>
<td>0xFFC0 2C5C</td>
<td>CAN_MB18_ID1</td>
<td>0xFFC0 2E5C</td>
</tr>
<tr>
<td>CAN_MB03_ID1</td>
<td>0xFFC0 2C7C</td>
<td>CAN_MB19_ID1</td>
<td>0xFFC0 2E7C</td>
</tr>
<tr>
<td>CAN_MB04_ID1</td>
<td>0xFFC0 2C9C</td>
<td>CAN_MB20_ID1</td>
<td>0xFFC0 2E9C</td>
</tr>
<tr>
<td>CAN_MB05_ID1</td>
<td>0xFFC0 2CBC</td>
<td>CAN_MB21_ID1</td>
<td>0xFFC0 2EBC</td>
</tr>
<tr>
<td>CAN_MB06_ID1</td>
<td>0xFFC0 2CDC</td>
<td>CAN_MB22_ID1</td>
<td>0xFFC0 2EDC</td>
</tr>
<tr>
<td>CAN_MB07_ID1</td>
<td>0xFFC0 2CFC</td>
<td>CAN_MB23_ID1</td>
<td>0xFFC0 2EFC</td>
</tr>
<tr>
<td>CAN_MB08_ID1</td>
<td>0xFFC0 2D1C</td>
<td>CAN_MB24_ID1</td>
<td>0xFFC0 2F1C</td>
</tr>
<tr>
<td>CAN_MB09_ID1</td>
<td>0xFFC0 2D3C</td>
<td>CAN_MB25_ID1</td>
<td>0xFFC0 2F3C</td>
</tr>
<tr>
<td>CAN_MB10_ID1</td>
<td>0xFFC0 2D5C</td>
<td>CAN_MB26_ID1</td>
<td>0xFFC0 2F5C</td>
</tr>
<tr>
<td>CAN_MB11_ID1</td>
<td>0xFFC0 2D7C</td>
<td>CAN_MB27_ID1</td>
<td>0xFFC0 2F7C</td>
</tr>
</tbody>
</table>
Mailbox Identifier Word Registers

Table 19-3. Mailbox Identifier Word 7 Register Addresses (Cont’d)

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN_MB12_ID1</td>
<td>0xFFC0 2D9C</td>
<td>CAN_MB28_ID1</td>
<td>0xFFC0 2F9C</td>
</tr>
<tr>
<td>CAN_MB13_ID1</td>
<td>0xFFC0 2DBC</td>
<td>CAN_MB29_ID1</td>
<td>0xFFC0 2FBC</td>
</tr>
<tr>
<td>CAN_MB14_ID1</td>
<td>0xFFC0 2DDC</td>
<td>CAN_MB30_ID1</td>
<td>0xFFC0 2FDC</td>
</tr>
<tr>
<td>CAN_MB15_ID1</td>
<td>0xFFC0 2DFC</td>
<td>CAN_MB31_ID1</td>
<td>0xFFC0 2FFC</td>
</tr>
</tbody>
</table>

Additional information for the CAN_MBxx_ID1 register bits includes:

- **Acceptance Mask Enable (AME).**
  - 1—Enables acceptance mask filtering on incoming messages.
  - 0—Disables acceptance mask filtering.

- **Remote Transmission Request (RTR).**
  - 1—Enables message as a remote frame.
  - 0—Enables message as a data frame.

- **Identifier Extension (IDE).**
  - 1—Enables 29-bit identifier.
  - 0—Enables 11-bit identifier.
CAN Mailbox Identifier 0 (CAN_MBxx_ID0)

Registers

The mailboxes are implemented as RAM, and have no reset value. Each mailbox must be reset manually before it is enabled by setting the corresponding CAN_MCx register bit.

Mailbox Identifier Word 6 (CAN_MBxx_ID0)

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN_MB00_ID0</td>
<td>0xFFC0 2C18</td>
<td>CAN_MB16_ID0</td>
<td>0xFFC0 2E18</td>
</tr>
<tr>
<td>CAN_MB01_ID0</td>
<td>0xFFC0 2C38</td>
<td>CAN_MB17_ID0</td>
<td>0xFFC0 2E38</td>
</tr>
<tr>
<td>CAN_MB02_ID0</td>
<td>0xFFC0 2C58</td>
<td>CAN_MB18_ID0</td>
<td>0xFFC0 2E58</td>
</tr>
<tr>
<td>CAN_MB03_ID0</td>
<td>0xFFC0 2C78</td>
<td>CAN_MB19_ID0</td>
<td>0xFFC0 2E78</td>
</tr>
<tr>
<td>CAN_MB04_ID0</td>
<td>0xFFC0 2C98</td>
<td>CAN_MB20_ID0</td>
<td>0xFFC0 2E98</td>
</tr>
<tr>
<td>CAN_MB05_ID0</td>
<td>0xFFC0 2CB8</td>
<td>CAN_MB21_ID0</td>
<td>0xFFC0 2EB8</td>
</tr>
<tr>
<td>CAN_MB06_ID0</td>
<td>0xFFC0 2CD8</td>
<td>CAN_MB22_ID0</td>
<td>0xFFC0 2ED8</td>
</tr>
<tr>
<td>CAN_MB07_ID0</td>
<td>0xFFC0 2CF8</td>
<td>CAN_MB23_ID0</td>
<td>0xFFC0 2EF8</td>
</tr>
<tr>
<td>CAN_MB08_ID0</td>
<td>0xFFC0 2D18</td>
<td>CAN_MB24_ID0</td>
<td>0xFFC0 2F18</td>
</tr>
<tr>
<td>CAN_MB09_ID0</td>
<td>0xFFC0 2D38</td>
<td>CAN_MB25_ID0</td>
<td>0xFFC0 2F38</td>
</tr>
<tr>
<td>CAN_MB10_ID0</td>
<td>0xFFC0 2D58</td>
<td>CAN_MB26_ID0</td>
<td>0xFFC0 2F58</td>
</tr>
<tr>
<td>CAN_MB11_ID0</td>
<td>0xFFC0 2D78</td>
<td>CAN_MB27_ID0</td>
<td>0xFFC0 2F78</td>
</tr>
<tr>
<td>CAN_MB12_ID0</td>
<td>0xFFC0 2D98</td>
<td>CAN_MB28_ID0</td>
<td>0xFFC0 2F98</td>
</tr>
</tbody>
</table>
Mailbox Identifier Word Registers

Table 19-4. Mailbox Identifier Word 6 Register Addresses (Cont’d)

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN_MB13_ID0</td>
<td>0xFFC0 2DB8</td>
<td>CAN_MB29_ID0</td>
<td>0xFFC0 2FB8</td>
</tr>
<tr>
<td>CAN_MB14_ID0</td>
<td>0xFFC0 2DD8</td>
<td>CAN_MB30_ID0</td>
<td>0xFFC0 2FD8</td>
</tr>
<tr>
<td>CAN_MB15_ID0</td>
<td>0xFFC0 2DF8</td>
<td>CAN_MB31_ID0</td>
<td>0xFFC0 2FF8</td>
</tr>
<tr>
<td>CAN_MB16_TIMESTAMP</td>
<td>0xFFC0 2E14</td>
<td>CAN_MB17_TIMESTAMP</td>
<td>0xFFC0 2E34</td>
</tr>
<tr>
<td>CAN_MB18_TIMESTAMP</td>
<td>0xFFC0 2E54</td>
<td>CAN_MB19_TIMESTAMP</td>
<td>0xFFC0 2E74</td>
</tr>
<tr>
<td>CAN_MB20_TIMESTAMP</td>
<td>0xFFC0 2E94</td>
<td>CAN_MB21_TIMESTAMP</td>
<td>0xFFC0 2EB4</td>
</tr>
</tbody>
</table>

CAN Mailbox Time Stamp (CAN_MBxx_TIMESTAMP) Registers

The mailboxes are implemented as RAM, and have no reset value. Each mailbox must be reset manually before it is enabled by setting the corresponding CAN_MCx register bit.

Mailbox Identifier Word 5 (CAN_MBxx_TIMESTAMP)

![Figure 19-10. Mailbox Identifier Word 5](image)

Table 19-5. Mailbox Identifier Word 5 Register Addresses

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN_MB00_TIMESTAMP</td>
<td>0xFFC0 2C14</td>
<td>CAN_MB16_TIMESTAMP</td>
<td>0xFFC0 2E14</td>
</tr>
<tr>
<td>CAN_MB01_TIMESTAMP</td>
<td>0xFFC0 2C34</td>
<td>CAN_MB17_TIMESTAMP</td>
<td>0xFFC0 2E34</td>
</tr>
<tr>
<td>CAN_MB02_TIMESTAMP</td>
<td>0xFFC0 2C54</td>
<td>CAN_MB18_TIMESTAMP</td>
<td>0xFFC0 2E54</td>
</tr>
<tr>
<td>CAN_MB03_TIMESTAMP</td>
<td>0xFFC0 2C74</td>
<td>CAN_MB19_TIMESTAMP</td>
<td>0xFFC0 2E74</td>
</tr>
<tr>
<td>CAN_MB04_TIMESTAMP</td>
<td>0xFFC0 2C94</td>
<td>CAN_MB20_TIMESTAMP</td>
<td>0xFFC0 2E94</td>
</tr>
<tr>
<td>CAN_MB05_TIMESTAMP</td>
<td>0xFFC0 2CB4</td>
<td>CAN_MB21_TIMESTAMP</td>
<td>0xFFC0 2EB4</td>
</tr>
</tbody>
</table>
Controller Area Network (CAN) Module

Table 19-5. Mailbox Identifier Word 5 Register Addresses (Cont’d)

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN_MB06_TIMESTAMP</td>
<td>0xFFC0 2CD4</td>
<td>CAN_MB22_TIMESTAMP</td>
<td>0xFFC0 2ED4</td>
</tr>
<tr>
<td>CAN_MB07_TIMESTAMP</td>
<td>0xFFC0 2CF4</td>
<td>CAN_MB23_TIMESTAMP</td>
<td>0xFFC0 2EF4</td>
</tr>
<tr>
<td>CAN_MB08_TIMESTAMP</td>
<td>0xFFC0 2D14</td>
<td>CAN_MB24_TIMESTAMP</td>
<td>0xFFC0 2F14</td>
</tr>
<tr>
<td>CAN_MB09_TIMESTAMP</td>
<td>0xFFC0 2D34</td>
<td>CAN_MB25_TIMESTAMP</td>
<td>0xFFC0 2F34</td>
</tr>
<tr>
<td>CAN_MB10_TIMESTAMP</td>
<td>0xFFC0 2D54</td>
<td>CAN_MB26_TIMESTAMP</td>
<td>0xFFC0 2F54</td>
</tr>
<tr>
<td>CAN_MB11_TIMESTAMP</td>
<td>0xFFC0 2D74</td>
<td>CAN_MB27_TIMESTAMP</td>
<td>0xFFC0 2F74</td>
</tr>
<tr>
<td>CAN_MB12_TIMESTAMP</td>
<td>0xFFC0 2D94</td>
<td>CAN_MB28_TIMESTAMP</td>
<td>0xFFC0 2F94</td>
</tr>
<tr>
<td>CAN_MB13_TIMESTAMP</td>
<td>0xFFC0 2DB4</td>
<td>CAN_MB29_TIMESTAMP</td>
<td>0xFFC0 2FB4</td>
</tr>
<tr>
<td>CAN_MB14_TIMESTAMP</td>
<td>0xFFC0 2DD4</td>
<td>CAN_MB30_TIMESTAMP</td>
<td>0xFFC0 2FD4</td>
</tr>
<tr>
<td>CAN_MB15_TIMESTAMP</td>
<td>0xFFC0 2DF4</td>
<td>CAN_MB31_TIMESTAMP</td>
<td>0xFFC0 2FF4</td>
</tr>
</tbody>
</table>

CAN Mailbox Length (CAN_MBxx_LENGTH)

The mailboxes are implemented as RAM, and have no reset value. Each mailbox must be reset manually before it is enabled by setting the corresponding CAN_MCx register bit.

Any DLC value greater than 8 is treated the same as a value of 8.

Mailbox Identifier Word 4 (CAN_MBxx_LENGTH)

For memory-mapped addresses, see Table 19-6.

Figure 19-11. Mailbox Identifier Word 4
Mailbox Identifier Word Registers

The CAN peripheral communicates data most significant bit first. Because of this, the CAN_MBSxx_DATAx registers are used as MSB-first. For example, if only one byte is transmitted or received (DLC = 1), then it was stored in the most significant byte of the CAN_MBSxx_DATA3 register. If two bytes are transmitted or received, they are stored in the upper half and the lower half of CAN_MBSxx_DATA3. Refer to Figure 19-12, Figure 19-13, Figure 19-14, and Figure 19-15.

Table 19-6. Mailbox Identifier Word 4 Register Addresses

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN_MB00_LENGTH</td>
<td>0xFFC0 2C10</td>
<td>CAN_MB16_LENGTH</td>
<td>0xFFC0 2E10</td>
</tr>
<tr>
<td>CAN_MB01_LENGTH</td>
<td>0xFFC0 2C30</td>
<td>CAN_MB17_LENGTH</td>
<td>0xFFC0 2E30</td>
</tr>
<tr>
<td>CAN_MB02_LENGTH</td>
<td>0xFFC0 2C50</td>
<td>CAN_MB18_LENGTH</td>
<td>0xFFC0 2E50</td>
</tr>
<tr>
<td>CAN_MB03_LENGTH</td>
<td>0xFFC0 2C70</td>
<td>CAN_MB19_LENGTH</td>
<td>0xFFC0 2E70</td>
</tr>
<tr>
<td>CAN_MB04_LENGTH</td>
<td>0xFFC0 2C90</td>
<td>CAN_MB20_LENGTH</td>
<td>0xFFC0 2E90</td>
</tr>
<tr>
<td>CAN_MB05_LENGTH</td>
<td>0xFFC0 2CB0</td>
<td>CAN_MB21_LENGTH</td>
<td>0xFFC0 2EB0</td>
</tr>
<tr>
<td>CAN_MB06_LENGTH</td>
<td>0xFFC0 2CD0</td>
<td>CAN_MB22_LENGTH</td>
<td>0xFFC0 2ED0</td>
</tr>
<tr>
<td>CAN_MB07_LENGTH</td>
<td>0xFFC0 2CF0</td>
<td>CAN_MB23_LENGTH</td>
<td>0xFFC0 2EF0</td>
</tr>
<tr>
<td>CAN_MB08_LENGTH</td>
<td>0xFFC0 2D10</td>
<td>CAN_MB24_LENGTH</td>
<td>0xFFC0 2F10</td>
</tr>
<tr>
<td>CAN_MB09_LENGTH</td>
<td>0xFFC0 2D30</td>
<td>CAN_MB25_LENGTH</td>
<td>0xFFC0 2F30</td>
</tr>
<tr>
<td>CAN_MB10_LENGTH</td>
<td>0xFFC0 2D50</td>
<td>CAN_MB26_LENGTH</td>
<td>0xFFC0 2F50</td>
</tr>
<tr>
<td>CAN_MB11_LENGTH</td>
<td>0xFFC0 2D70</td>
<td>CAN_MB27_LENGTH</td>
<td>0xFFC0 2F70</td>
</tr>
<tr>
<td>CAN_MB12_LENGTH</td>
<td>0xFFC0 2D90</td>
<td>CAN_MB28_LENGTH</td>
<td>0xFFC0 2F90</td>
</tr>
<tr>
<td>CAN_MB13_LENGTH</td>
<td>0xFFC0 2DB0</td>
<td>CAN_MB29_LENGTH</td>
<td>0xFFC0 2FB0</td>
</tr>
<tr>
<td>CAN_MB14_LENGTH</td>
<td>0xFFC0 2DD0</td>
<td>CAN_MB30_LENGTH</td>
<td>0xFFC0 2FD0</td>
</tr>
<tr>
<td>CAN_MB15_LENGTH</td>
<td>0xFFC0 2DF0</td>
<td>CAN_MB31_LENGTH</td>
<td>0xFFC0 2FF0</td>
</tr>
</tbody>
</table>
The mailboxes are implemented as RAM, and have no reset value. Each mailbox must be reset manually before it is enabled by setting the corresponding CAN_MCx register bit.

**Mailbox Identifier Word 3 (CAN_MBxx_DATA3)**

For memory-mapped addresses, see Table 19-7.

![](image)

**Figure 19-12. Mailbox Identifier Word 3**

**Table 19-7. Mailbox Identifier Word 3 Register Addresses**

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN_MB00_DATA3</td>
<td>0xFFC0 2C0C</td>
<td>CAN_MB16_DATA3</td>
<td>0xFFC0 2E0C</td>
</tr>
<tr>
<td>CAN_MB01_DATA3</td>
<td>0xFFC0 2C2C</td>
<td>CAN_MB17_DATA3</td>
<td>0xFFC0 2E2C</td>
</tr>
<tr>
<td>CAN_MB02_DATA3</td>
<td>0xFFC0 2C4C</td>
<td>CAN_MB18_DATA3</td>
<td>0xFFC0 2E4C</td>
</tr>
<tr>
<td>CAN_MB03_DATA3</td>
<td>0xFFC0 2C6C</td>
<td>CAN_MB19_DATA3</td>
<td>0xFFC0 2E6C</td>
</tr>
<tr>
<td>CAN_MB04_DATA3</td>
<td>0xFFC0 2C8C</td>
<td>CAN_MB20_DATA3</td>
<td>0xFFC0 2E8C</td>
</tr>
<tr>
<td>CAN_MB05_DATA3</td>
<td>0xFFC0 2CAC</td>
<td>CAN_MB21_DATA3</td>
<td>0xFFC0 2EAC</td>
</tr>
<tr>
<td>CAN_MB06_DATA3</td>
<td>0xFFC0 2CCC</td>
<td>CAN_MB22_DATA3</td>
<td>0xFFC0 2ECC</td>
</tr>
<tr>
<td>CAN_MB07_DATA3</td>
<td>0xFFC0 2CEC</td>
<td>CAN_MB23_DATA3</td>
<td>0xFFC0 2ECC</td>
</tr>
<tr>
<td>CAN_MB08_DATA3</td>
<td>0xFFC0 2D0C</td>
<td>CAN_MB24_DATA3</td>
<td>0xFFC0 2F0C</td>
</tr>
<tr>
<td>CAN_MB09_DATA3</td>
<td>0xFFC0 2D2C</td>
<td>CAN_MB25_DATA3</td>
<td>0xFFC0 2F2C</td>
</tr>
<tr>
<td>CAN_MB10_DATA3</td>
<td>0xFFC0 2D4C</td>
<td>CAN_MB26_DATA3</td>
<td>0xFFC0 2F4C</td>
</tr>
<tr>
<td>CAN_MB11_DATA3</td>
<td>0xFFC0 2D6C</td>
<td>CAN_MB27_DATA3</td>
<td>0xFFC0 2F6C</td>
</tr>
<tr>
<td>CAN_MB12_DATA3</td>
<td>0xFFC0 2D8C</td>
<td>CAN_MB28_DATA3</td>
<td>0xFFC0 2F8C</td>
</tr>
<tr>
<td>CAN_MB13_DATA3</td>
<td>0xFFC0 2DAC</td>
<td>CAN_MB29_DATA3</td>
<td>0xFFC0 2FAC</td>
</tr>
</tbody>
</table>
Mailbox Identifier Word Registers

Table 19-7. Mailbox Identifier Word 3 Register Addresses (Cont’d)

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN_MB14_DATA3</td>
<td>0xFFC0 2DCC</td>
<td>CAN_MB30_DATA3</td>
<td>0xFFC0 2FCC</td>
</tr>
<tr>
<td>CAN_MB15_DATA3</td>
<td>0xFFC0 2DEC</td>
<td>CAN_MB31_DATA3</td>
<td>0xFFC0 2FEC</td>
</tr>
</tbody>
</table>

Mailbox Identifier Word 2 (CAN_MBxx_DATA2)

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>Reset = 0xFFFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>XX XX XX XX XX XX XX XX XX</td>
<td></td>
</tr>
</tbody>
</table>

For memory-mapped addresses, see Table 19-8.

Data Field Byte 2

Figure 19-13. Mailbox Identifier Word 2

Table 19-8. Mailbox Identifier Word 2 Register Addresses

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN_MB00_DATA2</td>
<td>0xFFC0 2C08</td>
<td>CAN_MB16_DATA2</td>
<td>0xFFC0 2E08</td>
</tr>
<tr>
<td>CAN_MB01_DATA2</td>
<td>0xFFC0 2C28</td>
<td>CAN_MB17_DATA2</td>
<td>0xFFC0 2E28</td>
</tr>
<tr>
<td>CAN_MB02_DATA2</td>
<td>0xFFC0 2C48</td>
<td>CAN_MB18_DATA2</td>
<td>0xFFC0 2E48</td>
</tr>
<tr>
<td>CAN_MB03_DATA2</td>
<td>0xFFC0 2C68</td>
<td>CAN_MB19_DATA2</td>
<td>0xFFC0 2E68</td>
</tr>
<tr>
<td>CAN_MB04_DATA2</td>
<td>0xFFC0 2C88</td>
<td>CAN_MB20_DATA2</td>
<td>0xFFC0 2E88</td>
</tr>
<tr>
<td>CAN_MB05_DATA2</td>
<td>0xFFC0 2CA8</td>
<td>CAN_MB21_DATA2</td>
<td>0xFFC0 2EA8</td>
</tr>
<tr>
<td>CAN_MB06_DATA2</td>
<td>0xFFC0 2CC8</td>
<td>CAN_MB22_DATA2</td>
<td>0xFFC0 2EC8</td>
</tr>
<tr>
<td>CAN_MB07_DATA2</td>
<td>0xFFC0 2CE8</td>
<td>CAN_MB23_DATA2</td>
<td>0xFFC0 2EE8</td>
</tr>
<tr>
<td>CAN_MB08_DATA2</td>
<td>0xFFC0 2D08</td>
<td>CAN_MB24_DATA2</td>
<td>0xFFC0 2F08</td>
</tr>
<tr>
<td>CAN_MB09_DATA2</td>
<td>0xFFC0 2D28</td>
<td>CAN_MB25_DATA2</td>
<td>0xFFC0 2F28</td>
</tr>
<tr>
<td>CAN_MB10_DATA2</td>
<td>0xFFC0 2D48</td>
<td>CAN_MB26_DATA2</td>
<td>0xFFC0 2F48</td>
</tr>
<tr>
<td>CAN_MB11_DATA2</td>
<td>0xFFC0 2D68</td>
<td>CAN_MB27_DATA2</td>
<td>0xFFC0 2F68</td>
</tr>
</tbody>
</table>
### Table 19-8. Mailbox Identifier Word 2 Register Addresses (Cont’d)

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN_MB12_DATA2</td>
<td>0xFFC0 2D88</td>
<td>CAN_MB28_DATA2</td>
<td>0xFFC0 2F88</td>
</tr>
<tr>
<td>CAN_MB13_DATA2</td>
<td>0xFFC0 2DA8</td>
<td>CAN_MB29_DATA2</td>
<td>0xFFC0 2FA8</td>
</tr>
<tr>
<td>CAN_MB14_DATA2</td>
<td>0xFFC0 2DC8</td>
<td>CAN_MB30_DATA2</td>
<td>0xFFC0 2FC8</td>
</tr>
<tr>
<td>CAN_MB15_DATA2</td>
<td>0xFFC0 2DE8</td>
<td>CAN_MB31_DATA2</td>
<td>0xFFC0 2FE8</td>
</tr>
</tbody>
</table>

### Table 19-9. Mailbox Identifier Word 1 Register Addresses

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN_MB00_DATA1</td>
<td>0xFFC0 2C04</td>
<td>CAN_MB16_DATA1</td>
<td>0xFFC0 2E04</td>
</tr>
<tr>
<td>CAN_MB01_DATA1</td>
<td>0xFFC0 2C24</td>
<td>CAN_MB17_DATA1</td>
<td>0xFFC0 2E24</td>
</tr>
<tr>
<td>CAN_MB02_DATA1</td>
<td>0xFFC0 2C44</td>
<td>CAN_MB18_DATA1</td>
<td>0xFFC0 2E44</td>
</tr>
<tr>
<td>CAN_MB03_DATA1</td>
<td>0xFFC0 2C64</td>
<td>CAN_MB19_DATA1</td>
<td>0xFFC0 2E64</td>
</tr>
<tr>
<td>CAN_MB04_DATA1</td>
<td>0xFFC0 2C84</td>
<td>CAN_MB20_DATA1</td>
<td>0xFFC0 2E84</td>
</tr>
<tr>
<td>CAN_MB05_DATA1</td>
<td>0xFFC0 2CA4</td>
<td>CAN_MB21_DATA1</td>
<td>0xFFC0 2EA4</td>
</tr>
<tr>
<td>CAN_MB06_DATA1</td>
<td>0xFFC0 2CC4</td>
<td>CAN_MB22_DATA1</td>
<td>0xFFC0 2EC4</td>
</tr>
<tr>
<td>CAN_MB07_DATA1</td>
<td>0xFFC0 2CE4</td>
<td>CAN_MB23_DATA1</td>
<td>0xFFC0 2EE4</td>
</tr>
<tr>
<td>CAN_MB08_DATA1</td>
<td>0xFFC0 2D04</td>
<td>CAN_MB24_DATA1</td>
<td>0xFFC0 2F04</td>
</tr>
<tr>
<td>CAN_MB09_DATA1</td>
<td>0xFFC0 2D24</td>
<td>CAN_MB25_DATA1</td>
<td>0xFFC0 2F24</td>
</tr>
</tbody>
</table>

Mailbox Identifier Word 1 (CAN_MBxx_DATA1)

- **Reset = 0xXXXX**

Figure 19-14. Mailbox Identifier Word 1
Table 19-9. Mailbox Identifier Word 1 Register Addresses (Cont’d)

```
<table>
<thead>
<tr>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN_MB10_DATA1</td>
<td>0xFFC0 2D44</td>
<td>CAN_MB26_DATA1</td>
<td>0xFFC0 2F44</td>
</tr>
<tr>
<td>CAN_MB11_DATA1</td>
<td>0xFFC0 2D64</td>
<td>CAN_MB27_DATA1</td>
<td>0xFFC0 2F64</td>
</tr>
<tr>
<td>CAN_MB12_DATA1</td>
<td>0xFFC0 2D84</td>
<td>CAN_MB28_DATA1</td>
<td>0xFFC0 2F84</td>
</tr>
<tr>
<td>CAN_MB13_DATA1</td>
<td>0xFFC0 2DA4</td>
<td>CAN_MB29_DATA1</td>
<td>0xFFC0 2FA4</td>
</tr>
<tr>
<td>CAN_MB14_DATA1</td>
<td>0xFFC0 2DC4</td>
<td>CAN_MB30_DATA1</td>
<td>0xFFC0 2FC4</td>
</tr>
<tr>
<td>CAN_MB15_DATA1</td>
<td>0xFFC0 2DE4</td>
<td>CAN_MB31_DATA1</td>
<td>0xFFC0 2FE4</td>
</tr>
</tbody>
</table>
```

Mailbox Identifier Word 0 (CAN_MBxx_DATA0)

For memory-mapped addresses, see Table 19-10.

Figure 19-15. Mailbox Identifier Word 0

Table 19-10. Mailbox Identifier Word 0 Register Addresses

```
<table>
<thead>
<tr>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN_MB00_DATA0</td>
<td>0xFFC0 2C00</td>
<td>CAN_MB16_DATA0</td>
<td>0xFFC0 2E00</td>
</tr>
<tr>
<td>CAN_MB01_DATA0</td>
<td>0xFFC0 2C20</td>
<td>CAN_MB17_DATA0</td>
<td>0xFFC0 2E20</td>
</tr>
<tr>
<td>CAN_MB02_DATA0</td>
<td>0xFFC0 2C40</td>
<td>CAN_MB18_DATA0</td>
<td>0xFFC0 2E40</td>
</tr>
<tr>
<td>CAN_MB03_DATA0</td>
<td>0xFFC0 2C60</td>
<td>CAN_MB19_DATA0</td>
<td>0xFFC0 2E60</td>
</tr>
<tr>
<td>CAN_MB04_DATA0</td>
<td>0xFFC0 2C80</td>
<td>CAN_MB20_DATA0</td>
<td>0xFFC0 2E80</td>
</tr>
<tr>
<td>CAN_MB05_DATA0</td>
<td>0xFFC0 2CA0</td>
<td>CAN_MB21_DATA0</td>
<td>0xFFC0 2EA0</td>
</tr>
<tr>
<td>CAN_MB06_DATA0</td>
<td>0xFFC0 2CC0</td>
<td>CAN_MB22_DATA0</td>
<td>0xFFC0 2EC0</td>
</tr>
<tr>
<td>CAN_MB07_DATA0</td>
<td>0xFFC0 2CE0</td>
<td>CAN_MB23_DATA0</td>
<td>0xFFC0 2EE0</td>
</tr>
</tbody>
</table>
```
Controller Area Network (CAN) Module

Receive mailboxes use a temporary receive buffer, which is only overwritten by incoming bytes based on the message’s DLC code. For DLC codes less than 8, the least significant bytes are not overwritten and are stored to the receiving mailbox. Only the data bytes defined by the DLC contain valid data.

Software should check the DLC code of the received message to determine which bytes in the CAN_MBxx_DATAx registers to read.

Mailbox Area

Most, but not all, CAN mailboxes can be configured to transmit or receive. Each mailbox has an acceptance mask. Mailbox 31 is the highest numbered mailbox.

Table 19-10. Mailbox Identifier Word 0 Register Addresses (Cont’d)

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN_MB08_DATA0</td>
<td>0xFFC0 2D00</td>
<td>CAN_MB24_DATA0</td>
<td>0xFFC0 2F00</td>
</tr>
<tr>
<td>CAN_MB09_DATA0</td>
<td>0xFFC0 2D20</td>
<td>CAN_MB25_DATA0</td>
<td>0xFFC0 2F20</td>
</tr>
<tr>
<td>CAN_MB10_DATA0</td>
<td>0xFFC0 2D40</td>
<td>CAN_MB26_DATA0</td>
<td>0xFFC0 2F40</td>
</tr>
<tr>
<td>CAN_MB11_DATA0</td>
<td>0xFFC0 2D60</td>
<td>CAN_MB27_DATA0</td>
<td>0xFFC0 2F60</td>
</tr>
<tr>
<td>CAN_MB12_DATA0</td>
<td>0xFFC0 2D80</td>
<td>CAN_MB28_DATA0</td>
<td>0xFFC0 2F80</td>
</tr>
<tr>
<td>CAN_MB13_DATA0</td>
<td>0xFFC0 2DA0</td>
<td>CAN_MB29_DATA0</td>
<td>0xFFC0 2FA0</td>
</tr>
<tr>
<td>CAN_MB14_DATA0</td>
<td>0xFFC0 2DC0</td>
<td>CAN_MB30_DATA0</td>
<td>0xFFC0 2FC0</td>
</tr>
<tr>
<td>CAN_MB15_DATA0</td>
<td>0xFFC0 2DE0</td>
<td>CAN_MB31_DATA0</td>
<td>0xFFC0 2FE0</td>
</tr>
</tbody>
</table>
Mailbox Types

Mailboxes are used for receiving and transmitting. Eight mailboxes are for transmit only, eight are for receive only, and 16 are configurable. Only the configurable mailboxes support the remote frame-request feature. The mailbox control register area consists of these registers:

- **CAN_MC1** and **CAN_MC2** (mailbox enable registers)
- **CAN_MD1** and **CAN_MD2** (mailbox direction registers)
- **CAN_TA1** and **CAN_TA2** (transmit acknowledge registers)
- **CAN_AA1** and **CAN_AA2** (abort acknowledge registers)
- **CAN_TRS1** and **CAN_TRS2** (transmit request set registers)
- **CAN_TRR1** and **CAN_TRR2** (transmit request reset registers)
- **CAN_RMP1** and **CAN_RMP2** (receive message pending registers)
- **CAN_RML1** and **CAN_RML2** (receive message lost registers)
- **CAN_RFH1** and **CAN_RFH2** (remote frame handling registers)
- **CAN_OPSS1** and **CAN_OPSS2** (overwrite protection/single shot transmission registers)
- **CAN_MBIN1** and **CAN_MBIN2** (mailbox interrupt mask registers)
- **CAN_MBTIF1** and **CAN_MBTIF2** (mailbox transmit interrupt flag registers)
- **CAN_MBRIF1** and **CAN_MBRIF2** (mailbox receive interrupt flag registers)
Mailbox Control

The mailbox configuration (CAN_MCx) and mailbox direction (CAN_MDx) registers configure the CAN mailboxes. Each mailbox can be enabled or disabled separately.

CAN Mailbox Configuration (CAN_MCx) and Direction (CAN_MDx) Registers

The mailbox configuration (CAN_MCx) registers are used to enable or disable each mailbox. The mailbox direction (CAN_MDx) registers determine which mailboxes are used for transmit and which for receive. If the MCn bit in the CAN_MCx register is zero, the corresponding mailbox (mailbox n) is disabled. The mailbox must be disabled before writing to any identifier field. Refer to Figure 19-16 and Figure 19-17.

Do not write to the identifier of a message object while the mailbox is enabled for the CAN module (the corresponding bit in CAN_MCx is set). Mailboxes that are disabled may be used as additional memory for the CPU.

If a mailbox is used for transmission, the corresponding bit in the configuration register (MCn) and in the direction register (MDn) must be set before the transmission request set (TRSn) bit is set.

To disable a mailbox, the corresponding bits in the transmit request reset register (CAN_TRRx) and the transmit request set register (CAN_TRSx) must first be reset by the internal logic. Setting TRRn of the CAN_TRRx register and TRSn of the CAN_TRSx register in a disabled mailbox can cause undefined behavior of the CAN module.

If a mailbox is used for receiving (MDn = 1 in the CAN_MDx register) and is disabled, an ongoing receive message for this mailbox is lost even if a second mailbox is configured to receive the same identifier. This happens if
the mailbox is disabled \( MC_n = 0 \) in the CAN_MCx register) after the internal acceptance filtering run is finished and before the reception of this message is completed.

Figure 19-16. Mailbox Configuration Register 1

Mailbox Configuration Register 1 (CAN_MC1)
For all bits, 0 - Mailbox disabled, 1 - Mailbox enabled

0xFFC0 2A00

Reset = 0x0000

Figure 19-17. Mailbox Configuration Register 2

Mailbox Configuration Register 2 (CAN_MC2)
For all bits, 0 - Mailbox disabled, 1 - Mailbox enabled

0xFFC0 2A40

Reset = 0x0000
The **CAN_MD1** and **CAN_MD2** registers determine the direction of each of the mailboxes. By default, mailboxes 7–0 are configured as receive mailboxes and mailboxes 31–8 are configured as transmit mailboxes. Writing a 1 to the $MD_n$ bit for mailboxes 23–8 configures the corresponding mailbox to be a receive mailbox. Mailboxes 31–24 are always transmit mailboxes. Do not write to $MD_n$ if the mailbox is enabled, that is, if the corresponding bit in the configuration register $MC_n$ is set. Thus, the mailbox must be disabled before $MD_n$ is changed.

After software reset, the bits in **CAN_MD1** and **CAN_MD2** are cleared, except bits 7–0 in the **CAN_MD1** register.

Bits 7–0 in **CAN_MD1** and bits 15–8 in **CAN_MD2** are read-only.

Changing a bit in the mailbox direction register ($MD_n$) may lead to erroneous behavior if the corresponding mailbox is enabled ($MC_n = 1$).

**Mailbox Direction Register 1 (CAN_MD1)**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>MD15 - RO</td>
</tr>
<tr>
<td>14</td>
<td>MD14 - RO</td>
</tr>
<tr>
<td>13</td>
<td>MD13 - RO</td>
</tr>
<tr>
<td>12</td>
<td>MD12 - RO</td>
</tr>
<tr>
<td>11</td>
<td>MD11 - RO</td>
</tr>
<tr>
<td>10</td>
<td>MD10 - RO</td>
</tr>
<tr>
<td>9</td>
<td>MD9  - RO</td>
</tr>
<tr>
<td>8</td>
<td>MD8  - RO</td>
</tr>
<tr>
<td>7</td>
<td>MD7  - RO</td>
</tr>
<tr>
<td>6</td>
<td>MD6  - RO</td>
</tr>
<tr>
<td>5</td>
<td>MD5  - RO</td>
</tr>
<tr>
<td>4</td>
<td>MD4  - RO</td>
</tr>
<tr>
<td>3</td>
<td>MD3  - RO</td>
</tr>
<tr>
<td>2</td>
<td>MD2  - RO</td>
</tr>
<tr>
<td>1</td>
<td>MD1  - RO</td>
</tr>
<tr>
<td>0</td>
<td>MD0  - RO</td>
</tr>
</tbody>
</table>

For all bits, 0 - Mailbox configured as transmit mode, 1 - Mailbox configured as receive mode

Reset = 0xFF

Figure 19-18. Mailbox Direction Register 1
Receive Logic

If a message is received from the CAN bus and a matching mailbox is detected by the internal compare logic, the content of the received message is stored in the matching message center. The complete received identifier, the RTR bit, and the corresponding identifier extension bit (IDE) are stored in the first two words of the destination mailbox. The acceptance mask enable configuration bit (AME) of this mailbox is unchanged. If a base message is received, the extended part of the identifier in the mailbox is also unchanged. The DLC and the value of the time stamp counter are stored in the next two words and the received data field is stored in words 3 to 0 of this mailbox. The complete content of the temporary receive buffer is stored in the mailbox regardless of the DLC value of the received message. Only the data bytes defined by the DLC contain valid data, the rest of the mailbox data field is undefined. Refer to Figure 19-18 and Figure 19-19.
After a message is stored in a mailbox, the corresponding receive message pending bit ($RMP_n$ in the $CAN_{RMPx}$ register) is set and a mailbox receive interrupt is generated (if enabled).

**Acceptance Filter/Data Acceptance Filter**

Each incoming data frame is compared to all identifiers stored in active receive mailboxes ($MD_n = 1$ and $MC_n = 1$) and to all active transmit mailboxes with the remote frame handling feature enabled. If the acceptance filter finds a matching identifier, the content of the received data frame is stored in this mailbox and the corresponding bit in the receive message pending register ($CAN_{RMPx}$) is set. A received message is stored only once, even if multiple receive mailboxes match its identifier. If the current identifier does not match any mailbox, the message is not stored. The $RMP_n$ bit must be reset to 0.

If a second message was received for this mailbox and the $RMP_n$ bit is already set and the overwrite protection bit ($OPSS_n$) is not set, the corresponding message lost bit ($RML_n$) is set. If the $OPSS_n$ bit is set, the next mailboxes are checked.

If an acceptance mask is enabled, each bit of the received identifier is ignored by the compare logic if the corresponding bit in the acceptance mask is set to one.
Table 19-11. Mailbox Used for Acceptance Mask Filtering

<table>
<thead>
<tr>
<th>MCn</th>
<th>MDn</th>
<th>RFHn</th>
<th>Mailbox n</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>x</td>
<td>Ignored</td>
<td>Mailbox n disabled</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Ignored</td>
<td>Mailbox n enabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Mailbox n configured for transmit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Remote frame handling disabled</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Used</td>
<td>Mailbox n enabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Mailbox n configured for transmit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Remote frame handling enabled</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>x</td>
<td>Used</td>
<td>Mailbox n enabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Mailbox n configured for receive</td>
</tr>
</tbody>
</table>

The use of the acceptance mask can be enabled/disabled for each mailbox separately.

**CAN Acceptance Mask (CAN_AMxx) Registers**

The acceptance mask registers `CAN_AMxxH` and `CAN_AMxxL` are used for acceptance filtering.

The acceptance masks are implemented as RAM, and have no reset value. Each acceptance mask must be reset manually before it is enabled by setting the AME bit in the corresponding mailbox.

**Acceptance Mask Register (CAN_AMxxH)**

![Figure 19-20. Acceptance Mask Register (H)](X)
Table 19-12. Acceptance Mask Register (H) Addresses

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN_AM00H</td>
<td>0xFFC0 2B04</td>
<td>CAN_AM16H</td>
<td>0xFFC0 2B84</td>
</tr>
<tr>
<td>CAN_AM01H</td>
<td>0xFFC0 2B0C</td>
<td>CAN_AM17H</td>
<td>0xFFC0 2B8C</td>
</tr>
<tr>
<td>CAN_AM02H</td>
<td>0xFFC0 2B14</td>
<td>CAN_AM18H</td>
<td>0xFFC0 2B94</td>
</tr>
<tr>
<td>CAN_AM03H</td>
<td>0xFFC0 2B1C</td>
<td>CAN_AM19H</td>
<td>0xFFC0 2B9C</td>
</tr>
<tr>
<td>CAN_AM04H</td>
<td>0xFFC0 2B24</td>
<td>CAN_AM20H</td>
<td>0xFFC0 2BA4</td>
</tr>
<tr>
<td>CAN_AM05H</td>
<td>0xFFC0 2B2C</td>
<td>CAN_AM21H</td>
<td>0xFFC0 2BAC</td>
</tr>
<tr>
<td>CAN_AM06H</td>
<td>0xFFC0 2B34</td>
<td>CAN_AM22H</td>
<td>0xFFC0 2BB4</td>
</tr>
<tr>
<td>CAN_AM07H</td>
<td>0xFFC0 2B3C</td>
<td>CAN_AM23H</td>
<td>0xFFC0 2BBC</td>
</tr>
<tr>
<td>CAN_AM08H</td>
<td>0xFFC0 2B44</td>
<td>CAN_AM24H</td>
<td>0xFFC0 2BC4</td>
</tr>
<tr>
<td>CAN_AM09H</td>
<td>0xFFC0 2B4C</td>
<td>CAN_AM25H</td>
<td>0xFFC0 2BCC</td>
</tr>
<tr>
<td>CAN_AM10H</td>
<td>0xFFC0 2B54</td>
<td>CAN_AM26H</td>
<td>0xFFC0 2BD4</td>
</tr>
<tr>
<td>CAN_AM11H</td>
<td>0xFFC0 2B5C</td>
<td>CAN_AM27H</td>
<td>0xFFC0 2BDC</td>
</tr>
<tr>
<td>CAN_AM12H</td>
<td>0xFFC0 2B64</td>
<td>CAN_AM28H</td>
<td>0xFFC0 2BE4</td>
</tr>
<tr>
<td>CAN_AM13H</td>
<td>0xFFC0 2B6C</td>
<td>CAN_AM29H</td>
<td>0xFFC0 2BEC</td>
</tr>
<tr>
<td>CAN_AM14H</td>
<td>0xFFC0 2B74</td>
<td>CAN_AM30H</td>
<td>0xFFC0 2BF4</td>
</tr>
<tr>
<td>CAN_AM15H</td>
<td>0xFFC0 2B7C</td>
<td>CAN_AM31H</td>
<td>0xFFC0 2BFC</td>
</tr>
</tbody>
</table>

If DeviceNet mode is enabled (the DNM bit of the CAN_CONTROL register is 1) and the mailbox is set up for filtering on data field, the filtering is done on the standard ID of the message and data fields. The data field filtering can be programmed for either the first byte only or the first two bytes, as shown in Table 19-13.
Table 19-13. Data Field Filtering

<table>
<thead>
<tr>
<th>FDF Filter On Data Field</th>
<th>FMD Full Mask Data Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Do not allow filtering on the data field</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Not allowed. FMD must be 0 if FDF is 0.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Filter on first data byte only</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Filter on first two data bytes</td>
</tr>
</tbody>
</table>

Acceptance Mask Register (CAN_AMxxL)

For memory-mapped addresses, see Table 19-14.

Figure 19-21. Acceptance Mask Register (L)

If the FDF field of the corresponding CAN_AMxxH register is 1, the CAN_AMxxL register holds the data field mask (DFM[15:0]). If the FDF field of the corresponding CAN_AMxxH register is 0, the CAN_AMxxL register holds the extended identifier mask (EXTID[15:0]).
Controller Area Network (CAN) Module
Figure 19-20 and Figure 19-21 show the layout for every implemented acceptance mask register (CAN_AMxxH and CAN_AMxxL).

The acceptance filtering is done to allow groups of messages to be stored in a message center.

An incoming message is stored in the highest numbered mailbox with a matching identifier. If this mailbox already contains data (RMPn = 1 in the CAN_RMPx register), the further behavior depends on the content of the corresponding overwrite protection bit (OPSSn). The incoming identifier is compared to those stored in the RAM and the bits that should not be

Table 19-14. Acceptance Mask Register (L) Addresses

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN_AM00L</td>
<td>0xFFC0 2B00</td>
<td>CAN_AM16L</td>
<td>0xFFC0 2B80</td>
</tr>
<tr>
<td>CAN_AM01L</td>
<td>0xFFC0 2B08</td>
<td>CAN_AM17L</td>
<td>0xFFC0 2B88</td>
</tr>
<tr>
<td>CAN_AM02L</td>
<td>0xFFC0 2B10</td>
<td>CAN_AM18L</td>
<td>0xFFC0 2B90</td>
</tr>
<tr>
<td>CAN_AM03L</td>
<td>0xFFC0 2B18</td>
<td>CAN_AM19L</td>
<td>0xFFC0 2B98</td>
</tr>
<tr>
<td>CAN_AM04L</td>
<td>0xFFC0 2B20</td>
<td>CAN_AM20L</td>
<td>0xFFC0 2BA0</td>
</tr>
<tr>
<td>CAN_AM05L</td>
<td>0xFFC0 2B28</td>
<td>CAN_AM21L</td>
<td>0xFFC0 2BA8</td>
</tr>
<tr>
<td>CAN_AM06L</td>
<td>0xFFC0 2B30</td>
<td>CAN_AM22L</td>
<td>0xFFC0 2BB0</td>
</tr>
<tr>
<td>CAN_AM07L</td>
<td>0xFFC0 2B38</td>
<td>CAN_AM23L</td>
<td>0xFFC0 2BB8</td>
</tr>
<tr>
<td>CAN_AM08L</td>
<td>0xFFC0 2B40</td>
<td>CAN_AM24L</td>
<td>0xFFC0 2BC0</td>
</tr>
<tr>
<td>CAN_AM09L</td>
<td>0xFFC0 2B48</td>
<td>CAN_AM25L</td>
<td>0xFFC0 2BC8</td>
</tr>
<tr>
<td>CAN_AM10L</td>
<td>0xFFC0 2B50</td>
<td>CAN_AM26L</td>
<td>0xFFC0 2BD0</td>
</tr>
<tr>
<td>CAN_AM11L</td>
<td>0xFFC0 2B58</td>
<td>CAN_AM27L</td>
<td>0xFFC0 2BD8</td>
</tr>
<tr>
<td>CAN_AM12L</td>
<td>0xFFC0 2B60</td>
<td>CAN_AM28L</td>
<td>0xFFC0 2BE0</td>
</tr>
<tr>
<td>CAN_AM13L</td>
<td>0xFFC0 2B68</td>
<td>CAN_AM29L</td>
<td>0xFFC0 2BE8</td>
</tr>
<tr>
<td>CAN_AM14L</td>
<td>0xFFC0 2B70</td>
<td>CAN_AM30L</td>
<td>0xFFC0 2BF0</td>
</tr>
<tr>
<td>CAN_AM15L</td>
<td>0xFFC0 2B78</td>
<td>CAN_AM31L</td>
<td>0xFFC0 2BF8</td>
</tr>
</tbody>
</table>
compared are masked out. A 1 in the acceptance mask means don’t care and a 0 demands an identical match of the bit. The mask bits for the base identifier are stored in bits 12–2 of the acceptance mask registers (CAN_AMxxH) and bits for the extended identifier in 1–0 of CAN_AMxxH and 15–0 of CAN_AMxxL. Bit 13 of the CAN_AMxxH register is the mask bit for the identifier extension bit (AMIDE).

After power-up reset, the user must initialize all acceptance mask bits.

The acceptance mask area is implemented as a separate acceptance mask for every mailbox, so the reset value for software reset and power-up reset are unchanged. If the acceptance mask is enabled (AME of corresponding mailbox is set), it has to be initialized.

The content of the acceptance mask register may be changed only if the corresponding mailboxes are disabled.

Receive Control Registers

The following sections describe the receive control registers of the CAN module.

CAN Receive Message Pending (CAN_RMPx) Register

The bits in the receive message pending register (CAN_RMPx) can only be set by the internal logic. The RMPn bit indicates a message pending in mailbox n. The RMPn bits are write-1-to-clear (W1C). Clearing a RMPn bit also clears the corresponding receive message lost bit (RMLn) in the CAN_RMLx register. The RMPn bit may set the mailbox interrupt flag (MBRIFn) bit in the mailbox interrupt flag register (CAN_MBRIFx) if the corresponding interrupt mask bit in the MBIMn mailbox interrupt mask register (CAN_MBIMx) is set. The MBRIFn flag initiates a mailbox interrupt.
Receive Control Registers

**Receive Message Pending Register 1 (CAN_RMP1)**
All bits are write-1-to-clear

- **0xFFC0 2A18**
  - Bits: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  - Reset = 0x0000

  - RMP15
  - RMP14
  - RMP13
  - RMP12
  - RMP11
  - RMP10
  - RMP9
  - RMP8

  Figure 19-22. Receive Message Pending Register 1

**Receive Message Pending Register 2 (CAN_RMP2)**
All bits are write-1-to-clear

- **0xFFC0 2A58**
  - Bits: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  - Reset = 0x0000

  - RMP31 - RO
  - RMP30 - RO
  - RMP29 - RO
  - RMP28 - RO
  - RMP27 - RO
  - RMP26 - RO
  - RMP25 - RO
  - RMP24 - RO

  Figure 19-23. Receive Message Pending Register 2

**CAN Receive Message Lost (CAN_RMLx) Register**

The bits in the receive message lost register (CAN_RMLx) can only be reset by the device and can be set by the internal logic. The bits can be cleared by writing a 1 to RMPn in the CAN_RMPx register. A write access to the CAN_RMLx register has no effect.
If one or more bits in the CAN_RMLx register is set, the receive message lost interrupt status bit in the global interrupt status register (CAN_GIS) is also set. If the corresponding interrupt enable bit in CAN_GIM is set, the receive message lost flag (RMLIF) in the global interrupt flag register (CAN_GIF) is also set.

![Receive Message Lost Register 1 (CAN_RML1)](Figure 19-24. Receive Message Lost Register 1)

![Receive Message Lost Register 2 (CAN_RML2)](Figure 19-25. Receive Message Lost Register 2)
Receive Control Registers

CAN Overwrite Protection/Single Shot Transmission (CAN_OPSSx) Register

If a message is received for a mailbox and this mailbox still contains unread data (RMPn = 1), the user has to decide whether this old message should be overwritten or not. If the corresponding overwrite protection bit is reset (OPSSn = 0), the receive message lost bit (RMLn) is set and the stored message is overwritten.

If a message is received for a mailbox and this mailbox still contains unread data (RMPn = 1, OPSSn = 1), the next mailboxes are checked for another matching identifier.

The meaning of the bits in the overwrite protection/single shot transmission mode register (CAN_OPSSx) depends on the corresponding mailbox configuration. If a mailbox is configured as a receive mailbox, the content of OPSSn is interpreted as overwrite protection bit (OPSSn). If a mailbox is configured as a transmit mailbox, OPSSn is interpreted as single shot transmission mode bit (OPSSn). These bits can only be set/reset by the device. After power-up reset or software reset, all bits are cleared.

![Overwrite Protection/Single Shot Transmission Register 1 (CAN_OPSS1)](image)

Figure 19-26. Overwrite Protection/Single Shot Transmission Register 1
If the mailbox configuration is changed (receive mode <-> transmit mode), the content of the CAN_OPSSx register must be adapted by the user.

The content of a CAN_OPSSx bit must not be changed if the corresponding mailbox is enabled.

The overwrite protection cannot be used if automatic remote frame handling is enabled. In this case, the content of a mailbox is always overwritten by an incoming message.

**Transmit Logic**

The transmit data is stored in a mailbox configured as a transmit mailbox. After writing the data and the identifier into the RAM, the message is sent if the corresponding transmit request bit is set and the mailbox is enabled. The transmit control register is divided into two registers. The transmit request set register (CAN_TRSx) and the transmit request reset register (CAN_TRRx).
Transmit Logic

If there is more than one pending transmit request, the message objects are sent as defined in the transmit priority logic.

When transmission is successful, the corresponding bits in the `CAN_TRSx` register and in the `CAN_TRRx` register are cleared and the corresponding bit in the transmit acknowledge register is set.

The control bits to set or reset a transmission request (TRS and TRR, respectively) can be written independently.

Retransmission

Normally, the current message object is resent in case of a lost arbitration or an error frame on the CAN bus line. If there is more than one transmit message object pending, the message object with the highest priority is sent first. The priority is defined by the transmit priority logic. The currently aborted transmission is restarted after the message with the higher priority is sent.

A message which is currently under preparation is not replaced by another message which is written into the mailbox. The message under preparation is one that is copied into the temporary transmit buffer when the internal transmit request for the CAN core module is set. The message is not replaced until it is sent successfully, the arbitration on the CAN bus line is lost, or there is an error frame on the CAN bus line.

Single Shot Transmission

If the single shot transmission feature is used (OPSSn = 1), the transmission request set bit (TRSn) in the `CAN_TRSx` register is cleared after the message is successfully sent. The TRSn bit is also cleared if the transmission is aborted due to a lost arbitration or an error frame on the CAN bus line. Thus, the transmission of this message is not repeated in case of a lost arbitration or an error on the CAN bus line.
After a successful transmission, the corresponding $T_{An}$ bit in the $CAN\_T{A}_x$ register is set, and after an aborted transmission, the corresponding $A_{An}$ bit is set.

**Transmit Priority Defined by Mailbox Number**

If there is more than one pending transmit request, the sequence is started with the highest enabled mailbox down to the lowest enabled mailbox. The pointer to the next pending transmit message is generated from the content of the $CAN\_TRSx$, $CAN\_TRRx$, $CAN\_MDx$, and $CAN\_MCx$ registers. This pointer is available one cycle after a change to one of the registers. Thus, the new pointer is generated shortly before the content of the message to be sent is copied into the temporary transmit buffer. This normally happens during the intermission field of a CAN message. After this pointer is generated, all further changes in the mailbox area are ignored until the next pointer generation event.

**Transmit Control Registers**

If a message is to be sent, the corresponding mailbox must first be configured as a transmit mailbox. After the data is stored in the mailbox, the transmission can be initiated by setting the corresponding bit in the transmit request set register. A requested transmission can be aborted by setting the corresponding bit in $CAN\_TRRx$.

**CAN Transmission Request Set (CAN\_TRSx) Registers**

The bits in the $CAN\_TRSx$ register can be set by the device and reset/set by the internal logic. The $CAN\_TRSx$ bits are set by writing a 1. Writing a 0 has no effect. They are set by the CAN module in case of a remote frame request (or in auto-transmit mode). This is only possible for the receive/transmit mailboxes if the automatic remote frame handling feature
is enabled ($RFHn = 1$). If $TRSn = 1$, the write access to the corresponding mailbox is denied (but not locked) and message $n$ is transmitted. Several $CAN\_TRSx$ bits can be set simultaneously and are reset after either a successful or an aborted transmission.

After power-up reset or software reset, all bits in $CAN\_TRSx$ are cleared. The $CAN\_TRSx$ bits are only implemented for transmit mailboxes and standard mailboxes. The value of $TRSn$ for a receive mailbox is always read as 0.

Write access to a mailbox is possible, even if the corresponding $TRSn$ bit is set. However, changing data in such a mailbox may lead to unexpected data during transmission.

Setting $TRSn$ when the corresponding mailbox is disabled ($MCn = 0$) may lead to erroneous behavior.

Unexpected behavior may also occur if a mailbox is disabled before the corresponding $TRSn$ bit is reset by the internal logic.

Before $TRSn$ is set, the corresponding mailbox must contain valid transmit data.

Transmission Request Set Register 1 ($CAN\_TRS1$)

![Figure 19-28. Transmission Request Set Register 1](image-url)
Controller Area Network (CAN) Module

CAN Transmission Request Reset (CAN_TRRx) Registers

The bits in the CAN_TRRx registers can only be set by the device and reset by the internal logic. The CAN_TRR bits are set by writing a 1. Writing a 0 has no effect. After power-up reset or software reset, all bits are cleared.

If TRRn is set, the write access to the corresponding mailbox is denied but not locked. If TRRn is set and the transmission which was initiated by TRSn is not currently processed, the corresponding transmission request is cancelled immediately. If the corresponding message is currently being processed, the bits in CAN_TRSx and CAN_TRRx remain set until the transmission is aborted or successfully finished. The abort acknowledge bit (AAn) or the transmit acknowledge bit (TAn) is not set until:

- Successful transmission
- Abortion due to a lost arbitration
- Error condition detected on the CAN bus line
If the transmission was successful, the transmit acknowledge bit ($TAn$) is set. If the transmission was aborted, the corresponding abort acknowledge bit ($AAn$) is set. In both cases, $TRSn$ and $TRRn$ are reset.

The status of the $TRR$ bits can be read from the $CAN_{TRSx}$ bits. If $CAN_{TRSx}$ is set and a transmission is taking place, $CAN_{TRRx}$ can only be reset by the actions described above. If the $CAN_{TRSx}$ bit is reset and the $CAN_{TRRx}$ bit is set, there is no effect since the $CAN_{TRRx}$ bit is immediately reset by internal logic.

After power-up reset or software reset, all bits in $CAN_{TRRx}$ are cleared. The $CAN_{TRRx}$ bits are only implemented for transmit mailboxes and standard mailboxes. The value of $TRRn$ for a receive mailbox is always read as 0.

The $TRRn$ bit must not be set if the corresponding mailbox is disabled ($MCn = 0$).

The $TRRn$ bit must not be set if the corresponding $TRSn$ bit is not set.

A currently processed message continues to transmit if the corresponding bits $TRSn$ and $TRRn$ are set because of an abort requested by the user. The current transmit operation is finished if $TAn$ of the $CAN_{TAx}$ register or $AAn$ of the $CAN_{AAx}$ register is set.

The transmission of a message is immediately aborted if the corresponding mailbox is temporary disabled and the $TRRn$ bit for this message is set ($TRSn$ and $TRRn$ are reset, $AAn$ is set).
Transmission Request Reset Register 1 (CAN_TRR1)

Figure 19-30. Transmission Request Reset Register 1

Transmission Request Reset Register 2 (CAN_TRR2)

Figure 19-31. Transmission Request Reset Register 2
**CAN Abort Acknowledge (CAN_AAx) Register**

The abort acknowledge register (CAN_AAx) indicates whether a transmission was aborted. If the transmission of the message in mailbox n was aborted, bit AAn is set. The AAn bits are write-1-to-clear bits. Writing a 0 has no effect. The abort acknowledge bit AAn is reset if TRSn is set again. If a mailbox is disabled (MCn is reset) and the corresponding AAn bit in the transmit abort register is set, this bit remains set. The AAn bit can only be cleared by writing a 1 to it.

Setting a bit in CAN_AAx sets a flag (AAIS) in the global interrupt status register (CAN_GIS). If the interrupt mask bit AAIM is set (AAIM = 1 => interrupt is enabled), the corresponding AAIF bit in the global interrupt flag register (CAN_GIF) is also set and a global interrupt is asserted.

After power-up reset or software reset, all bits in CAN_AAx are cleared.

![Abort Acknowledge Register 1](image)

Figure 19-32. Abort Acknowledge Register 1
The $AA_n$ bit is not reset if the corresponding $T_{RN}$ bit is set by internal logic. The $AA_n$ bit is only reset if $T_{RN}$ is set.

**CAN Transmission Acknowledge (CAN_TAx) Register**

The transmission acknowledge register (CAN_TAx) indicates whether a transmission was sent successfully. If the message in mailbox n was sent successfully, the transmit acknowledge bit ($T_{An}$) is set. The $T_{An}$ bits are write-1-to-clear. Writing a 0 has no effect. The $T_{An}$ bit is also reset if $T_{RN}$ is set again. If a mailbox is disabled ($MC_n$ is reset) and the corresponding bit in the CAN_TAx register is set, this bit remains set. The $T_{An}$ bit can only be cleared by writing a 1 to it.

Setting a bit in CAN_TA sets a mailbox transmit interrupt flag (MBTIFn) if the corresponding interrupt mask bit (MBIMn in the CAN_MBIMx register) is set ($MBIMn = 1 \Rightarrow$ interrupt is enabled).

After power-up reset or software reset, all bits in CAN_TAx are cleared.

Figure 19-33. Abort Acknowledge Register 2
The $TAn$ bit is not reset if the corresponding $TRS_n$ bit is set by internal logic. The $TAn$ bit is only reset if $TRS_n$ is set or by writing a 1 to the corresponding bit location.
Controller Area Network (CAN) Module

CAN Mailbox Temporary Disable (CAN_MBTD) Register

The mailbox temporary disable feature can be used by programming the mailbox temporary disable register (CAN_MBTD). If a mailbox is enabled and configured to transmit, write accesses to the data field are denied. If this mailbox is used for automatic remote frame handling, the data field must be updated without losing an incoming remote request frame and without sending inconsistent data.

The pointer to the requested mailbox must be written to bits 4 to 0 of the CAN_MBTD register and the mailbox TDR bit must be set. The corresponding mailbox TDA flag is set by the internal logic.

If a mailbox is configured to transmit ($MDn = 0$) and the TDA flag is set by the FSM, the content of the data field of that mailbox can be updated. If there is an incoming remote request frame while the mailbox is temporary disabled, the corresponding TRSn bit is set by the FSM and the DLC of the incoming message is written to the corresponding mailbox. However, the message is not sent until the temporary disable request is reset.

If a mailbox is configured to receive ($MDn = 1$), the TDA flag is set by the FSM and the mailbox is not currently processed. If there is an incoming message for the requested mailbox n (the number of mailbox n is identical to the number of the temporary disabled mailbox), the internal logic waits until the reception is complete or there is an error on the CAN bus or until the TDA flag is set. If the TDA flag is set, the mailbox can be completely disabled ($MCn = 0$) without the risk of losing an incoming frame. The temporary disable request (TDR) must be reset as soon as possible.

If the TDA flag for a mailbox is set by the internal logic, only the data field of this mailbox can be updated (last 8 bytes of the mailbox). Accesses to the control bits and the Identifier are denied.

The temporary disabled mailbox is ignored for transmission as long as the corresponding request bit is set.
The reserved bits of the CAN_MBTD register always read 0. When writing this register, always write these bits as 0.

**CAN Remote Frame Handling (CAN_RFHx) Registers**

Automatic handling of remote frames can be enabled/disabled by setting/resetting the corresponding bit in the remote frame handling registers (CAN_RFHx).

Remote frames are data frames without a data field with the RTR bit set. The data length code of the data frame is equal to the DLC of the corresponding remote frame. A DLC can be programmed with values in the range of 0 to 15. The DLC value greater than 8 is considered as 8. A remote frame contains:

- the identifier bits
- the control bits, (that is, the data length code)
- the remote transmission request (RTR) bit

Only mailboxes that can both transmit and receive (mailboxes 8–23) can process remote frames.
All message centers can receive and transmit remote frame requests. They are capable of automatically sending a remote frame request to another node and answering incoming remote frame requests.

Note that a mailbox is only enabled for transmission or reception if its $MC_n$ bit in the mailbox configuration register is set. A disabled mailbox does not transmit or receive any messages.

Remote Frame Handling Register 1 (CAN_RFH1)

![Remote Frame Handling Register 1](image1)

Remote Frame Handling Register 2 (CAN_RFH2)

![Remote Frame Handling Register 2](image2)
CAN Interrupts

Write access to an RFHn bit is denied (but not locked) if the corresponding mailbox n is enabled. The CAN_RFHx register can only be read and written by the device. If bit RFHn is set, the automatic remote frame handling feature for the corresponding mailbox is enabled.

After power-up reset or software reset, all bits are cleared.

Erroneous behavior may result when the RFHn bit is changed and the corresponding mailbox is currently processed.

The length of a data frame is defined by the DLC of the corresponding remote frame. If a remote frame is received, the DLC of the corresponding mailbox is overwritten with the received value.

CAN Interrupts

The CAN module provides three independent interrupts: two mailbox interrupts (mailbox receive interrupt MBRIRQ and mailbox transmit interrupt MBTIRQ) and the global interrupt GIRQ. The values of these three interrupts can also be read back in the interrupt status registers.

CAN Interrupt (CAN_INTR) Register

The CAN interrupt register (CAN_INTR) holds information about the CAN interrupts.

The interrupt status bit is set as long as the interrupt output line is active. All bits in CAN_INTR are read only. Write access to this register has no effect. (Exception: wake up, if sleep mode is entered). After power-up reset or software reset, all interrupts are cleared.

The reserved bits of the CAN_INTR register always read 0. When writing this register, always write these bits as 0.
For test or debugging purposes, the values of the CAN serial-in (CANRX) and the CAN serial-out (CANTX) bits can also be read. These two bits show the state of the CANRX and CANTX pins, respectively, at the time the register was read.

**CAN Interrupt Register (CAN_INTR)**

![CAN Interrupt Register](image)

- **CANRX (Serial Input From Transceiver) - RO**
  - Serial input from CAN bus line from transceiver
  - 0 - Value is dominant
  - 1 - Value is recessive

- **CANTX (Serial Input To Transceiver) - RO**
  - Serial input from CAN bus line to transceiver
  - 0 - Value is dominant
  - 1 - Value is recessive

- **SMACK (Sleep Mode Acknowledge)**
  - 0 - Not in Sleep mode
  - 1 - Full-CAN module in Sleep mode

Figure 19-39. CAN Interrupt Register

Additional information for the CAN_INTR register bits includes:

- **Serial Input From CAN Bus Line From Transceiver (CANRX)**.
  - This bit is read-only.
  - 1 — The current value of the CAN bus is recessive.
  - 0 — The current value of the CAN bus is dominant.

- **Serial Output to CAN Bus Line to Transceiver (CANTX)**.
  - This bit is read-only.
  - 1 — The output to the CAN bus line is recessive.
  - 0 — The output to the CAN bus line is dominant.
Mailbox Interrupts

- **Global Interrupt Output** (GIRQ).
  
  1—At least one global interrupt flag in the global interrupt flag register CAN_GIF is set.
  
  0—None of the global interrupt flags is set.

- **Sleep Mode Acknowledge** (SMACK).
  
  1—The full-CAN module is in sleep mode.
  
  0—Sleep mode is not active.

- **Mailbox Transmit Interrupt Output** (MBTIRO).
  
  1—At least one transmit interrupt flag in the transmit interrupt flag register CAN_MBTIFx is set.
  
  0—No transmit interrupt flag bit in CAN_MBTIFx is set.

- **Mailbox Receive Interrupt Output** (MBRIRO).
  
  1—At least one receive interrupt flag in the receive interrupt flag register CAN_MBRIFx is set.
  
  0—No transmit interrupt flag bit in CAN_MBRIFx is set.

Mailbox Interrupts

Each of the 32 mailboxes in the CAN module may generate an interrupt as a result of one of the two mailbox interrupts. These interrupts can be receive or transmit interrupts, depending on the mailbox configuration.

A transmit (MBTIFn) or receive (MBRIFn) mailbox interrupt flag can only be set if the corresponding MBIMn bit in the mailbox interrupt mask register (CAN(mbIMx)) is set.
If a mailbox is configured as a receive mailbox, the corresponding receive interrupt flag (MBRIFn) is set after a received message is stored in mailbox n. If the automatic remote frame handling feature is used, the receive interrupt flag is set after the requested data frame is stored in the mailbox. The mailbox receive interrupt (MBRIFn) is always asserted if a new receive message is written to mailbox n and if MBIMn is set.

If a mailbox is configured as a transmit mailbox, the corresponding transmit interrupt flag (MBTIFn) is set after the message in mailbox n is sent correctly. If the automatic remote frame handling feature is used, the transmit interrupt flag is set after the requested data frame is sent from the mailbox.

**CAN Mailbox Interrupt Mask (CAN_MBIMx) Registers**

The mailbox interrupt mask registers (CAN_MBIMx) control whether or not mailbox interrupt flags are set. There is one interrupt flag available for each mailbox. This may be a receive or a transmit interrupt depending on the configuration register. If the MBIMn bit is 1, an interrupt is generated if a message has been transmitted successfully (for a transmit mailbox) or a message has been received without any errors (for a receive mailbox). After power-up or software reset, all interrupt mask bits are cleared and the interrupts are disabled.
Mailbox Interrupts

Mailbox Interrupt Mask Register 1 (CAN_MBIM1)

Figure 19-40. Mailbox Interrupt Mask Register 1

Mailbox Interrupt Mask Register 2 (CAN_MBIM2)

Figure 19-41. Mailbox Interrupt Mask Register 2
CAN Mailbox Interrupt Mask Flag
(CAN_MBTIFx) Registers

A transmit interrupt flag (MBTIFn) in the mailbox transmit interrupt flag register (CAN_MBTIFx) is set if a message is sent correctly from a mailbox, the corresponding bit MBIMn is set, and the mailbox is configured as a transmit mailbox. If the mailbox is configured as a receive mailbox (MDn = 1) or the mailbox is disabled (MCn = 0), the corresponding bit in the transmit mailbox interrupt flag register (MBTIFn) remains set.

A MBTIFn can be reset by writing a 1 to the corresponding bit location. Writing a 0 has no effect. The MBTIFn bit is also reset if the mailbox configuration register bit (MCn) is reset or the corresponding bit in the mailbox interrupt mask register (CAN_MBIMx) is reset.

The mailbox transmit interrupt output (MBTIRO) is active as long as at least one bit in the CAN_MBTIFx register is set.

If the mailbox direction (MDn) for a mailbox is changed after the MBTIFn bit has been set, the value of the MBTIFn bit is reset and the corresponding MBRIFn bit in the CAN_MBRIFx register is set.

<table>
<thead>
<tr>
<th>Mailbox Transmit Interrupt Flag Register 1 (CAN_MBTIF1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0 2A20</td>
</tr>
<tr>
<td>0x0000 0x0000 0x0000 0x0000 0x0000 0x0000 0x0000 0x0000</td>
</tr>
</tbody>
</table>

Figure 19-42. Mailbox Transmit Interrupt Flag Register 1
Mailbox Interrupts

Mailbox Transmit Interrupt Flag Register 2 (CAN_MBTIF2)

A receive interrupt flag bit (MBRIFn) in the mailbox receive interrupt flag register (CAN_MBRIFx) is set if a message is received and stored correctly in mailbox n, the corresponding bit MBIMn is set, and the mailbox is configured as a receive mailbox. If the mailbox is configured as a transmit mailbox (MDn = 0) or the mailbox is disabled (MCn = 0), the MBRIFn bit in the mailbox receive interrupt flag register remains set.

A receive interrupt flag bit (MBRIFn) can be reset by writing a 1 to the corresponding bit location. Writing a 0 has no effect. The MBRIFn bit is also reset if the MCn bit in the mailbox configuration register is reset, or the corresponding MBIMn bit in the mailbox interrupt mask register is reset.

The mailbox receive interrupt output (MBRIQ) is active as long as at least one bit in the CAN_MBRIFx register is set.

If the mailbox direction (MDn) is changed after the MBRIFn bit has been set, the value of MBRIFn is reset and the corresponding MBTIFn bit is set.
Controller Area Network (CAN) Module

Mailbox Receive Interrupt Flag Register 1 (CAN_MBRIF1)

![Figure 19-44. Mailbox Receive Interrupt Flag Register 1](image)

Mailbox Receive Interrupt Flag Register 2 (CAN_MBRIF2)

![Figure 19-45. Mailbox Receive Interrupt Flag Register 2](image)
Global Interrupt

In addition to the mailbox interrupts, the CAN module provides a global interrupt. There are several interrupt events to activate this interrupt. Global interrupts are generated if there is a change to some status bits in the CAN controller module. Each interrupt can be masked separately. All bits in the interrupt status and in the interrupt flag registers remain set until cleared by software or a software reset has occurred.

In the ISR, the interrupt latch should be cleared by a W1C operation to the corresponding bit of the CAN_GIS register. This clears the related bits of both the CAN_GIS and CAN_GIF registers.

The interrupt sources are:

- **Access Denied Interrupt** (ADIM, ADIS, ADIF).
  1—At least one access to the mailbox RAM occurred during a data update by internal logic.
  0—All accesses to the mailbox RAM are valid.

- **External Trigger Output Interrupt** (EXTIM, EXTIS, EXTIF).
  1—The external trigger event occurred.
  0—There was no external trigger event.

- **Universal Counter Exceeded Interrupt** (UCEIM, UCEIS, UCEF).
  There was an overflow of the universal counter (in time stamp mode or event counter mode) or the counter has reached the value 0x0000 (in watchdog mode).
  1—The universal counter was exceeded.
  0—The universal counter was not exceeded.
Controller Area Network (CAN) Module

- **Receive Message Lost Interrupt** (RMLIM, RMLIS, RMLIF).

A message has been received for a mailbox that currently contains unread data. At least one bit in the receive message lost register (CAN_RMLx) is set. If the bit in CAN_GIS (and CAN_GIF) is reset and there is at least one bit in CAN_RML still set, the bit in CAN_GIS (and CAN_GIF) is not set again. The internal interrupt source signal is only active if a new bit in CAN_RML is set.

1—At least one message has been lost.
0—No message lost event detected.

- **Abort Acknowledge Interrupt** (AAIM, AAIS, AAIF).

A requested transmission abort of a message was successful. At least one bit in the abort acknowledge registers CAN_AAx is set. If the bit in CAN_GIS (and CAN_GIF) is reset and there is at least one bit in CAN_AAx still set, the bit in CAN_GIS (and CAN_GIF) is not set again. The internal interrupt source signal is only active if a new bit in CAN_AAx is set.

1—At least one transmit request was successfully aborted.
0—No transmission aborted.

- **Access to Unimplemented Address Interrupt** (UIAIM, UIAIS, UIAIF).

There was a CPU access to an address which is not implemented in the controller module.

1—Access to unimplemented address detected.
0—No access to unimplemented address detected.
Global Interrupt

- **Wake-up Interrupt** (WUIM, WUIS, WUIF).
  
  1—The CAN module has left the sleep mode because of detected activity on the CAN bus line.
  
  0—The wake-up event is not active.

- **Bus-Off Interrupt** (BOIM, BOIS, BOIF).
  
  The CAN module has entered the bus-off state. This interrupt source is active if the status of the CAN core changes from normal operation mode to the bus-off mode. If the bit in CAN_GIS (and CAN_GIF) is reset and the bus-off mode is still active, this bit is not set again. If the module leaves the bus-off mode, the bit in CAN_GIS (and CAN_GIF) remains set.
  
  1—The CAN module has entered its bus-off mode.
  
  0—The CAN module has not entered the bus-off mode.

- **Error-Passive Interrupt** (EPIM, EPIS, EPIF).
  
  The CAN module has entered the error-passive state. This interrupt source is active if the status of the CAN module changes from the error-active mode to the error-passive mode. If the bit in CAN_GIS (and CAN_GIF) is reset and the error-passive mode is still active, this bit is not set again. If the module leaves the error-passive mode, the bit in CAN_GIS (and CAN_GIF) remains set.
  
  1—The CAN module is in error-passive mode.
  
  0—The CAN module has not entered error-passive mode.
Controller Area Network (CAN) Module

- **Error Warning Receive Interrupt** \((EWRIM, EWRIS, EWRIF)\).

  The CAN receive error counter \((RXECNT)\) has reached the warning limit. If the bit in \(CAN\_GIS\) (and \(CAN\_GIF\)) is reset and the error warning mode is still active, this bit is not set again. If the module leaves the error warning mode, the bit in \(CAN\_GIS\) (and \(CAN\_GIF\)) remains set.

  1—The warning level for the CAN receive error counter was exceeded.

  0—The warning level for the CAN receive error counter was not exceeded.

- **Error Warning Transmit Interrupt** \((EWTIM, EWTIS, EWTIF)\).

  The CAN transmit error counter \((TXECNT)\) has reached the warning limit. If the bit in \(CAN\_GIS\) (and \(CAN\_GIF\)) is reset and the error warning mode is still active, this bit is not set again. If the module leaves the error warning mode, the bit in \(CAN\_GIS\) (and \(CAN\_GIF\)) remains set.

  After a software reset, all bits in \(CAN\_GIF\), \(CAN\_GIS\), and \(CAN\_GIM\) are cleared.

  1—The warning level for the CAN transmit error counter was exceeded.

  0—The warning level for the CAN receive error counter was not exceeded.
Global Interrupt Logic

The global interrupt logic is implemented with three registers: the global interrupt mask register (CAN_GIM), where each interrupt source can be enabled or disabled separately; the global interrupt status register (CAN_GIS); and the global interrupt flag register (CAN_GIF). The interrupt mask bits only affect the content of the global interrupt flag register (CAN_GIF). The interrupt status bits in the global interrupt status register are always set if the corresponding interrupt event occurs, independent of the mask bits. Thus, the interrupt status bits can be used for polling of interrupt events. An interrupt in the global interrupt status register is only asserted if a bit in the CAN_GIF register is set. The global interrupt remains active as long as at least one bit in the interrupt flag register CAN_GIF is set.

CAN Global Interrupt Mask (CAN_GIM) Register

Each source for the global status interrupt (GIRQ) can be enabled or disabled separately with the global interrupt mask register (CAN_GIM). If a bit in the CAN_GIM register is set, the corresponding interrupt source for GIRQ is enabled. The upper bits (15 to 11) are not implemented and always read as 0. After power-up reset or software reset, all bits are cleared, therefore all global status interrupts are disabled.

Figure 19-46. Global Interrupt Mask Register
Controller Area Network (CAN) Module

CAN Global Interrupt Status (CAN_GIS) Register

If a global interrupt event occurs, the corresponding bit in the global interrupt status register (CAN_GIS) is set, independent of the content of the global interrupt mask register (CAN_GIM). If a bit in the CAN_GIS register is cleared and the corresponding interrupt event is still active, this bit is not set again. If a bit in CAN_GIS is cleared, the corresponding bit in CAN_GIF is also cleared (if it was set before).

A bit in the CAN_GIS register can be cleared by writing a 1 to the corresponding bit location. Writing a 0 has no effect. The upper bits (15 to 11) are not implemented and always read as 0. After power-up reset or software reset, all bits are cleared.

Figure 19-47. Global Interrupt Status Register

CAN Global Interrupt Flag (CAN_GIF) Register

If a global interrupt event occurs, the corresponding bit in the global interrupt flag register (CAN_GIF) is set only if the corresponding bit in the global interrupt mask register (CAN_GIM) is set. If a bit in the CAN_GIF register is cleared and the corresponding interrupt event is still active, this bit is not set again.

If at least one bit is set in the global interrupt flag register, the interrupt is active. The interrupt remains active until all bits in CAN_GIF are cleared.
The interrupt flag bits in CAN_GIF can be cleared separately by writing a 1 to the corresponding bit location in the global interrupt status register (CAN_GIS). A write access to CAN_GIF has no effect. The upper bits (15 to 11) are not implemented and always read as 0. After power-up reset or software reset, all bits are cleared.

If an interrupt source is active and the corresponding bit in the CAN_GIF register is still set, this bit remains unchanged. If a bit in the CAN_GIF register is set and then a different bit in the CAN_GIF register is set, the interrupt remains active (only the new bit in CAN_GIF is set). If a bit in the CAN_GIF register is reset and then a different bit in the CAN_GIF register is still set, the interrupt remains active.

If an interrupt status bit in the CAN_GIF register is set and the corresponding interrupt mask bit in the CAN_GIM register is set/reset after the interrupt status bit has been set, the interrupt flag bit in the CAN_GIF register is also set/reset. If no further bit in the CAN_GIF register is set, the interrupt output line (GIRQ) behaves according to the programming of the CAN_GIM register.

Global Interrupt Flag Register (CAN_GIF)

![Figure 19-48. Global Interrupt Flag Register](image-url)
Universal Counter Module

The universal 16-bit counter is operated at the same frequency as the bit clock of the CAN core module. It is clocked using the same parameters used for the bit rate prescaler (BRP, TSEG1 and TSEG2).

The universal counter can be used in several modes, defined by the values in the CAN_UCCNF register. The modes are time stamp mode, watchdog mode, auto transmit mode, and event counter mode.

Time Stamp Mode

To get an indication of the time of reception or the time of transmission for each message, the value of the 16-bit free-running counter (CAN_UCCNT) is written into the CAN_MBxx_TIMESTAMP register of the corresponding mailbox when a received message has been stored or a message has been transmitted.

The time stamp value is captured at the sample point of the start of frame (SOF) bit of each incoming or outgoing message. Afterwards, this time stamp value is copied to the CAN_MBxx_TIMESTAMP register of the corresponding mailbox.

If the mailbox is configured for automatic remote frame handling, the time stamp value is written for transmission of a data frame (mailbox configured as transmit) or the reception of the requested data frame (mailbox configured as receive).

The counter can be cleared or disabled by writing to the CAN_UCCNF register. The counter can also be loaded with a value by writing to the counter register itself (CAN_UCCNT).

It is also possible to clear the counter (CAN_UCCNT) by reception of a message in mailbox number 4 (synchronization of all time stamp counters in the system). This is accomplished by setting the UCCT bit in the CAN_UCCNT register.
Universal Counter Module

An overflow of the counter sets a bit in the global interrupt status register (UCEIS in the CAN_GIS register). A global interrupt can optionally occur by unmasking the bit in the global interrupt mask register (UCEIM in the CAN_GIM register). If the interrupt source is unmasked, a bit in the global interrupt flag register is also set (UCEIF in the CAN_GIF register).

Watchdog Mode

Watchdog mode is used to make sure messages are received periodically. Upon entering watchdog mode, the counter in the CAN_UCCNT register is loaded with a predefined value contained in the CAN universal counter reload/capture register (CAN_UCRC). This counter then decrements at the CAN bit rate. If the UCCT bit in the CAN_UCCNT register is set and a message is received in mailbox 4 before the counter counts down to 0, the counter is reloaded with the CAN_UCRC contents. If the counter has counted down to 0 without receiving a message in mailbox 4, the UCEIS bit in the global interrupt status (CAN_GIS) register is set, and the counter is automatically reloaded with the contents of the CAN_UCRC register. If an interrupt is desired, the UCEIM bit in the CAN_GIM register should be set. With the mask bit set, when a watchdog interrupt occurs, the UCEF bit in the CAN_GIF register is also set.

The counter can be reloaded with the contents of CAN_UCRC or disabled by writing to the CAN_UCCNF register.

The time period it takes for the watchdog interrupt to occur is controlled by the value written into the CAN_UCRC register by the user.
Controller Area Network (CAN) Module

Auto Transmit Mode

In auto transmit mode, the universal counter initiates a cyclic transmission of a CAN message. The counter (CAN_UCCNT) is loaded with the value stored in the CAN_UCRC register. The counter is decremented to 0 and reloaded automatically. When the counter reaches a value of 0, the TRS bit of mailbox 11 is automatically set by internal logic. The corresponding message is sent automatically.

Mailbox 11 must be configured as a transmit mailbox and must contain valid data (identifier, control bits, and data).

Since the TRSn bit for a mailbox is set by internal logic, the corresponding TAn and AAn bits remain unchanged. These bits must be reset by the CPU.

Event Counter Mode

For diagnostic functions, it is possible to use the universal counter as an event counter. The counter can be programmed to increment on one of these conditions:

- CAN error frame counter is incremented if there is an error frame on the CAN bus line.
- CAN overload frame counter is incremented if there is an overload frame on the CAN bus line.
- Arbitration on CAN line lost during transmission.
- Transmission is aborted (AAn is set).
- Successful transmission of message without detected errors (TAn is set).
- Receive message rejected (a message is received without detected errors but not stored in a mailbox because there is no matching identifier found).
Universal Counter Module

- Successful reception of a message without detected errors. The counter is incremented if the received message is rejected or stored in a mailbox.

- Receive message lost (a message is received without detected errors but not stored in a mailbox because this mailbox contains unread data (RMLn is set)).

- Successful reception and matching identifier. A message for one of the configured mailboxes is received. The counter is incremented if the message is stored in the corresponding mailbox (RMPn is set).

- A valid message on the CAN bus line is detected. This may be a reception or a transmission.

**CAN Universal Counter Configuration (CAN_UCCNF) Register**

**Universal Counter Mode Register (CAN_UCCNF)**

![Figure 19-49. Universal Counter Configuration Mode Register](image-url)

Reset = 0x0000
Controller Area Network (CAN) Module

Additional information for the CAN_UCCNF register bits includes:

- **Universal Counter Enable** (*UCE*).

  1—The counter is enabled and incremented/decremented with the programmed clock (bit clock of CAN module).

  0—The counter is disabled.

- **Universal Counter CAN Trigger** (*UCCT*).

  1—In watchdog mode, the counter is reloaded if a message for the mailbox 4 is received. In time stamp mode, the counter is cleared if a message for the mailbox 4 is received (synchronization of all time stamp counters in the system). This bit has no effect in any other mode.

  0—No effect on CAN message reception.

- **Universal Counter Reload/Clear** (*UCRC*).

  1—In watchdog mode, writing a 1 to this bit reloads the counter with the value of the reload/capture register. In time stamp mode, writing a 1 to this bit resets the counter to zero. In all other modes, writing a 1 to this bit resets the counter. Note that this register bit is always read as 0.

  0—Writing a 0 has no effect.

- **Universal Counter Mode** (*UCCNF*).

  0—Reserved.

  1—Time stamp mode.

  The content of the capture register is written to the current mailbox if there was a reception for this mailbox or a successful transmission from this mailbox.
2—Watchdog mode.

The universal counter is reloaded with the value of the reload register CAN_UCRC if there was a valid reception of a message for mailbox number 4 (default).

3—Auto transmit mode.

The universal counter is always reloaded with the value of the reload register if the counter (down counter) has reached the value of 0x0000. On a reload, the transmit request set bit TRS11 (default) is set automatically by the internal logic and the corresponding message in mailbox number 11 is sent.

4—Reserved.

5—Reserved.

6—Event counter mode, increment: counter is incremented if there is an error frame on the CAN bus line.

7—Event counter mode, increment: counter is incremented if there is an overload frame on the CAN bus line.

8—Event counter mode, increment: arbitration on CAN line lost during transmission.

9—Event counter mode, increment: transmission is aborted (AA_n is set).

A—Event counter mode, increment: successful transmission of message without detected errors (TA_n is set)

B—Event counter mode, increment: receive message rejected (a message is received without detected errors but not stored in a mailbox because there is no matching identifier found).
Controller Area Network (CAN) Module

C—Event counter mode, increment: receive message lost (a message is received without detected errors but not stored in a mailbox because this mailbox contains unread data \((RML_n \text{ is set})\)).

D—Event counter mode, increment: successful reception of a message without detected errors. The counter is incremented if the received message is rejected or stored in a mailbox.

E—Event counter mode, increment: successful reception and matching identifier. A message for one of the configured mailboxes is received. The counter is incremented if the message is stored in the corresponding mailbox \((RMP_n \text{ is set})\).

F—Event counter mode, increment: a valid message on the CAN bus line is detected. This may be a reception or a transmission.

**CAN Universal Counter (CAN_UCCNT) Register**

The CAN universal counter \((CAN_{UCCNT})\) register holds the counter value. In time stamp counter mode, this counter is cleared by writing a 1 to the \(UCRC\) bit of the \(CAN_{UCCNF}\) register. In watchdog mode, this counter is cleared by writing 0x0000 to this register.

**Universal Counter Register (CAN_UCCNT)**

![Universal Counter Register](image)

Figure 19-50. Universal Counter Register
Programmable Warning Limit for RXECNT and TXECNT

CAN Universal Counter Reload/Capture (CAN_UCRC) Register

The CAN universal counter reload/capture (CAN_UCRC) register holds a counter value that can be used to reload the CAN_UCCNT register. In time stamp counter mode, a read of this register gives the value of CAN_UCCNT at the time of the last successful reception or transmission. In watchdog mode, a read of this register gives the reload value.

Universal Counter Reload/Capture Register (CAN_UCRC)

![Figure 19-51. Universal Counter Reload/Capture Register](image)

Programmable Warning Limit for RXECNT and TXECNT

It is possible to program the warning level for EWTIS (error warning transmit interrupt status) and EWRIS (error warning receive interrupt status) separately.

After power-up reset, the register is set to the default warning level of 96 for both error counters. After software reset, the content of this register remains unchanged.
CAN Errors and Warnings

CAN errors and warnings are controlled using the CAN_CEC register, the CAN_ESR register, and the CAN_EWR register.

CAN Error Counter (CAN_CEC) Register

The CAN error counter register (CAN_CEC) holds the receive error counter and the transmit error counter. The values of these counters cannot be changed in normal operation mode. The value of CAN_CEC is held during read access. After power-up reset, all bits are cleared. The software reset (SRS bit in CAN_CONTROL) has no effect on this register.

CAN Error Counter Register (CAN_CEC)

![CAN Error Counter Register Diagram]

Figure 19-52. CAN Error Counter Register

The value of CAN_CEC is undefined if the CAN module is in its bus-off mode.

After exiting the bus-off or configuration modes, the CAN error counters are reset. However, the software reset sets the CAN configuration mode request bit (CCR = 1 in the CAN_CONTROL register) and the module changes to the requested mode after all currently performed activities (transmission/reception of a CAN message) are finished.
CAN Errors and Warnings

CAN Error Status (CAN_ESR) Register

The error status register (CAN_ESR) is used to display errors that were generated during operation. Only the first error is stored, any subsequent errors do not change the status of the register. These registers are cleared by writing a 1 to them, except for the SA0 flag, which is cleared by any recessive bit on the bus.

Additional information for the CAN_ESR register bits includes:

- **Form Error Flag (FER).**
  1—A form error occurred on the bus. This means that one or more of the fixed-form bit fields had the wrong level on the bus.
  0—The CAN module was able to send and receive correctly.

- **Bit Error Flag (BEF).**
  1—The received bit does not match the transmitted bit outside of the arbitration field; or, during transmission of the arbitration field, a dominant bit was sent but a recessive bit was received.
  0—The CAN module was able to send and receive correctly.

Figure 19-53. Error Status Register
Controller Area Network (CAN) Module

- **Stuck at Dominant Error** \( (SA0) \).
  
  1—The \( SA0 \) bit is set if the CAN module is in configuration mode or the module enters bus-off mode. The bit is reset if the module samples a recessive bit on the RX input line.
  
  0—The CAN module detected a recessive bit.

- **CRC Error** \( (CRCE) \).
  
  1—The CAN module received an incorrect CRC.
  
  0—The CAN module never received an incorrect CRC.

- **Stuff Error** \( (SER) \).
  
  1—The stuff bit rule was violated.
  
  0—No stuff bit error occurred.

- **Acknowledge Error** \( (ACKE) \).
  
  1—The CAN module received no acknowledge.
  
  0—The CAN module received a correct acknowledge.
CAN Errors and Warnings

CAN Error Counter Warning Level (CAN_EWR) Register

The error warning interrupt bits EWRIS and EWTIS in the CAN_GIS register are set if one of the CAN error counters reaches the default warning level of 96 errors. The error counter warning level register (CAN_EWR) allows the warning level that triggers the EWTIS and EWRIS events to be programmed separately. After power-up reset, the register is set to the default warning level of 96 for both error counters. After software reset, the content of this register remains unchanged.

Additional information for the CAN_EWR register fields includes:

- **Error Warning Level for Receive Error Count** (EWLREC).
  Number of receive errors required before the error warning receive interrupt (EWRIS) is generated.

- **Error Warning Level for Transmit Error Count** (EWLTEC).
  Number of transmit errors required before the error warning transmit interrupt (EWTIS) is generated.
20 TWO-WIRE INTERFACE CONTROLLERS

The two, two-wire interface (TWI) controllers allow a device to interface to an Inter IC bus as specified by the *Philips I2C Bus Specification version 2.1* dated January 2000.

Overview

Each TWI is fully compatible with the widely used I²C bus standard. They are designed with a high level of functionality and are compatible with multimaster, multislave bus configurations. To preserve processor bandwidth the TWI controllers can be set up and a transfer initiated with interrupts only to service FIFO buffer data reads and writes. Protocol related interrupts are optional.

Each TWI externally moves 8-bit data while maintaining compliance with the I²C bus protocol. The *Philips I2C Bus Specification version 2.1* covers many variants of I²C. The TWI controllers include these features:

- Simultaneous master and slave operation on multiple device systems
- Support for multimaster data arbitration
- 7 bit addressing
- 100K bytes/second and 400K bytes/second data rates
- General call address support
Architecture

- Master clock synchronization and support for clock low extension
- Separate multiple-byte receive and transmit FIFOs
- Low interrupt rate
- Individual override control of data and clock lines in the event of bus lock-up
- Input filter for spike suppression
- Serial camera control bus support as specified in *OmniVision Serial Camera Control Bus (SCCB) Functional Specification version 2.1.*

Table 20-1 shows the pins for the TWIs. Two bidirectional pins externally interface each TWI controller to the I²C bus. The interface is simple and no other external connections or logic are required.

Table 20-1. TWI Pins

<table>
<thead>
<tr>
<th>Pin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDAx</td>
<td>In/Out TWI serial data, high impedance reset value</td>
</tr>
<tr>
<td>SCLx</td>
<td>In/Out TWI serial clock, high impedance reset value</td>
</tr>
</tbody>
</table>

Architecture

*Figure 20-1* illustrates the overall architecture of the TWI controllers.

The peripheral interface supports the transfer of 16-bit wide data and is used by the processor in the support of register and FIFO buffer reads and writes.

The register block contains all control and status bits and reflects what can be written or read as outlined by the programmer’s model. Status bits can be updated by their respective functional blocks.
The FIFO buffer is configured as a 1-byte-wide 2-deep transmit FIFO buffer and a 1-byte-wide 2-deep receive FIFO buffer.

The transmit shift register serially shifts its data out externally off chip. The output can be controlled for generation of acknowledgements or it can be manually overwritten.

The receive shift register receives its data serially from off chip. The receive shift register is 1 byte wide and data received can either be transferred to the FIFO buffer or used in an address comparison.

The address compare block supports address comparison in the event a TWI controller module is accessed as a slave.
The prescaler block must be programmed to generate a 10 MHz time reference relative to the system clock. This time base is used for filtering of data and timing events specified by the electrical data sheet (See the Philips Specification), as well as for SCL clock generation.

The clock generation module is used to generate an external SCL clock when in master mode. It includes the logic necessary for synchronization in a multimaster clock configuration and clock stretching when configured in slave mode.

Register Descriptions

Each TWI controller has 16 registers described in the following sections.

TWI Control (TWIx_CONTROL) Registers

The TWI control register (TWIx_CONTROL), shown in Figure 20-2, is used to enable the TWI module as well as to establish a relationship between the system clock (SCLK) and the TWI controller’s internally timed events. The internal time reference is derived from SCLK using a prescaled value.

Figure 20-2. TWI Control Register
Additional information for the `TWIx_CONTROL` register bits are:

- **SCCB Compatibility** (`SCCB`).
  
  SCCB compatibility is an optional feature and should not be used in an I2C bus system. This feature is valid only during transfers where the TWI is mastering an SCCB bus. Slave mode transfers should be avoided when this feature is enabled because the TWI controller always generates an acknowledge in slave mode.

  [1] Master transfers are SCCB compatible. All slave asserted acknowledgement bits are ignored by this master.

  [0] Master transfers are not SCCB compatible.

- **TWI Enable** (`TWIx_ENA`).
  
  This bit must be set for slave mode or master mode operation. It is recommended that this bit be set at the time `PRESCALE` is initialized and remain set. This guarantees accurate operation of bus busy detection logic.

  [1] Internal time reference is enabled. Slave mode and master mode operation is reliable.

  [0] Internal time reference is disabled.

- **Prescale** (`PRESCALE`).
  
  The number of system clock (SCLK) periods used in the generation of one internal time reference. The value of `PRESCALE` must be set to create an internal time reference with a period of 10 MHz. Represented as a 7-bit binary value.
It is not always possible to achieve 10 MHz accuracy. In such cases, it is safe to round up the PRESCALE value to the next highest integer. For example, if SCLK is 133 MHz, the PRESCALE value is calculated as 133 MHz/10 MHz = 13.3. In this case, a PRESCALE value of 14 ensures that all timing requirements are met.

TWI Clock Divider (TWIx_CLKDIV) Registers

During master mode operation, the SCL clock divider register (TWIx_CLKDIV) values are used to create the high and low durations of the serial clock (SCL). (See Figure 20-3.) Serial clock frequencies can vary from 400 KHz to less than 20 KHz. The resolution of the clock generated is 1/10 MHz or 100 ns.

\[
CLKDIV = \text{TWIx SCL period/10 MHz time reference}
\]

For example, for an SCL of 400 KHz (period = 1/400 KHz = 2500 ns) and an internal time reference of 10 MHz (period = 100 ns):

\[
CLKDIV = 2500 \text{ ns} / 100 \text{ ns} = 25
\]

For an SCL with a 30% duty cycle, then CLKLOW = 17 and CLKHI = 8.

Note that CLKLOW and CLKHI add up to CLKDIV.

SCL Clock Divider Register (TWIx_CLKDIV)

![SCL Clock Divider Register](image)

Figure 20-3. SCL Clock Divider Registers
Two-Wire Interface Controllers

Additional information for the TWIx_CLKDIV register bits includes:

- **Clock High (CLKHI).**
  Number of 10 MHz time reference periods the serial clock (SCL) waits before a new clock low period begins, assuming a single master. Represented as an 8-bit binary value.

- **Clock Low (CLKLO).**
  Number of internal time reference periods the serial clock (SCL) is held low. Represented as an 8-bit binary value.

**TWI Slave Mode Control (TWIx_SLAVE_CTRL) Registers**

The TWI slave mode control registers (TWIx_SLAVE_CTRL), as shown in Figure 20-4, control the logic associated with slave mode operation. Settings in these registers do not affect master mode operation and should not be modified to control master mode functionality.

![TWI Slave Mode Control Registers (TWIx_SLAVE_CTRL)](image)

Figure 20-4. TWI Slave Mode Control Registers
Additional information for the `TWIx_SLAVE_CTRL` register bits includes:

- **General Call Enable** (\texttt{GEN}).
  
  General call address detection is available only when slave mode is enabled.

  [1] General call address matching is enabled. A general call slave receive transfer is accepted. All status and interrupt source bits associated with transfers are updated.

  [0] General call address matching is not enabled.

- **NAK** (\texttt{NAK}).
  
  [1] Slave receive transfers generate a data NAK (not acknowledge) at the conclusion of a data transfer. The slave is still considered to be addressed.

  [0] Slave receive transfers generate an ACK at the conclusion of a data transfer.

- **Slave Transmit Data Valid** (\texttt{STDVAL}).
  
  [1] Data in the transmit FIFO is available for a slave transmission.

  [0] Data in the transmit FIFO is for master mode transmits and is not allowed to be used during a slave transmit, and the transmit FIFO is treated as if it is empty.

- **Slave Enable** (\texttt{SEN}).
  
  [1] The slave is enabled. Enabling slave and master modes of operation concurrently is allowed.

  [0] The slave is not enabled. No attempt is made to identify a valid address. If cleared during a valid transfer, clock stretching ceases, the serial data line is released, and the current byte is not acknowledged.
TWI Slave Mode Address (TWIx_SLAVE_ADDR) Registers

The TWI slave mode address registers (TWIx_SLAVE_ADDR), shown in Figure 20-5, hold the slave mode address, which is the valid address that the slave-enabled TWI controller responds to. The TWI controller compares this value with the received address during the addressing phase of a transfer.

**TWI Slave Mode Address Registers (TWIx_SLAVE_ADDR)**

![Figure 20-5. TWI Slave Mode Address Registers](image-url)

**TWI Slave Mode Status (TWIx_SLAVE_STAT) Registers**

During and at the conclusion of slave mode transfers, the TWI slave mode status registers (TWIx_SLAVE_STAT) hold information on the current transfer. (See Figure 20-6.) Generally slave mode status bits are not associated with the generation of interrupts. Master mode operation does not affect slave mode status bits.

**TWI Slave Mode Status Registers (TWIx_SLAVE_STAT)**

![Figure 20-6. TWI Slave Mode Status Registers](image-url)
Additional information for the TWIx_SLAVE_STAT register bits includes:

- **General Call (GCALL).**
  
  This bit self clears if slave mode is disabled (SEN = 0).

  [1] At the time of addressing, the address was determined to be a general call.

  [0] At the time of addressing, the address was not determined to be a general call.

- **Slave Transfer Direction (SDIR).**
  
  This bit self clears if slave mode is disabled (SEN = 0).

  [1] At the time of addressing, the transfer direction was determined to be slave transmit.

  [0] At the time of addressing, the transfer direction was determined to be slave receive.
TWI Master Mode Control (TWIx_MASTER_CTRL) Registers

The TWI master mode control registers (TWIx_MASTER_CTRL) control the logic associated with master mode operation. (See Figure 20-7.) Bits in these registers do not affect slave mode operation and should not be modified to control slave mode functionality.

TWI Master Mode Control Registers (TWIx_MASTER_CTRL)

![Diagram of TWI Master Mode Control Registers]

Additional information for the TWIx_MASTER_CTRL register bits includes:

- **Serial Clock Override (SCLOVR).**

  This bit can be used when direct control of the serial clock line is required. Normal master and slave mode operation should not require override operation.

- [1] Serial clock output is driven to an active 0 level overriding all other logic. This state is held until this bit is cleared.

- [0] Normal serial clock operation under the control of master mode clock generation and slave mode clock stretching logic.
Register Descriptions

- **Serial Data (SDA) Override (SDAovr).**

  This bit can be used when direct control of the serial data line is required. Normal master and slave mode operation should not require override operation.

  [1] Serial data output is driven to an active 0 level overriding all other logic. This state is held until this bit is cleared.

  [0] Normal serial data operation under the control of the transmit shift register and acknowledge logic.

- **Data Transfer Count (DCNT).**

  Indicates the number of data bytes to transfer. As each data word is transferred, DCNT is decremented. When DCNT is 0, a stop condition is generated. Setting DCNT to 0xFF disables the counter. In this transfer mode, data continues to be transferred until it is concluded by setting the STOP bit.

- **Repeat Start (RSTART).**

  [1] Issue a repeat start condition at the conclusion of the current transfer (DCNT = 0) and begin the next transfer. The current transfer concludes with updates to the appropriate status and interrupt bits. If errors occurred during the previous transfer, a repeat start does not occur. In the absence of any errors, master enable (MEN) does not self clear on a repeat start.

  [0] Transfer concludes with a stop condition.
• **Issue Stop Condition** *(STOP).*

[1] The transfer concludes as soon as possible avoiding any error conditions (as if data transfer count had been reached) and at that time the TWI interrupt status registers *(TWIx_INT_STAT)* are updated.

[0] Normal transfer operation.

• **Fast Mode** *(FAST).*

[1] Fast mode (up to 400 Kbits/s) timing specifications in use.

[0] Standard mode (up to 100K bits/s) timing specifications in use.

• **Master Transfer Direction** *(MDIR).*

[1] The initiated transfer is master receive.

[0] The initiated transfer is master transmit.

• **Master Mode Enable** *(MEN).*

This bit self clears at the completion of a transfer (after the *DCNT* bit decrements to zero), including transfers terminated due to errors.

[1] Master mode functionality is enabled. A start condition is generated if the bus is idle.

[0] Master mode functionality is disabled. If this bit is cleared during operation, the transfer is aborted and all logic associated with master mode transfers are reset. Serial data and serial clock *(SDA, SCL)* are no longer driven. Write-1-to-clear status bits are not affected.
Register Descriptions

**TWI Master Mode Address (TWIx_MASTER_ADDR) Registers**

During the addressing phase of a transfer, the TWI controller, with its master enabled, transmits the contents of the TWI master mode address registers (TWIx_MASTER_ADDR), shown in Figure 20-8. When programming these registers, omit the read/write bit. That is, only the upper 7 bits that make up the slave address should be written to these registers. For example, if the slave address is b#1010000X, where X is the read/write bit, then TWIx_MASTER_ADDR is programmed with b#1010000, which corresponds to 0x50. When sending out the address on the bus, the TWI controller appends the read/write bit as appropriate based on the state of the MDIR bit in the master mode control register.

![Figure 20-8. TWI Master Mode Address Registers](image)

**TWI Master Mode Status (TWIx_MASTER_STAT) Registers**

The TWI master mode status registers (TWIx_MASTER_STAT), shown in Figure 20-9, hold information during master mode transfers and at their conclusion. Generally, master mode status bits are not directly associated with the generation of interrupts but offer information on the current transfer. Slave mode operation does not affect master mode status bits.
Two-Wire Interface Controllers

TWI Master Mode Status Registers (TWIx_MASTER_STAT)

![TWI Master Mode Status Registers Diagram]

Figure 20-9. TWI Master Mode Status Registers

Additional information for the TWIx_MASTER_STAT registers bits includes:

- **Bus Busy (BUSBUSY).**

  Indicates whether the bus is currently busy or free. This indication is for all devices not this device alone. Upon a start condition, the setting of the register value is delayed due to the input filtering. Upon a stop condition the clearing of the register value occurs after \( t_{BUF} \).

  [1] The bus is busy. Clock or data activity has been detected.

  [0] The bus is free. The clock and data bus signals have been inactive for the appropriate bus free time.
Register Descriptions

- **Serial Clock Sense** *(SCLSEN)*.

  This status bit can be used when direct sensing of the serial clock line is required. The register value is delayed due to the input filter (nominally 50 ns). Normal master and slave mode operation should not require this feature.

  [1] An active “zero” is currently being sensed on the serial clock. The source of the active driver is not known and can be internal or external.

  [0] An inactive “one” is currently being sensed on the serial clock.

- **Serial Data Sense** *(SDASEN)*.

  This status bit can be used when direct sensing of the serial data line is required. The register value is delayed due to the input filter (nominally 50 ns). Normal master and slave mode operation should not require this feature.

  [1] An active “zero” is currently being sensed on the serial data line. The source of the active driver is not known and can be internal or external.

  [0] An inactive “one” is currently being sensed on the serial data line.

- **Buffer Write Error** *(BUFWRERR)*.

  [1] The current master transfer was aborted due to a receive buffer write error. The receive buffer and receive shift register were both full at the same time. This bit is "write-1-to-clear".

  [0] The current master receive has not detected a receive buffer write error.
• **Buffer Read Error** (BUFRDERR).

[1] The current master transfer was aborted due to a transmit buffer read error. At the time data was required by the transmit shift register the buffer was empty. This bit is "write-1-to-clear".

[0] The current master transmit has not detected a buffer read error.

• **Data Not Acknowledged** (DNAK).

[1] The current master transfer was aborted due to the detection of a NAK during data transmission. This bit is "write-1-to-clear".

[0] The current master receive has not detected a NAK during data transmission.

• **Address Not Acknowledged** (ANAK).

[1] The current master transfer was aborted due to the detection of a NAK during the address phase of the transfer. This bit is "write-1-to-clear".

[0] The current master transmit has not detected NAK during addressing.

• **Lost Arbitration** (LOSTARB).

[1] The current transfer was aborted due to the loss of arbitration with another master. This bit is "write-1-to-clear".

[0] The current transfer has not lost arbitration with another master.
Register Descriptions

- Master Transfer in Progress (MPROG).

[1] A master transfer is in progress.

[0] Currently no transfer is taking place. This can occur once a transfer is complete or while an enabled master is waiting for an idle bus.

TWI FIFO Control (TWIx_FIFO_CTRL) Registers

The TWI FIFO control register (TWIx_FIFO_CTRL) control bits affect only the FIFO and are not tied in any way with master or slave mode operation. See Figure 20-10.

TWI FIFO Control Registers (TWIx_FIFO_CTRL)

![TWI FIFO Control Registers Diagram](image-url)

Figure 20-10. TWI FIFO Control Registers
Two-Wire Interface Controllers

Additional information for the \texttt{TWIx\_FIFO\_CTRL} register bits includes:

- **Receive Buffer Interrupt Length** (\texttt{RCVINTLEN}).

  This bit determines the rate at which receive buffer interrupts are to be generated. Interrupts may be generated with each byte received or after two bytes are received.

  [0] An interrupt (\texttt{RCVSERV}) is set when \texttt{RCVSTAT} indicates one or two bytes in the FIFO are full (01 or 11).

  [1] An interrupt (\texttt{RCVSERV}) is set when \texttt{RCVSTAT} indicates two bytes in the FIFO are full (11).

- **Transmit Buffer Interrupt Length** (\texttt{XMTINTLEN}).

  This bit determines the rate at which transmit buffer interrupts are to be generated. Interrupts may be generated with each byte transmitted or after two bytes are transmitted.

  [0] An interrupt (\texttt{XMTSERV}) is set when \texttt{XMTSTAT} indicates one or two bytes in the FIFO are empty (00).

  [1] An interrupt (\texttt{XMTSERV}) is set when \texttt{XMTSTAT} indicates two bytes in the FIFO are empty (00).

- **Receive Buffer Flush** (\texttt{RCVFLUSH}).

  [0] Normal operation of the receive buffer and its status bits.

  [1] Flush the contents of the receive buffer and update the \texttt{RCVSTAT} status bit to indicate the buffer is empty. This state is held until this bit is cleared. During an active receive the receive buffer in this state responds to the receive logic as if it is full.
Register Descriptions

- **Transmit Buffer Flush** (*XMTFLUSH*).

  [1] Flush the contents of the transmit buffer and update the *XMTSTAT* status bit to indicate the buffer is empty. This state is held until this bit is cleared. During an active transmit the transmit buffer in this state responds to the as if the transmit buffer is empty.

  [0] Normal operation of the transmit buffer and its status bits.

**TWI FIFO Status (TWIx_FIFO_STAT) Registers**

The fields in the TWI FIFO status registers (*TWIx_FIFO_STAT*), shown in Figure 20-11, indicate the state of the FIFO buffers’ receive and transmit contents. The FIFO buffers do not discriminate between master data and slave data. By using the status and control bits provided, the FIFO can be managed to allow simultaneous master and slave operation.

**TWI FIFO Status Registers (TWIx_FIFO_STAT)**

All bits are RO.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Reset Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCVSTAT[1:0]</td>
<td>Receive FIFO status</td>
<td>0xFFC0</td>
</tr>
<tr>
<td>XMTSTAT[1:0]</td>
<td>Transmit FIFO status</td>
<td>0x0000</td>
</tr>
</tbody>
</table>

Figure 20-11. TWI FIFO Status Registers
Two-Wire Interface Controllers

Additional information for the $\text{TWIx_FIFO_STAT}$ register bits includes:

- **Receive FIFO Status** ($\text{RCVSTAT}$).
  
The $\text{RCVSTAT}$ field is read only. It indicates the number of valid data bytes in the receive FIFO buffer. The status is updated with each FIFO buffer read using the peripheral data bus or write access by the receive shift register. Simultaneous accesses are allowed.

  [11] The FIFO is full and contains two bytes of data. Either a single or double byte peripheral read of the FIFO is allowed.


  [01] The FIFO contains one byte of data. A single byte peripheral read of the FIFO is allowed.

  [00] The FIFO is empty.

- **Transmit FIFO Status** ($\text{XMTSTAT}$).
  
The $\text{XMTSTAT}$ field is read only. It indicates the number of valid data bytes in the FIFO buffer. The status is updated with each FIFO buffer write using the peripheral data bus or read access by the transmit shift register. Simultaneous accesses are allowed.

  [11] The FIFO is full and contains two bytes of data.


  [01] The FIFO contains one byte of data. A single byte peripheral write of the FIFO is allowed.

  [00] The FIFO is empty. Either a single or double byte peripheral write of the FIFO is allowed.
TWI Interrupt Mask (TWIx_INT_ENABLE) Registers

The TWI interrupt mask registers (TWIx_INT_ENABLE), shown in Figure 20-12, enable interrupt sources to assert the interrupt output. Each mask bit corresponds with one interrupt source bit in the TWI interrupt status (TWIx_INT_STAT) registers. Reading and writing the TWI interrupt mask registers does not affect the contents of the TWI interrupt status registers.

### TWI Interrupt Enable Registers (TWIx_INT_ENABLE)

For all bits, 0 = interrupt generation disabled, 1 = interrupt generation enabled.

![TWI Interrupt Mask Register Diagram](image)

Figure 20-12. TWI Interrupt Mask Registers

Additional information for the TWIx_INT_ENABLE register bits includes:

- **Receive FIFO Service (RCVSERV).**

  If RCVINTLEN in the TWIx_FIFO_CTRL registers is 0, this bit is set each time the RCVSTAT field in the TWIx_FIFO_STAT registers is updated to either 01 or 11. If RCVINTLEN is 1, this bit is set each time RCVSTAT is updated to either 10 or 11.

  [0] The corresponding interrupt source is prevented from asserting the interrupt output.

  [1] Contents of “one” in the corresponding interrupt source will result in asserting the interrupt output.
• **Transmit FIFO Service** (**XMTSERV**).

If **XMTINTLEN** in the **TWIX_FIFO_CTRL** registers is 0, this bit is set each time the **XMTSTAT** field in the **TWIX_FIFO_STAT** registers is updated to either 01 or 00. If **XMTINTLEN** is 1, this bit is set each time **XMTSTAT** is updated to 00.

• **Master Transfer Error** (**MERR**).

[0] The corresponding interrupt source is prevented from asserting the interrupt output.

[1] Contents of “one” in the corresponding interrupt source will result in asserting the interrupt output.

• **Master Transfer Complete** (**MCOMP**).

[0] The corresponding interrupt source is prevented from asserting the interrupt output.

[1] Contents of “one” in the corresponding interrupt source will result in asserting the interrupt output.

• **Slave Overflow** (**SOVF**).

[0] The corresponding interrupt source is prevented from asserting the interrupt output.

[1] Contents of “one” in the corresponding interrupt source will result in asserting the interrupt output.

• **Slave Transfer Error** (**SERR**).

[0] The corresponding interrupt source is prevented from asserting the interrupt output.

[1] Contents of “one” in the corresponding interrupt source will result in asserting the interrupt output.
Register Descriptions

- **Slave Transfer Complete** (*SCOMP*).
  
  [0] The corresponding interrupt source is prevented from asserting the interrupt output.
  
  [1] Contents of “one” in the corresponding interrupt source will result in asserting the interrupt output.

- **Slave Transfer Initiated** (*SINIT*).
  
  [0] The corresponding interrupt source is prevented from asserting the interrupt output.
  
  [1] Contents of “one” in the corresponding interrupt source will result in asserting the interrupt output.

**TWI Interrupt Status (TWIx_INT_STAT) Registers**

The TWI interrupt status registers (*TWIx_INT_STAT*), shown in Figure 20-13, contain information about functional areas requiring servicing. Many of the bits serve as an indicator to further read and service various status registers. After servicing the interrupt source associated with a bit, the user must clear that interrupt source bit.

**TWI Interrupt Status Registers (TWIx_INT_STAT)**

All bits are sticky and "write 1 to clear".

---

**Figure 20-13. TWI Interrupt Status Registers**
Two-Wire Interface Controllers

Additional information for the \texttt{TWIx\_INT\_STAT} register bits includes:

- **Receive FIFO Service** (\texttt{RCVSERV}).

  If \texttt{RCVINTLEN} in the \texttt{TWIx\_FIFO\_CTRL} registers is 0, this bit is set each time the \texttt{RCVSTAT} field in the \texttt{TWIx\_FIFO\_STAT} registers is updated to either 01 or 11. If \texttt{RCVINTLEN} is 1, this bit is set each time \texttt{RCVSTAT} is updated to either 10 or 11.

  [1] The receive FIFO buffer has one or two 8-bit locations containing data to be read.

  [0] FIFO does not require servicing or \texttt{RCVSTAT} field has not changed since this bit was last cleared.

- **Transmit FIFO Service** (\texttt{XMTSERV}).

  If \texttt{XMTINTLEN} in the \texttt{TWIx\_FIFO\_CTRL} registers is 0, this bit is set each time the \texttt{XMTSTAT} field in the \texttt{TWIx\_FIFO\_STAT} registers is updated to either 01 or 00. If \texttt{XMTINTLEN} is 1, this bit is set each time \texttt{XMTSTAT} is updated to 00.

  [1] The transmit FIFO buffer has one or two 8-bit locations available to be written.

  [0] FIFO does not require servicing or \texttt{XMTSTAT} field has not changed since this bit was last cleared.

- **Master Transfer Error** (\texttt{MERR}).

  [1] A master error has occurred. The conditions surrounding the error are indicated by the master status registers (\texttt{TWIx\_MASTER\_STAT}).

  [0] No errors have been detected.
Register Descriptions

- **Master Transfer Complete** (MCOMP).
  
  [1] The initiated master transfer has completed. In the absence of a repeat start, the bus has been released.
  
  [0] The completion of a transfer has not been detected.

- **Slave Overflow** (SOVF).
  
  [1] The slave transfer complete (SCOMP) bit was set at the time a subsequent transfer has acknowledged an address phase. The transfer continues, however, it may be difficult to delineate data of one transfer from another.
  
  [0] No overflow has been detected.

- **Slave Transfer Error** (SERR).
  
  [1] A slave error has occurred. A restart or stop condition has occurred during the data receive phase of a transfer.
  
  [0] No errors have been detected.

- **Slave Transfer Complete** (SCOMP).
  
  [1] The transfer is complete and either a stop, or a restart was detected.
  
  [0] The completion of a transfer has not been detected.

- **Slave Transfer Initiated** (SINIT).
  
  [1] The slave has detected an address match and a transfer has been initiated.
  
  [0] A transfer is not in progress. An address match has not occurred since the last time this bit was cleared.
TWI FIFO Transmit Data Single Byte (TWIx_XMT_DATA8) Registers

The TWI FIFO transmit data single byte registers (TWIx_XMT_DATA8), shown in Figure 20-14, hold an 8-bit data value written into the FIFO buffer.

Transmit data is entered into the corresponding transmit buffer in a first-in first-out order. Although peripheral bus writes are 16 bits, a write access to TWIx_XMT_DATA8 adds only one transmit data byte to the FIFO buffer. With each access, the transmit status (XMTSTAT) field in the TWIx_FIFO_STAT registers is updated. If an access is performed while the FIFO buffer is full, the write is ignored and the existing FIFO buffer data and its status remains unchanged.

TWI FIFO Transmit Data Single Byte Registers (TWIx_XMT_DATA8)
All bits are WO. This register always reads as 0x0000.

```
0xFFC0 1480
0xFFC0 2280
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Figure 20-14. TWI FIFO Transmit Data Single Byte Registers

TWI FIFO Transmit Data Double Byte (TWIx_XMT_DATA16) Registers

The TWI FIFO transmit data double byte registers (TWIx_XMT_DATA16), shown in Figure 20-15, hold a 16-bit data value written into the FIFO buffer.

To reduce interrupt output rates and peripheral bus access times, a double byte transfer data access can be performed. Two data bytes can be written, effectively filling the transmit FIFO buffer with a single access.
The data is written in little endian byte order as shown in Figure 20-16 where byte 0 is the first byte to be transferred and byte 1 is the second byte to be transferred. With each access, the transmit status (XMTSTAT) field in the TWIx_FIFO_STAT registers is updated. If an access is performed while the FIFO buffer is not empty, the write is ignored and the existing FIFO buffer data and its status remains unchanged.

**TWI FIFO Transmit Data Double Byte Registers (TWIx_XMT_DATA16)**

All bits are WO. This register always reads as 0x0000.

![Figure 20-15. TWI FIFO Transmit Data Double Byte Registers](image)

**Figure 20-15. TWI FIFO Transmit Data Double Byte Registers**

**Data In Register**

![Data In Register](image)

**Figure 20-16. Little Endian Byte Order**

**TWI FIFO Receive Data Single Byte (TWIx_RCV_DATA8) Registers**

The TWI FIFO receive data single byte registers (TWIx_RCV_DATA8), shown in Figure 20-17, hold an 8-bit data value read from the FIFO buffer. Receive data is read from the corresponding receive buffer in a first-in first-out order. Although peripheral bus reads are 16 bits, a read access to TWIx_RCV_DATA8 will access only one transmit data byte from the FIFO buffer. With each access, the receive status (RCVSTAT) field in the TWIx_FIFO_STAT registers is updated. If an access is performed while the FIFO buffer is empty, the data is unknown and the FIFO buffer status remains indicating it is empty.
The TWI FIFO receive data double byte registers (TWIx_RCV_DATA16), shown in Figure 20-18, hold a 16-bit data value read from the FIFO buffer. To reduce interrupt output rates and peripheral bus access times, a double byte receive data access can be performed. Two data bytes can be read, effectively emptying the receive FIFO buffer with a single access. The data is read in little endian byte order as shown in Figure 20-19 where byte 0 is the first byte received and byte 1 is the second byte received. With each access, the receive status (RCVSTAT) field in the TWIx_FIFO_STAT registers is updated to indicate it is empty. If an access is performed while the FIFO buffer is not full, the read data is unknown and the existing FIFO buffer data and its status remains unchanged.
Data Transfer Mechanics

The TWI controllers follow the transfer protocol of the *Philips I2C Bus Specification version 2.1* dated January 2000. A simple complete transfer is shown in Figure 20-20.

<table>
<thead>
<tr>
<th>S</th>
<th>7-BIT ADDRESS</th>
<th>R/W</th>
<th>ACK</th>
<th>8-BIT DATA</th>
<th>ACK</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>S = START</td>
<td>P = STOP</td>
<td>ACK = ACKNOWLEDGE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 20-20. Basic Data Transfer

To better understand the mapping of the TWI controller register contents to a basic transfer, Figure 20-21 details the same transfer as above noting the corresponding TWI controller bit names. In this illustration, the TWI controller successfully transmits one byte of data. The slave has acknowledged both address and data.

<table>
<thead>
<tr>
<th>S</th>
<th>MADDR[6:0]</th>
<th>MDIR</th>
<th>ACK</th>
<th>XMITDATA[7:0]</th>
<th>ACK</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>S = START</td>
<td>P = STOP</td>
<td>ACK = ACKNOWLEDGE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 20-21. Data Transfer With Bit Illustration
Clock Generation and Synchronization

The TWI controller implementation only issues a clock during master mode operation and only at the time a transfer has been initiated. If arbitration for the bus is lost, the serial clock output immediately three-states. If multiple clocks attempt to drive the serial clock line, the TWI controller synchronizes its clock with the other remaining clocks. This is shown in Figure 20-22.

![TWI Clock Synchronization Diagram](Figure 20-22. TWI Clock Synchronization)

The TWI controller’s serial clock (SCL) output follows these rules:

- Once the clock high (CLKHI) count is complete, the serial clock output is driven low and the clock low (CLKLOW) count begins.

- Once the clock low count is complete, the serial clock line is three-stated and the clock synchronization logic enters into a delay mode (shaded area) until the SCL line is detected at a logic 1 level. At this time the clock high count begins.
Bus Arbitration

The TWI controllers initiate a master mode transmission (MEN) only when the bus is idle. If the bus is idle and two masters initiate a transfer, arbitration for the bus begins. This is shown in Figure 20-23.

![Figure 20-23. TWI Bus Arbitration](image)

The TWI controllers monitor the serial data bus (SDA) while SCL is high and if SDA is determined to be an active logic 0 level while the TWI controller’s data is a logic 1 level, the TWI controller has lost arbitration and ends generation of clock and data. Note arbitration is not performed only at serial clock edges, but also during the entire time SCL is high.

Start and Stop Conditions

Start and stop conditions involve serial data transitions while the serial clock is a logic 1 level. The TWI controllers generate and recognize these transitions. Typically start and stop conditions occur at the beginning and at the conclusion of a transmission with the exception repeated start “combined” transfers, as shown in Figure 20-24.
The TWI controller’s special case start and stop conditions include:

- **TWI controller addressed as a slave-receiver**
  
  If the master asserts a stop condition during the data phase of a transfer, the TWI controller concludes the transfer (SCOMP).

- **TWI controller addressed as a slave-transmitter**
  
  If the master asserts a stop condition during the data phase of a transfer, the TWI controller concludes the transfer (SCOMP) and indicates a slave transfer error (SERR).

- **TWI controller as a master-transmitter or master-receiver**
  
  If the stop bit is set during an active master transfer, the TWI controller issues a stop condition as soon as possible avoiding any error conditions (as if data transfer count had been reached).

### General Call Support

The TWI controllers always decode and acknowledge a general call address if it is enabled as a slave (SEN) and if general call is enabled (GEN). General call addressing (0x00) is indicated by the GCALL bit being set and by nature of the transfer the TWI controller is a slave-receiver. If the data associated with the transfer is to be NAK’ed, the NAK bit can be set.
Programming Examples

If the TWI controller is to issue a general call as a master-transmitter the appropriate address and transfer direction can be set along with loading transmit FIFO data.

The byte following the general call address usually defines what action needs to be taken by the slaves in response to the call. The command in the second byte is interpreted based on the value of its LSB. For a TWI slave device, this is not applicable, and the bytes received after the general call address are considered data.

Fast Mode

Fast mode essentially uses the same mechanics as standard mode of operation. It is the electrical specifications and timing that are most effected. When fast mode is enabled (FAST) the following timings are modified to meet the electrical requirements.

- Serial data rise times before arbitration evaluation ($t_r$)
- Stop condition set-up time from serial clock to serial data ($t_{SU;STO}$)
- Bus free time between a stop and start condition ($t_{BUF}$)

Programming Examples

The following sections include programming examples for general setup, slave mode, and master mode, as well as guidance for repeated start conditions.
General Setup

General setup refers to register writes that are required for both slave mode operation and master mode operation. General setup should be performed before either the master or slave enable bits are set.

Program the \texttt{TWIx\_CONTROL} registers to enable the TWI controller and set the prescale value. Program the prescale value to the binary representation of \( f_{SCLK} / 10 \text{ MHz} \)

All values should be rounded up to the next whole number. The \texttt{TWIx\_ENA} bit enable must be set. Note once the TWI controller is enabled a bus busy condition may be detected. This condition should clear after \( t_{BUF} \) has expired assuming additional bus activity has not been detected.

Slave Mode

When enabled, slave mode operation supports both receive and transmit data transfers. It is not possible to enable only one data transfer direction and not acknowledge (NAK) the other. This is reflected in the following setup.

1. Program \texttt{TWIx\_SLAVE\_ADDR}. The appropriate 7 bits are used in determining a match during the address phase of the transfer.

2. Program \texttt{TWIx\_XMT\_DATA8} or \texttt{TWIx\_XMT\_DATA16}. These are the initial data values to be transmitted in the event the slave is addressed and a transmit is required. This is an optional step. If no data is written and the slave is addressed and a transmit is required, the serial clock (\texttt{SCL}) is stretched and an interrupt is generated until data is written to the transmit FIFO.
3. Program TWIx_INT_ENABLE. Enable bits are associated with the desired interrupt sources. As an example, programming the value 0x000F results in an interrupt output to the processor in the event that a valid address match is detected, a valid slave transfer completes, a slave transfer has an error, a subsequent transfer has begun yet the previous transfer has not been serviced.

4. Program TWIx_SLAVE_CTRL. Ultimately this prepares and enables slave mode operation. As an example, programming the value 0x0005 enables slave mode operation, requires 7-bit addressing, and indicates that data in the transmit FIFO buffer is intended for slave mode transmission.

Table 20-2 shows what the interaction between a TWI controller and the processor might look like using this example.

Table 20-2. Slave Mode Setup Interaction

<table>
<thead>
<tr>
<th>TWI Controller Master</th>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Master Mode Clock Setup**

Master mode operation is set up and executed on a per-transfer basis. An example of programming steps for a receive and for a transmit are given separately in following sections. The clock setup programming step listed here is common to both transfer types.

Program TWIx_CLKDIV. This defines the clock high duration and clock low duration.
Master Mode Transmit

Follow these programming steps for a single master mode transmit:

1. Program TWIx_MASTER_ADDR. This defines the address transmitted during the address phase of the transfer.

2. Program TWIx_XMT_DATA8 or TWIx_XMT_DATA16. This is the initial data transmitted. It is considered an error to complete the address phase of the transfer and not have data available in the transmit FIFO buffer.

3. Program TWIx_FIFO_CTRL. Indicate if transmit FIFO buffer interrupts should occur with each byte transmitted (8 bits) or with each 2 bytes transmitted (16 bits).

4. Program TWIx_INT_ENABLE. Enable bits associated with the desired interrupt sources. As an example, programming the value 0x0030 results in an interrupt output to the processor in the event that the master transfer completes, and the master transfer has an error.

5. Program TWIx_MASTER_CTRL. Ultimately this prepares and enables master mode operation. As an example, programming the value 0x0201 enables master mode operation, generates a 7-bit address, sets the direction to master-transmit, uses standard mode timing, and transmits 8 data bytes before generating a stop condition.

Table 20-3 shows what the interaction between a TWI controller and the processor might look like using this example.
Master Mode Receive

Follow these programming steps for a single master mode transmit:

1. Program \texttt{TWI_MASTER_ADDR}. This defines the address transmitted during the address phase of the transfer.

2. Program \texttt{TWI_FIFO_CTRL}. Indicate if receive FIFO buffer interrupts should occur with each byte received (8 bits) or with each 2 bytes received (16 bits).

3. Program \texttt{TWI_INT_ENABLE}. Enable bits associated with the desired interrupt sources. For example, programming the value 0x0030 results in an interrupt output to the processor in the event that the master transfer completes, and the master transfer has an error.

4. Program \texttt{TWI_MASTER_CTRL}. Ultimately this prepares and enables master mode operation. As an example, programming the value 0x0205 enables master mode operation, generates a 7-bit address, sets the direction to master-receive, uses standard mode timing, and receives 8 data bytes before generating a stop condition.

After the \texttt{TWI_DCNT} bit is decremented to zero, the TWI master device sends a NAK to indicate to the slave transmitter that the bus should be released. This allows the master to send the STOP signal to terminate the transfer.
Table 20-4 shows what the interaction between a TWI controller and the processor might look like using this example.

Table 20-4. Master Mode Receive Setup Interaction

<table>
<thead>
<tr>
<th>TWI Controller Master</th>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Repeated Start Condition

In general, a repeated start condition is the absence of a stop condition between two transfers. The two transfers can be of any direction type. Examples include a transmit followed by a receive, or a receive followed by a transmit. The following sections contain information intended to be a guide to assist the programmer in their service routine development.

Transmit/Receive Repeated Start Sequence

Figure 20-25 illustrates a repeated start data transmit followed by a data receive sequence.

Figure 20-25. Transmit/Receive Data Repeated Start
The tasks performed at each interrupt are:

- **XMTSERV interrupt**

  This interrupt was generated due to a FIFO access. Since this is the last byte of this transfer, FIFO_STATUS would indicate the transmit FIFO is empty. When read, DCNT would be zero. Set the RSTART bit to indicate a repeated start and set the MDIR bit should a subsequent transfer be a data receive.

- **MCOMP interrupt**

  This interrupt was generated since all data has been transferred (DCNT = 0). If no errors were generated, a start condition is initiated. Clear the RSTART bit and program the DCNT with the desired number of bytes to receive.

- **RCVSERV interrupt**

  This interrupt is generated due to the arrival of a byte into the receive FIFO. Simple data handling is all that is required.

- **MCOMP interrupt**

  The transfer is complete.

**Receive/Transmit Repeated Start Sequence**

Figure 20-26 illustrates a repeated start data receive followed by a data transmit sequence.
The tasks performed at each interrupt are:

- **RCVSERV interrupt**
  
  This interrupt is generated due to the arrival of a data byte into the receive FIFO. Set the `RSTART` bit to indicate a repeated start and clear the `MDIR` bit should a subsequent transfer be a data transmit.

- **MCOMP interrupt**
  
  This interrupt has occurred due to the completion of the data receive transfer. If no errors were generated, a start condition is initiated. Clear the `RSTART` bit and program the `DCNT` with the desired number of bytes to transmit.

- **XMTSERV interrupt**
  
  This interrupt is generated due to a FIFO access. Simple data handling is all that is required.

- **MCOMP interrupt**
  
  The transfer is complete.

There is no timing constraint to meet the above conditions; the user can program the bits as required. Refer to “Clock Stretching During Repeated Start Condition” on page 20-45 for more on how the controller stretches the clock during repeated start transfers.
Clock Stretching

Clock stretching is an added functionality of the TWI controller in master mode operation. This new behavior utilizes self-induced stretching of the I²C clock while waiting on servicing interrupts. Stretching is done automatically by the hardware and no programming is required for this. The TWI Controller as master supports three modes of clock stretching:

- “Clock Stretching During FIFO Underflow” on page 20-42
- “Clock Stretching During FIFO Overflow” on page 20-44
- “Clock Stretching During Repeated Start Condition” on page 20-45

Clock Stretching During FIFO Underflow

During a master mode transmit, an interrupt is generated at the instant the transmit FIFO becomes empty. At this time, the most recent byte begins transmission. If the XMTSERV interrupt is not serviced, the concluding acknowledge phase of the transfer is stretched. Stretching of the clock continues until new data bytes are written to the transmit FIFO (TWI_XMT_DATA8 or TWI_XMT_DATA16). No other action is required to release the clock and continue the transmission. This behavior continues until the transmission is complete (DCNT = 0) at which time the transmission is concluded (MCOMP) as shown in Figure 20-27 and described in Table 20-5.
Figure 20-27. Clock Stretching During FIFO Underflow

Table 20-5. FIFO Underflow Case

<table>
<thead>
<tr>
<th>TWI Controller</th>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt: XMTSERV – Transmit FIFO buffer is empty.</td>
<td>Acknowledge: Clear interrupt source bits. Write transmit FIFO buffer.</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Interrupt: MCOMP – Master transmit complete (DCNT= 0x00).</td>
<td>Acknowledge: Clear interrupt source bits.</td>
</tr>
</tbody>
</table>
Clock Stretching During FIFO Overflow

During a master mode receive, an interrupt is generated at the instant the receive FIFO becomes full. It is during the acknowledge phase of this received byte that clock stretching begins. No attempt is made to initiate the reception of an additional byte. Stretching of the clock continues until the data bytes previously received are read from the receive FIFO buffer (TWI_RCV_DATA8, TWI_RCV_DATA16). No other action is required to release the clock and continue the reception of data. This behavior continues until the reception is complete (DCNT = 0x00) at which time the reception is concluded (MCOMP) as shown in Figure 20-28 and described in Table 20-6.

![Figure 20-28. Clock Stretching During FIFO Overflow](image-url)
Clock Stretching During Repeated Start Condition

The repeated start feature in I²C protocol requires transitioning between two subsequent transfers. With the use of clock stretching, the task of managing transitions becomes simpler, and common to all transfer types.

Once an initial TWI master transfer has completed (transmit or receive) the clock initiates a stretch during the repeated start phase between transfers. Concurrent with this event the initial transfer will generate a transfer complete interrupt (MCOMP) to signify the initial transfer has completed (DCNT = 0). This initial transfer is handled without any special bit setting sequences or timings. The clock stretching logic described above applies here. With no system related timing constraints the subsequent transfer (receive or transmit) is setup and activated. This sequence can be repeated as many times as required to string a series of repeated start transfers together. This is shown in Figure 20-29 and described in Table 20-7.

Table 20-6. FIFO Overflow Case

<table>
<thead>
<tr>
<th>TWI Controller</th>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Figure 20-29. Clock Stretching During Repeated Start Condition

Table 20-7. Repeated Start Case

<table>
<thead>
<tr>
<th>TWI Controller</th>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt: MCOMP – Initial transmit has completed and DCNT = 0x00.</td>
<td></td>
</tr>
<tr>
<td>Note: transfer in progress, RSTART previously set.</td>
<td>Acknowledge: Clear interrupt source bits.</td>
</tr>
<tr>
<td>Write TWI_MASTER_CTL, setting MDIR (receive), clearing RSTART, and setting new DCNT value (nonzero).</td>
<td></td>
</tr>
<tr>
<td>Read receive FIFO buffer.</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Two-Wire Interface Controllers

Electrical Specifications

This chapter provides hardware, software and system design information to aid users in developing systems based on the Blackfin processor. The design options implemented in a system are influenced by cost, performance, and system requirements. In many cases, design issues cited here are discussed in detail in other sections of this manual. In such cases, a reference appears to the corresponding section of the text, instead of repeating the discussion in this chapter.

Pin Descriptions

Refer to ADSP-BF538/ADSP-BF538F Embedded Processor Data Sheet for pin information, including pin numbers for the 160-lead PBGA package.

Recommendations for Unused Pins

Refer to ADSP-BF538/ADSP-BF538F Embedded Processor Data Sheet for detailed pin descriptions.

Resetting the Processor

In addition to the hardware reset mode provided via the \texttt{RESET} pin, the processor supports several software reset modes. For detailed information on the various modes, see “System Reset and Power-up” on page 3-12.

The processor state after reset is described in “Reset State” on page 3-11.
Booting the Processor

The processor can be booted via a variety of methods. These include executing from external 16-bit memory, booting from a ROM configured to load code from 8-bit flash memory, or booting from a serial ROM (8-bit, 16-bit, or 24-bit address range). For more information on boot modes, see “Booting Methods” on page 3-19.

Figure 21-1 and Figure 21-2 show the connections necessary for 8-bit and 16-bit booting, respectively. Notice that the address connections are made in the same manner for both 8- and 16-bit peripherals. Only the lower byte of each 16-bit word is accessed if byte-wide memory is used.

For example, on core reads of the form:

\[
R0 = W[P0] (Z); \quad // P0 points to a 16-bit aligned ASYNC memory location
\]

only the lower 8 bits of \( R0 \) contain the actual value read from the 8-bit device.

For core writes of the form:

\[
W[P0] = R0.L; \quad // P0 points to a 16-bit aligned ASYNC memory location
\]

The 8-bit value to be written to the 8-bit device should be first loaded into the lower byte of \( R0 \).
Figure 21-1. Interface to 8-Bit SRAM or Flash

Figure 21-2. Interface to 16-Bit SRAM or Flash
Managing Clocks

Systems can drive the clock inputs with a crystal oscillator or a buffered, shaped clock derived from an external clock oscillator. The external clock connects to the processor’s CLKin pin. It is not possible to halt, change, or operate CLKin below the specified frequency during normal operation. The processor uses the clock input (CLKin) to generate on-chip clocks. These include the core clock (CCLK) and the peripheral clock (SCLK).

Managing Core and System Clocks

The processor produces a multiplication of the clock input provided on the CLKin pin to generate the PLL VCO clock. This VCO clock is divided to produce the core clock (CCLK) and the system clock (SCLK). The core clock is based on a divider ratio that is programmed via the CSEL bit settings in the PLL_DIV register. The system clock is based on a divider ratio that is programmed via the SSEL bit settings in the PLL_DIV register. For detailed information about how to set and change CCLK and SCLK frequencies, see Chapter 8, “Dynamic Power Management”.

Configuring and Servicing Interrupts

A variety of interrupts are available. They include both core and peripheral interrupts. The processor assigns default core priorities to system-level interrupts. However, these system interrupts can be remapped via the System interrupt Assignment registers (SIC_IARx). For more information, see “System Interrupt Assignment (SIC_IARx) Registers” on page 4-34.

The processor core supports nested and non-nested interrupts, as well as self-nested interrupts. For explanations of the various modes of servicing events, see “Nesting of Interrupts” on page 4-55.
Semaphores

Semaphores provide a mechanism for communication between multiple processors or processes/threads running in the same system. They are used to coordinate resource sharing. For instance, if a process is using a particular resource and another process requires that same resource, it must wait until the first process signals that it is no longer using the resource. This signalling is accomplished via semaphores.

Semaphore coherency is guaranteed by using the test and set byte (atomic) instruction (TESTSET). The TESTSET instruction performs these functions.

- Loads the half word at memory location pointed to by a P-register. The P-register must be aligned on a half-word boundary.
- Sets $CC$ if the value is equal to zero.
- Stores the value back in its original location (but with the most significant bit (MSB) of the low byte set to 1).

The events triggered by TESTSET are atomic operations. The bus for the memory where the address is located is acquired and not relinquished until the store operation completes. In multithreaded systems, the TESTSET instruction is required to maintain semaphore consistency.

To ensure that the store operation is flushed through any store or write buffers, issue an SSYNC instruction immediately after semaphore release.

The TESTSET instruction can be used to implement binary semaphores or any other type of mutual exclusion method. The TESTSET instruction supports a system-level requirement for a multicycle bus lock mechanism.

The processor restricts use of the TESTSET instruction to the external memory region only. Use of the TESTSET instruction to address any other area of the memory map may result in unreliable behavior.
Example Code for Query Semaphore

Listing 21-1 provides an example of a query semaphore that checks the availability of a shared resource.

Listing 21-1. Query Semaphore

/* Query semaphore. Denotes "Busy" if its value is nonzero. Wait until free (or reschedule thread-- see note below). PO holds address of semaphore. */
QUERY:
TESTSET ( PO ) ;
IF !CC JUMP QUERY ;
/* At this point, semaphore has been granted to current thread, and all other contending threads are postponed because semaphore value at [PO] is nonzero. Current thread could write thread_id to semaphore location to indicate current owner of resource. */
RO.L = THREAD_ID ;
B[PO] = RO ;
/* When done using shared resource, write a zero byte to [PO] */
RO = 0 ;
B[PO] = RO ;
SSYNC ;
/* NOTE: Instead of busy idling in the QUERY loop, one can use an operating system call to reschedule the current thread. */

Data Delays, Latencies and Throughput

For detailed information on latencies and performance estimates on the DMA and external memory buses, refer to Chapter 7, “Chip Bus Hierarchy”.

**Bus Priorities**

For an explanation of prioritization between the various internal buses, refer to Chapter 7, “Chip Bus Hierarchy”.

**External Memory Design Issues**

This section describes design issues related to external memory.

**Example Asynchronous Memory Interfaces**

This section shows glueless connections to 16-bit wide SRAM. Note this interface does not require external assertion of ARDY, since the internal wait state counter is sufficient for deterministic access times of memories.

Figure 21-3 shows the system interconnect required to support 16-bit memories. The programming model must ensure that data is only accessed on 16-bit boundaries.

![Figure 21-3. Interface to 16-Bit SRAM](image-url)
External Memory Design Issues

Using SDRAMs Smaller Than 16M Byte

It is possible to use SDRAMs smaller than 16M byte on the ADSP-BF538 processor, as long as it is understood how the resulting memory map is altered. Figure 21-4 shows an example where a 2M byte SDRAM (512K x 16 bits x 2 banks) is mapped to the external memory interface. In this example, there are 11 row addresses and 8 column addresses per bank. Referring to Table 18-5, the lowest available bank size (16M byte) for a device with 8 column addresses has 2 bank address lines (IA[23:22]) and 13 row address lines (IA[21:9]). Therefore, 1 processor bank address line and 2 row address lines are unused when hooking up to the SDRAM in the example. This causes aliasing in the processor’s external memory map, which results in the SDRAM being mapped into non-contiguous regions of the processor’s memory space.

Referring to the table in Figure 21-4, note that each line in the table corresponds to $2^{19}$ bytes, or 512K byte. Thus, the mapping of the 2M byte SDRAM is non-contiguous in Blackfin memory, as shown by the memory mapping in the left side of the figure.

Managing SDRAM Refresh During PLL Transitions

Since the processor’s SDRAM refresh rate is based on the SCLK frequency, lowering SCLK after configuring SDRAM can result in an improper refresh rate, which could compromise the data stored in SDRAM. Raising SCLK after configuring SDRAM, however, would merely result in a less efficient use of SDRAM, since the processor would just refresh the memory at an unnecessarily fast rate.

In systems where SDRAM is used, the recommended procedure for changing the PLL VCO frequency is:

1. Issue an SSYNC instruction to ensure all pending memory operations have completed.
2. Set the SDRAM to self-refresh mode by writing a 1 to the SRFS bit of EBIU_SDGCTL.

3. Execute the desired PLL programming sequence (refer to Chapter 8, “Dynamic Power Management” for details).

4. After the wake-up occurs that signifies the PLL has settled to the new VCO frequency, reprogram the SDRAM refresh rate control register (EBIU_SDRRC) with a value appropriate to the new SCLK frequency.

5. Bring the SDRAM out of self-refresh mode by clearing the SRFS bit of EBIU_SDGCTL. If it is desired to change the SDRAM Mode register, write these changes to EBIU_SDGCTL as well, making sure the PSSE bit is set.
Changing the SCLK frequency using the SSEL bits in PLL_DIV, as opposed to actually changing the VCO frequency, should be done using these steps:

1. Issue an SSYNC instruction to ensure all pending memory operations have completed.

2. Set the SDRAM to self-refresh mode by writing a 1 to the SRFS bit of EBIU_SDGCTL.

3. Execute the desired write to the SSEL bits.

4. Reprogram the SDRAM refresh rate control register (EBIU_SDRRC) with a value appropriate to the new SCLK frequency.

5. Bring the SDRAM out of self-refresh mode by clearing the SRFS bit of EBIU_SDGCTL. If it is desired to change the SDRAM mode register, write these changes to EBIU_SDGCTL as well, making sure the PSSE bit is set.

Note that steps 2 and 4 are not strictly necessary if changing SCLK to a higher value, but they should always be performed when decreasing SCLK.

For more information on SDRAM refresh, refer to “SDRAM Controller (SDC)” on page 18-22.

Avoiding Bus Contention

Because the three-stated data bus is shared by multiple devices in a system, be careful to avoid contention. Contention causes excessive power dissipation and can lead to device failure. Contention occurs during the time one device is getting off the bus and another is getting on. If the first device is slow to three-state and the second device is quick to drive, the devices contend.

There are two cases where contention can occur. The first case is a read followed by a write to the same memory space. In this case, the data bus drivers can potentially contend with those of the memory device addressed
by the read. The second case is back-to-back reads from two different memory spaces. In this case, the two memory devices addressed by the two reads can potentially contend at the transition between the two read operations.

To avoid contention, program the turnaround time (bank transition time) appropriately in the asynchronous memory bank control registers. This feature allows software to set the number of clock cycles between these types of accesses on a bank-by-bank basis. Minimally, the external bus interface unit (EBIU) provides one cycle for the transition to occur.

High Frequency Design Considerations

Because the processor can operate at very fast clock frequencies, signal integrity and noise problems must be considered for circuit board design and layout. The following sections discuss these topics and suggest various techniques to use when designing and debugging signal processing systems.

Point-to-Point Connections on Serial Ports

Although the serial ports may be operated at a slow rate, the output drivers still have fast edge rates and for longer distances the drivers may require source termination.

You can add a series termination resistor near the pin for point-to-point connections. Typically, serial port applications use this termination method when distances are greater than 6 inches. For details, see the reference source in “Recommended Reading” on page 21-14 for suggestions on transmission line termination. Also, see ADSP-BF538/ADSP-BF538F Embedded Processor Data Sheet for rise and fall time data for the output drivers.
High Frequency Design Considerations

Signal Integrity

The capacitive loading on high-speed signals should be reduced as much as possible. Loading of buses can be reduced by using a buffer for devices that operate with wait states (for example, DRAMs). This reduces the capacitance on signals tied to the zero-wait-state devices, allowing these signals to switch faster and reducing noise-producing current spikes.

Signal run length (inductance) should also be minimized to reduce ringing. Extra care should be taken with certain signals such as external memory, read, write, and acknowledge strobes.

Other recommendations and suggestions to promote signal integrity:

- Use more than one ground plane on the printed circuit board (PCB) to reduce crosstalk. Be sure to use lots of vias between the ground planes. These planes should be in the center of the PCB.

- Keep critical signals such as clocks, strobes, and bus requests on a signal layer next to a ground plane and away from or laid out perpendicular to other non-critical signals to reduce crosstalk.

- Design for lower transmission line impedances to reduce crosstalk and to allow better control of impedance and delay.

- Experiment with the board and isolate crosstalk and noise issues from reflection issues. This can be done by driving a signal wire from a pulse generator and studying the reflections while other components and signals are passive.

Decoupling Capacitors and Ground Planes

Ground planes must be used for the ground and power supplies. The capacitors should be placed very close to the \texttt{VDEXT} and \texttt{VDDINT} pins of the package as shown in Figure 21-5. Use. Use short and fat traces for this. The ground end of the capacitors should be tied directly to the ground.
plane inside the package footprint of the processor (underneath it, on the bottom of the board), not outside the footprint. A surface-mount capacitor is recommended because of its lower series inductance.

Figure 21-5. Bypass Capacitor Placement

Connect the power plane to the power supply pins directly with minimum trace length. The ground planes must not be densely perforated with vias or traces as their effectiveness is reduced. In addition, there should be several large tantalum capacitors on the board.

 Designs can use either bypass placement case or combinations of the two. Designs should try to minimize signal feedthroughs that perforate the ground plane.
High Frequency Design Considerations

Oscilloscope Probes

When making high-speed measurements, be sure to use a “bayonet” type or similarly short (< 0.5 inch) ground clip, attached to the tip of the oscilloscope probe. The probe should be a low-capacitance active probe with 3 pF or less of loading. The use of a standard ground clip with 4 inches of ground lead causes ringing to be seen on the displayed trace and makes the signal appear to have excessive overshoot and undershoot. To see the signals accurately, a 1 GHz or better sampling oscilloscope is needed.

Recommended Reading


This book is a technical reference that covers the problems encountered in state of the art, high-frequency digital circuit design. It is an excellent source of information and practical ideas. Topics covered in the book include:

- High-speed Properties of Logic Gates
- Measurement Techniques
- Transmission Lines
- Ground Planes and Layer Stacking
- Terminations
- Vias
- Power Systems
- Connectors
System Design

- Ribbon Cables
- Clock Distribution
- Clock Oscillators
The Blackfin processor’s debug functionality is used for software debugging. It also complements some services often found in an operating system (OS) kernel. The functionality is implemented in the processor hardware and is grouped into multiple levels.

A summary of available debug features is shown in Table 22-1.

<table>
<thead>
<tr>
<th>Debug Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Watchpoints</td>
<td>Specify address ranges and conditions that halt the processor when satisfied.</td>
</tr>
<tr>
<td>Trace History</td>
<td>Stores the last 16 discontinuous values of the program counter in an on-chip trace buffer.</td>
</tr>
<tr>
<td>Cycle Count</td>
<td>Provides functionality for all code profiling functions.</td>
</tr>
<tr>
<td>Performance Monitoring</td>
<td>Allows internal resources to be monitored and measured non-intrusively.</td>
</tr>
</tbody>
</table>

Watchpoint Unit

By monitoring the addresses on both the instruction bus and the data bus, the watchpoint unit provides several mechanisms for examining program behavior. After counting the number of times a particular address is matched, the unit schedules an event based on this count.
Watchpoint Unit

In addition, information that the watchpoint unit provides helps in the optimization of code. The unit also makes it easier to maintain executables through code patching.

The watchpoint unit contains these memory-mapped registers (MMRs), which are accessible in supervisor and emulator modes:

- The watchpoint status register (WPSTAT)
- Six instruction watchpoint address registers (WPIA[5:0])
- Six instruction watchpoint address count registers (WPIACNT[5:0])
- The instruction watchpoint address control register (WPIACTL)
- Two data watchpoint address registers (WPDA[1:0])
- Two data watchpoint address count registers (WPDACNT[1:0])
- The data watchpoint address control register (WPDACTL)

Two operations implement instruction watchpoints:

- The values in the six instruction watchpoint address registers, WPIA[5:0], are compared to the address on the instruction bus.
- Corresponding count values in the instruction watchpoint address count registers, WPIACNT[5:0], are decremented on each match.

The six instruction watchpoint address registers may be further grouped into three ranges of instruction-address-range watchpoints. The ranges are identified by the addresses in WPIA0 to WPIA1, WPIA2 to WPIA3, and WPIA4 to WPIA5.
The address ranges stored in WPIA0, WPIA1, WPIA2, WPIA3, WPIA4, and WPIA5 must satisfy these conditions:

- WPIA0 <= WPIA1
- WPIA2 <= WPIA3
- WPIA4 <= WPIA5

Two operations implement data watchpoints:

- The values in the two data watchpoint address registers, WPDA[1:0], are compared to the address on the data buses.

- Corresponding count values in the data watchpoint address count registers, WPDACNT[1:0], are decremented on each match.

The two data watchpoint address registers may be further grouped together into one data-address-range watchpoint, WPDA[1:0].

The instruction and data count value registers must be loaded with the number of times the watchpoint must match minus one. After the count value reaches zero, the subsequent watchpoint match results in an exception or emulation event.

Note count values must be reinitialized after the event has occurred.

An event can also be triggered on a combination of the instruction and data watchpoints. If the WPAND bit in the WPIACTL register is set, then an event is triggered only when both an instruction address watchpoint matches and a data address watchpoint matches. If the WPAND bit is 0, then an event is triggered when any of the enabled watchpoints or watchpoint ranges match.
Watchpoint Unit

To enable the watchpoint unit, the \texttt{WPPWR} bit in the \texttt{WPIACTL} register must be set. If \( \texttt{WPPWR} = 1 \), then the individual watchpoints and watchpoint ranges may be enabled using the specific enable bits in the \texttt{WPIACTL} and \texttt{WPDACTL} MMRs. If \( \texttt{WPPWR} = 0 \), then all watchpoint activity is disabled.

Instruction Watchpoints

Each instruction watchpoint is controlled by three bits in the \texttt{WPIACTL} register, as shown in Table 22-2.

Table 22-2. WPIACTL Control Bits

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{EMUSWx}</td>
<td>Determines whether an instruction-address match causes either an emulation event or an exception event.</td>
</tr>
<tr>
<td>\texttt{WPICNTENx}</td>
<td>Enables the 16-bit counter that counts the number of address matches. If the counter is disabled, then every match causes an event.</td>
</tr>
<tr>
<td>\texttt{WPIAENx}</td>
<td>Enables the address watchpoint activity.</td>
</tr>
</tbody>
</table>

When two watchpoints are associated to form a range, two additional bits are used, as shown in Table 22-3.

Table 22-3. WPIACTL Watchpoint Range Control Bits

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{WPIRENxy}</td>
<td>Indicates the two watchpoints that are to be associated to form a range.</td>
</tr>
<tr>
<td>\texttt{WPIRINVxy}</td>
<td>Determines whether an event is caused by an address within the range identified or outside of the range identified.</td>
</tr>
</tbody>
</table>
Code patching allows software to replace sections of existing code with new code. The watchpoint registers are used to trigger an exception at the start addresses of the earlier code. The exception routine then vectors to the location in memory that contains the new code.

On the processor, code patching can be achieved by writing the start address of the earlier code to one of the $\text{WPIAn}$ registers and setting the corresponding $\text{EMUSWx}$ bit to trigger an exception. In the exception service routine, the $\text{WPSTAT}$ register is read to determine which watchpoint triggered the exception. Next, the code writes the start address of the new code in the $\text{RETX}$ register, and then returns from the exception to the new code. Because the exception mechanism is used for code patching, event service routines of the same or higher priority (exception, NMI, and reset routines) cannot be patched.

A write to the $\text{WPSTAT}$ MMR clears all the sticky status bits. The data value written is ignored.

**Instruction Watchpoint Address (WPIAn) Registers**

When the watchpoint unit is enabled, the values in the instruction watchpoint address registers ($\text{WPIAn}$) are compared to the address on the instruction bus. Corresponding count values in the instruction watchpoint address count registers ($\text{WPIACNTn}$) are decremented on each match.

Figure 22-1 shows the instruction watchpoint address registers, $\text{WPIA}[5:0]$. 
Watchpoint Unit

Instruction Watchpoint Address Count (WPIACNTn) Registers

When the watchpoint unit is enabled, the count values in the instruction watchpoint address count registers (WPIACNT[5:0]) are decremented each time the address or the address bus matches a value in the WPIAn registers. Load the WPIACNTn register with a value that is one less than the number of times the watchpoint must match before triggering an event (see Figure 22-2). The WPIACNTn register decrements to 0x0000 when the programmed count expires.

Table 22-4. Instruction Watchpoint Register Memory-Mapped Addresses

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>WPIA0</td>
<td>0xFFF0 7040</td>
</tr>
<tr>
<td>WPIA1</td>
<td>0xFFF0 7044</td>
</tr>
<tr>
<td>WPIA2</td>
<td>0xFFF0 7048</td>
</tr>
<tr>
<td>WPIA3</td>
<td>0xFFF0 704C</td>
</tr>
<tr>
<td>WPIA4</td>
<td>0xFFF0 7050</td>
</tr>
<tr>
<td>WPIA5</td>
<td>0xFFF0 7054</td>
</tr>
</tbody>
</table>

Figure 22-1. Instruction Watchpoint Address Registers

Table 22-4. Instruction Watchpoint Register Memory-Mapped Addresses

For memory-mapped addresses, see Table 22-4.
Blackfin Processor Debug

Instruction Watchpoint Address Count Registers (WPIACNTn)

For memory-mapped addresses, see Table 22-5.

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Memory-Mapped Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>WPIACNT0</td>
<td>0xFFE0 7080</td>
</tr>
<tr>
<td>WPIACNT1</td>
<td>0xFFE0 7084</td>
</tr>
<tr>
<td>WPIACNT2</td>
<td>0xFFE0 7088</td>
</tr>
<tr>
<td>WPIACNT3</td>
<td>0xFFE0 708C</td>
</tr>
<tr>
<td>WPIACNT4</td>
<td>0xFFE0 7090</td>
</tr>
<tr>
<td>WPIACNT5</td>
<td>0xFFE0 7094</td>
</tr>
</tbody>
</table>

Instruction Watchpoint Address Count Registers (WPIACNTn)

Figure 22-2. Instruction Watchpoint Address Count Registers

Table 22-5. Instruction Watchpoint Address Count Register Memory-Mapped Addresses

Instruction Watchpoint Address Control (WPIACTL) Register

Three bits in the instruction watchpoint address control register (WPIACTL) control each instruction watchpoint. Figure 22-3 describes the upper half of the register. Figure 22-4 describes the lower half of the register. For more information about the bits in this register, see “Instruction Watchpoints” on page 22-4.

The bits in the WPIACTL register have no effect unless the WPPWR bit is set.
Watchpoint Unit

Instruction Watchpoint Address Control Register (WPIACTL)

In range comparisons, IA = instruction address

Figure 22-3. Instruction Watchpoint Address Control Register (WPIACTL)[31:16]
### Instruction Watchpoint Address Control Register (WPIACTL)

<table>
<thead>
<tr>
<th>Bit 15</th>
<th>Bit 14</th>
<th>Bit 13</th>
<th>Bit 12</th>
<th>Bit 11</th>
<th>Bit 10</th>
<th>Bit 9</th>
<th>Bit 8</th>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EMUSW2</strong></td>
<td><strong>EMUSW1</strong></td>
<td><strong>WPIWR</strong></td>
<td><strong>WPIREN01</strong></td>
<td><strong>WPIREN23</strong></td>
<td><strong>WPIREN23</strong></td>
<td><strong>WPIRINV01</strong></td>
<td><strong>WPIRINV23</strong></td>
<td><strong>WPIAEN0</strong></td>
<td><strong>WPIAEN1</strong></td>
<td><strong>WPIAEN2</strong></td>
<td><strong>WPIAEN3</strong></td>
<td><strong>WPICNTEN0</strong></td>
<td><strong>WPICNTEN1</strong></td>
<td><strong>WPICNTEN2</strong></td>
<td><strong>WPICNTEN3</strong></td>
</tr>
<tr>
<td>0 - Match on WPIA2 (or range 23) causes an exception event</td>
<td>0 - Match on WPIA1 causes an exception event</td>
<td>0 - Watchpoints Unit disabled</td>
<td>0 - Disable range comparison</td>
<td>0 - Disable range comparison</td>
<td>0 - Disable range comparison</td>
<td>Valid when WPIREN01 = 1</td>
<td>0 - Inclusive range comparison: WPIA2 &lt; IA &lt;= WPIA3</td>
<td>Valid when WPIREN01 = 0</td>
<td>0 - Disable instruction address watchpoints, WPIA0</td>
<td>0 - Disable instruction address watchpoints, WPIA3</td>
<td>0 - Disable instruction address watchpoints, WPIA0</td>
<td>0 - Disable instruction address watchpoints, WPIA2</td>
<td>0 - Disable instruction address watchpoints, WPIA2</td>
<td>0 - Disable instruction address watchpoints, WPIA2</td>
<td></td>
</tr>
<tr>
<td>1 - Match on WPIA2 (or range 23) causes an emulation event</td>
<td>1 - Match on WPIA1 causes an emulation event</td>
<td>1 - Watchpoints Unit enabled</td>
<td>1 - Enable range comparison: (Start address = WPIA0, End address = WPIA1)</td>
<td>1 - Enable range comparison: (Start address = WPIA2, End address = WPIA3)</td>
<td>1 - Enable range comparison: (Start address = WPIA2, End address = WPIA3)</td>
<td>Valid when WPIREN01 = 1</td>
<td>1 - Inclusive range comparison: IA &lt;= WPIA0</td>
<td></td>
<td>IA &gt; WPIA1</td>
<td>0 - Disable instruction address watchpoints, WPIA0</td>
<td>1 - Enable instruction address watchpoints, WPIA1</td>
<td>1 - Enable instruction address watchpoints, WPIA3</td>
<td>1 - Enable instruction address watchpoints, WPIA2</td>
<td>1 - Enable instruction address watchpoints, WPIA2</td>
<td>1 - Enable instruction address watchpoints, WPIA2</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>WPIREN01</strong></td>
<td><strong>WPIREN23</strong></td>
<td><strong>WPIREN23</strong></td>
<td><strong>WPIREN23</strong></td>
<td></td>
<td></td>
<td><strong>WPICNTEN0</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - Disable range comparison</td>
<td>0 - Disable range comparison</td>
<td>0 - Disable range comparison</td>
<td></td>
<td></td>
<td>Valid when WPIREN01 = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - Enable range comparison: (Start address = WPIA0, End address = WPIA1)</td>
<td>1 - Enable range comparison: (Start address = WPIA2, End address = WPIA3)</td>
<td>1 - Enable range comparison: (Start address = WPIA2, End address = WPIA3)</td>
<td></td>
<td></td>
<td>0 - Inclusive range comparison: WPIA2 &lt; IA &lt;= WPIA3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Valid when WPIREN01 = 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 - Disable instruction address watchpoints, WPIA0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 - Enable instruction address watchpoints, WPIA1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>WPIAEN0</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 - Disable instruction address watchpoints, WPIA0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 - Enable instruction address watchpoints, WPIA0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>WPIAEN1</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 - Disable instruction address watchpoints, WPIA1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 - Enable instruction address watchpoints, WPIA1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>WPIAEN2</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 - Disable instruction address watchpoints, WPIA2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 - Enable instruction address watchpoints, WPIA2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>WPICNTEN0</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 - Disable watchpoints instruction address counter 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 - Enable watchpoints instruction address counter 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>WPICNTEN1</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 - Disable watchpoints instruction address counter 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 - Enable watchpoints instruction address counter 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 22-4. Instruction Watchpoint Address Control Register \([15:0]\)**
Watchpoint Unit

Data Address Watchpoints

Each data watchpoint is controlled by four bits in the WPDACTL register, as shown in Table 22-6.

Table 22-6. Data Address Watchpoints

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WPDACCn</td>
<td>Determines whether the match should be on a read or write access.</td>
</tr>
<tr>
<td>WPDSRCn</td>
<td>Determines which DAG the unit should monitor.</td>
</tr>
<tr>
<td>WPDCNTENn</td>
<td>Enables the counter that counts the number of address matches. If the counter is disabled, then every match causes an event.</td>
</tr>
<tr>
<td>WPDAENn</td>
<td>Enables the data watchpoint activity.</td>
</tr>
</tbody>
</table>

Alternatively, the watchpoint unit can be configured to monitor a range of data addresses. To enable this function, the WPDAENO and WPDAEN1 bits are not used and must be set to 0. Instead, the WPDREN01 and WPDRINV01 bits are used to configure the watchpoint unit, as described in Table 22-7.

Table 22-7. WPDACTL Watchpoint Control Bits

<table>
<thead>
<tr>
<th>Bit Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WPDREN01</td>
<td>Indicates the two watchpoints associated to form a range.</td>
</tr>
<tr>
<td>WPDRINV01</td>
<td>Determines whether an event is caused by an address within the range identified or outside the range.</td>
</tr>
</tbody>
</table>

Note data address watchpoints always trigger emulation events.
Data Watchpoint Address (WPDAn) Registers

When the watchpoint unit is enabled, the values in the data watchpoint address registers (WPDAn) are compared to the address on the data buses. Corresponding count values in the data watchpoint address count registers (WPDA[1:0]) are decremented on each match.

Figure 22-5 shows the data watchpoint address registers, WPDA[1:0].

Data Watchpoint Address Registers (WPDAn)

| WPDA0: 0xFFE0 7140 | X X X X X X X X X X X X X X |
| WPDA1: 0xFFE0 7144 | X X X X X X X X X X X X X X |

WPDA (Data Address)[31:16]

X X X X X X X X X X X X X X

WPDA (Data Address)[15:0]

X X X X X X X X X X X X X X

Figure 22-5. Data Watchpoint Address Registers

Data Watchpoint Address Count Value (WPDA[1:0]) Registers

When the watchpoint unit is enabled, the count values in the data watchpoint address count value registers (WPDA[1:0]) are decremented each time the address or the address bus matches a value in the WPDAn registers. Load this WPDA[1:0] register with a value that is one less than the number of times the watchpoint must match before triggering an event. The WPDA[1:0] register will decrement to 0x0000 when the programmed count expires. Figure 22-6 shows the WPDA[1:0] registers.

Figure 22-6 shows the WPDA[1:0] registers.
Data Watchpoint Address Count Value Registers (WPDACNTn)

- **WPDACNT0**: 0xFFE0 7180
- **WPDACNT1**: 0xFFE0 7184

![Figure 22-6. Data Watchpoint Address Count Value Registers](image)

**Data Watchpoint Address Control (WPDACTL) Register**

For more information about the bits in the data watchpoint address control register (WPDACTL), see “Data Address Watchpoints” on page 22-10.

**Watchpoint Status (WPSTAT) Register**

The watchpoint status register (WPSTAT) monitors the status of the watchpoints. It may be read and written in supervisor or emulator modes only. When a watchpoint or watchpoint range matches, this register reflects the source of the watchpoint. The status bits in the WPSTAT register are sticky, and all of them are cleared when any write, regardless of the value, is performed to the register.

*Figure 22-8 shows the watchpoint status register.*
Data Watchpoint Address Control Register (WPDACTL)

**Figure 22-7. Data Watchpoint Address Control Register**
Trace Unit

The trace unit stores a history of the last 16 changes in program flow taken by the program sequencer. The history allows the user to recreate the program sequencer’s recent path.

The trace buffer can be enabled to cause an exception when full. The exception service routine associated with the exception saves trace buffer entries to memory. Thus, the complete path of the program sequencer since the trace buffer was enabled can be recreated.

Figure 22-8. Watchpoint Status Register
Changes in program flow because of zero-overhead loops are not stored in the trace buffer. For debugging code that is halted within a zero-overhead loop, the iteration count is available in the loop count registers, \( LC_0 \) and \( LC_1 \).

The trace buffer can be configured to omit the recording of changes in program flow that match either the last entry or one of the last two entries. Omitting one of these entries from the record prevents the trace buffer from overflowing because of loops in the program. Because zero-overhead loops are not recorded in the trace buffer, this feature can be used to prevent trace overflow from loops that are nested four deep.

When read, the trace buffer register (\( TBUF \)) returns the top value from the trace unit stack, which contains as many as 16 entries. Each entry contains a pair of branch source and branch target addresses. A read of \( TBUF \) returns the newest entry first, starting with the branch destination. The next read provides the branch source address.

The number of valid entries in \( TBUF \) is held in the \( TBUFCNT \) field of the \( TBUFSTAT \) register. On every second read, \( TBUFCNT \) is decremented. Because each entry corresponds to two pieces of data, a total of \( 2 \times TBUFCNT \) reads empties the \( TBUF \) register.

Discontinuities that are the same as either of the last two entries in the trace buffer are not recorded.

Because reading the trace buffer is a destructive operation, it is recommended that \( TBUF \) be read in a non-interruptible section of code.

Note, if single-level compression has occurred, the least significant bit (LSB) of the branch target address is set. If two-level compression has occurred, the LSB of the branch source address is set.
Trace Buffer Control (TBUFCTL) Register

The trace unit is enabled by two control bits in the trace buffer control register (TBUFCTL) register. First, the trace unit must be activated by setting the TBUFPWR bit. If TBUFPWR = 1, then setting TBUFEN to 1 enables the trace unit.

Figure 22-9 describes the TBUFCTL register. If TBUFOVF = 1, then the trace unit does not record discontinuities in the exception, NMI, and reset routines.

Figure 22-9. Trace Buffer Control Register
Trace Buffer Status (TBUFSTAT) Register

Figure 22-10 shows the trace buffer status register (TBUFSTAT). Two reads from TBUF decrements TBUFCNT by one.

Trace Buffer Status Register (TBUFSTAT)

![TBUFSTAT Register](image)

TBUFCNT[4:0]
Number of valid discontinuities stored in the trace buffer

Figure 22-10. Trace Buffer Status Register

Trace Buffer (TBUF) Register

Figure 22-11 shows the trace buffer register (TBUF). The first read returns the latest branch target address. The second read returns the latest branch source address.

Trace Buffer Register (TBUF)

![TBUF Register](image)

TBUF[31:16]
Alias to all trace buffer entries

Figure 22-11. Trace Buffer Register
The trace unit does not record changes in program flow in:

- Emulator mode
- The exception or higher priority service routines (if $TBUFOVF = 1$)

In the exception service routine, the program flow discontinuities may be read from $TBUF$ and stored in memory by the code shown in Listing 22-1.

⚠️ While $TBUF$ is being read, be sure to disable the trace buffer from recording new discontinuities.

### Code to Recreate the Execution Trace in Memory

Listing 22-1 provides code that recreates the entire execution trace in memory.

#### Listing 22-1. Recreating the Execution Trace in Memory

```assembly
[--sp] = (r7:7, p5:2); /* save registers used in this routine */
p5 = 32; /* 32 reads are needed to empty TBUF */
p2.l = buf; /* pointer to the header (first location) of the software trace buffer */
p2.h = buf; /* the header stores the first available empty buf location for subsequent trace dumps */

p4 = [p2++]; /* get the first available empty buf location from the buf header */
p3.l = TBUF & 0xffff; /* low 16 bits of TBUF */
p3.h = TBUF >> 16; /* high 16 bits of TBUF */
```
lsetup(loop1_start, loop1_end) lc0 = p5;
loop1_start: r7 = [p3];  /* read from TBUF */
loop1_end: [p4++] = r7;  /* write to memory and increment */
[p2] = p4;  /* pointer to the next available buf location is saved in the header of buf */
(r7:7, p5:3) = [sp++];  /* restore saved registers */

Performance Monitoring Unit

Two 32-bit counters, the performance monitor counter registers (PFCNTR[1:0]) and the performance control register (PFCTL), count the number of occurrences of an event from within a processor core unit during a performance monitoring period. These registers provide feedback indicating the measure of load balancing between the various resources on the chip so that expected and actual usage can be compared and analyzed. In addition, events such as mispredictions and hold cycles can also be monitored.

Performance Monitor Counter (PFCNTRn) Registers

Figure 22-12 shows the performance monitor counter registers, PFCNTR[1:0]. The PFCNTR0 register contains the count value of performance counter 0. The PFCNTR1 register contains the count value of performance counter 1.
To enable the performance monitoring unit, set the PFPWR bit in the performance monitor control register (PFCTL), shown in Figure 22-13. Once the unit is enabled, individual count-enable bits (PFCENn) take effect. Use the PFCENx bits to enable or disable the performance monitors in user mode, supervisor mode, or both. Use the PEMUSWx bits to select the type of event triggered.
**Performance Monitor Control Register (PFCTL)**

![Diagram of Performance Monitor Control Register](image)

**Performance Monitor Control Register (PFCTL)**

- **0xFFE0 8000**
- **PFMON1[7:0]**
  - Refer to Event Monitor table on page 22-22.
- **PFPWR**
  - 0 - Performance Monitor disabled
  - 1 - Performance Monitor enabled
- **PEMUSW0**
  - 0 - Count down of performance counter PFCNTR0 causes exception event
  - 1 - Count down of performance counter PFCNTR0 causes emulation event
- **PEMUSW1**
  - 0 - Count down of performance counter PFCNTR1 causes exception event
  - 1 - Count down of performance counter PFCNTR1 causes emulation event
- **PFCEN0[1:0]**
  - 00 - Disable Performance Monitor 0
  - 01 - Enable Performance Monitor 0 in User mode only
  - 10 - Enable Performance Monitor 0 in Supervisor mode only
  - 11 - Enable Performance Monitor 0 in both User and Supervisor modes
- **PFCEN1[1:0]**
  - 00 - Disable Performance Monitor 1
  - 01 - Enable Performance Monitor 1 in User mode only
  - 10 - Enable Performance Monitor 1 in Supervisor mode only
  - 11 - Enable Performance Monitor 1 in both User and Supervisor modes
- **PFCNT0**
  - 0 - Count number of cycles asserted
  - 1 - Count positive edges only
- **PFCNT1**
  - 0 - Count number of cycles asserted
  - 1 - Count positive edges only

**Figure 22-13. Performance Monitor Control Register**
**Event Monitor Table**

Table 22-8 identifies events that cause the performance monitor counter registers (PFMON0 or PFMON1) to increment.

### Table 22-8. Event Monitor Table

<table>
<thead>
<tr>
<th>PFMONx Fields</th>
<th>Events That Cause the Count Value to Increment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Loop 0 iterations</td>
</tr>
<tr>
<td>0x01</td>
<td>Loop 1 iterations</td>
</tr>
<tr>
<td>0x02</td>
<td>Loop buffer 0 not optimized</td>
</tr>
<tr>
<td>0x03</td>
<td>Loop buffer 1 not optimized</td>
</tr>
<tr>
<td>0x04</td>
<td>PC invariant branches (requires trace buffer to be enabled. See “Trace Buffer Control (TBUFCTL) Register” on page 22-16).</td>
</tr>
<tr>
<td>0x06</td>
<td>Conditional branches</td>
</tr>
<tr>
<td>0x09</td>
<td>Total branches including calls, returns, branches, but not interrupts (requires trace buffer to be enabled. See “Trace Buffer Control (TBUFCTL) Register” on page 22-16).</td>
</tr>
<tr>
<td>0x0A</td>
<td>Stalls due to CSYNC, SSYNC</td>
</tr>
<tr>
<td>0x0B</td>
<td>EXCPT instructions</td>
</tr>
<tr>
<td>0x0C</td>
<td>CSYNC, SSYNC instructions</td>
</tr>
<tr>
<td>0x0D</td>
<td>Committed instructions</td>
</tr>
<tr>
<td>0x0E</td>
<td>Interrupts taken</td>
</tr>
<tr>
<td>0x0F</td>
<td>Misaligned address violation exceptions</td>
</tr>
<tr>
<td>0x10</td>
<td>Stall cycles due to read after write hazards on DAG registers</td>
</tr>
<tr>
<td>0x13</td>
<td>Stall cycles due to RAW data hazards in computes</td>
</tr>
<tr>
<td>0x80</td>
<td>Code memory fetches postponed due to DMA collisions (minimum count of two per event)</td>
</tr>
<tr>
<td>0x81</td>
<td>Code memory TAG stalls (cache misses, or FlushI operations, count of 3 per FlushI). Note code memory stall results in a processor stall only if instruction assembly unit FIFO empties.</td>
</tr>
</tbody>
</table>
Table 22-8. Event Monitor Table (Cont’d)

<table>
<thead>
<tr>
<th>PFMONx Fields</th>
<th>Events That Cause the Count Value to Increment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x82</td>
<td>Code memory fill stalls (cacheable or non-cacheable). Note code memory stall results in a processor stall only if instruction assembly unit FIFO empties.</td>
</tr>
<tr>
<td>0x83</td>
<td>Code memory 64-bit words delivered to processor instruction assembly unit</td>
</tr>
<tr>
<td>0x90</td>
<td>Processor stalls to memory</td>
</tr>
<tr>
<td>0x91</td>
<td>Data memory stalls to processor not hidden by processor stall</td>
</tr>
<tr>
<td>0x92</td>
<td>Data memory store buffer full stalls</td>
</tr>
<tr>
<td>0x93</td>
<td>Data memory write buffer full stalls due to high-to-low priority code transition</td>
</tr>
<tr>
<td>0x94</td>
<td>Data memory store buffer forward stalls due to lack of committed data from the processor</td>
</tr>
<tr>
<td>0x95</td>
<td>Data memory fill buffer stalls</td>
</tr>
<tr>
<td>0x96</td>
<td>Data memory array or TAG collision stalls (DAG to DAG, or DMA to DAG)</td>
</tr>
<tr>
<td>0x97</td>
<td>Data memory array collision stalls (DAG to DAG or DMA to DAG)</td>
</tr>
<tr>
<td>0x98</td>
<td>Data memory stalls</td>
</tr>
<tr>
<td>0x99</td>
<td>Data memory stalls sent to processor</td>
</tr>
<tr>
<td>0x9A</td>
<td>Data memory cache fills completed to bank A</td>
</tr>
<tr>
<td>0x9B</td>
<td>Data memory cache fills completed to bank B</td>
</tr>
<tr>
<td>0x9C</td>
<td>Data memory cache victims delivered from bank A</td>
</tr>
<tr>
<td>0x9D</td>
<td>Data memory cache victims delivered from bank B</td>
</tr>
<tr>
<td>0x9E</td>
<td>Data memory cache high priority fills requested</td>
</tr>
<tr>
<td>0x9F</td>
<td>Data memory cache low priority fills requested</td>
</tr>
</tbody>
</table>

**Cycle Counter**

The cycle counter counts $CCLK$ cycles while the program is executing. All cycles, including execution, wait state, interrupts, and events, are counted while the processor is in user or supervisor mode, but the cycle counter stops counting in emulator mode.
The cycle counter is 64 bits and increments every cycle. The count value is stored in two 32-bit registers, CYCLES and CYCLES2. The least significant 32 bits (LSBs) are stored in CYCLES. The most significant 32 bits (MSBs) are stored in CYCLES2.

To ensure read coherency, a read of CYCLES stores the current CYCLES2 value in a shadow register, and a subsequent read of CYCLES2 comes from the shadow register.

To enable the cycle counters, set the CCEN bit in the SYSCFG register. The following example shows how to use the cycle counter to benchmark a piece of code:

```
R2 = 0;
CYCLES = R2;
CYCLES2 = R2;
R2 = SYSCFG;
BITSET(R2,1);
SYSCFG = R2;
/* Insert code to be benchmarked here. */
R2 = SYSCFG;
BITCLR(R2,1);
SYSCFG = R2;
```

**CYCLES and CYCLES2 Registers**

The execution cycle count registers (CYCLES and CYCLES2) are shown in Figure 22-14. This 64-bit counter increments every CCLK cycle. The CYCLES register contains the least significant 32 bits of the cycle counter’s 64-bit count value. The most significant 32 bits are contained by CYCLES2.
The CYCLES and CYCLES2 registers are not system MMRs, but are instead system registers. See “Register File Instruction Summary” on page 2-8 for a full listing of system registers.

Execution Cycle Count Registers (CYCLES and CYCLES2)

<table>
<thead>
<tr>
<th>RW</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16</td>
</tr>
<tr>
<td>X X X X X X X X X X X X X X X X</td>
</tr>
</tbody>
</table>

Reset = Undefined

CYCLES / CYCLES2[31:16]

<table>
<thead>
<tr>
<th>RW</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>X X X X X X X X X X X X X X</td>
</tr>
</tbody>
</table>

Figure 22-14. Execution Cycle Count Registers

Product Identification Register

The 32-bit DSP device ID register (DSPID) is a core MMR that contains core identification and revision fields for the core.

DSP Device ID (DSPID) Register

The DSP device ID register (DSPID), shown in Figure 22-15, is a read-only register and is part of the core.
## Product Identification Register

**DSP Device ID Register (DSPID)**

RO

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Reset = E504 0000

Analog Devices, Inc.[7:0]

Major Architectural Change[7:0]

Implementation[15:0]

Figure 22-15. DSP Device ID Register
A BLACKFIN PROCESSOR
CORE MMR ASSIGNMENTS

The Blackfin processor’s memory-mapped registers (MMRs) are in the address range 0xFFE0 0000 – 0xFFFF FFFF.

All core MMRs must be accessed with a 32-bit read or write access.

This appendix lists core MMR addresses and register names. To find more information about an MMR, refer to the page shown in the “See Section” column. When viewing the PDF version of this document, click a reference in the “See Section” column to jump to additional information about the MMR.

L1 Data Memory Controller Registers

L1 data memory controller registers (0xFFE0 0000 – 0xFFE0 0404)

Table A-1. L1 Data Memory Controller Registers

| Memory-Mapped Address | Register Name             | See Section                                                                 |
|-----------------------|---------------------------|                                                                            |
| 0xFFE0 0004           | DMEM_CONTROL              | “Data Memory Control (DMEM_CONTROL) Register” on page 6-24                  |
| 0xFFE0 0008           | DCPLB_STATUS              | “Instruction and Data CPLB Status (ICPLB_STATUS, DCPLB_STATUS) Registers” on page 6-59 |
| 0xFFE0 000C           | DCPLB_FAULT_ADDR          | “Instruction and Data CPLB Fault Address (ICPLB_FAULT_ADDR, DCPLB_FAULT_ADDR) Registers” on page 6-60 |
### Table A-1. L1 Data Memory Controller Registers (Cont’d)

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFE0 0100</td>
<td>DCPLB_ADDR0</td>
<td>“Data CPLB Address (DCPLB_ADDRx) Registers” on page 6-56</td>
</tr>
<tr>
<td>0xFFE0 0104</td>
<td>DCPLB_ADDR1</td>
<td>“Data CPLB Address (DCPLB_ADDRx) Registers” on page 6-56</td>
</tr>
<tr>
<td>0xFFE0 0108</td>
<td>DCPLB_ADDR2</td>
<td>“Data CPLB Address (DCPLB_ADDRx) Registers” on page 6-56</td>
</tr>
<tr>
<td>0xFFE0 010C</td>
<td>DCPLB_ADDR3</td>
<td>“Data CPLB Address (DCPLB_ADDRx) Registers” on page 6-56</td>
</tr>
<tr>
<td>0xFFE0 0110</td>
<td>DCPLB_ADDR4</td>
<td>“Data CPLB Address (DCPLB_ADDRx) Registers” on page 6-56</td>
</tr>
<tr>
<td>0xFFE0 0114</td>
<td>DCPLB_ADDR5</td>
<td>“Data CPLB Address (DCPLB_ADDRx) Registers” on page 6-56</td>
</tr>
<tr>
<td>0xFFE0 0118</td>
<td>DCPLB_ADDR6</td>
<td>“Data CPLB Address (DCPLB_ADDRx) Registers” on page 6-56</td>
</tr>
<tr>
<td>0xFFE0 011C</td>
<td>DCPLB_ADDR7</td>
<td>“Data CPLB Address (DCPLB_ADDRx) Registers” on page 6-56</td>
</tr>
<tr>
<td>0xFFE0 0120</td>
<td>DCPLB_ADDR8</td>
<td>“Data CPLB Address (DCPLB_ADDRx) Registers” on page 6-56</td>
</tr>
<tr>
<td>0xFFE0 0124</td>
<td>DCPLB_ADDR9</td>
<td>“Data CPLB Address (DCPLB_ADDRx) Registers” on page 6-56</td>
</tr>
<tr>
<td>0xFFE0 0128</td>
<td>DCPLB_ADDR10</td>
<td>“Data Watchpoint Address Count Value (WPDACNTn) Registers” on page 22-11</td>
</tr>
<tr>
<td>0xFFE0 012C</td>
<td>DCPLB_ADDR11</td>
<td>“Data CPLB Address (DCPLB_ADDRx) Registers” on page 6-56</td>
</tr>
<tr>
<td>0xFFE0 0130</td>
<td>DCPLB_ADDR12</td>
<td>“Data CPLB Address (DCPLB_ADDRx) Registers” on page 6-56</td>
</tr>
<tr>
<td>0xFFE0 0134</td>
<td>DCPLB_ADDR13</td>
<td>“Data CPLB Address (DCPLB_ADDRx) Registers” on page 6-56</td>
</tr>
<tr>
<td>0xFFE0 0138</td>
<td>DCPLB_ADDR14</td>
<td>“Data CPLB Address (DCPLB_ADDRx) Registers” on page 6-56</td>
</tr>
</tbody>
</table>
## Table A-1. L1 Data Memory Controller Registers (Cont’d)

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFE0 013C</td>
<td>DCPLB_ADDR15</td>
<td>“Data CPLB Address (DCPLB_ADDRx) Registers” on page 6-56</td>
</tr>
<tr>
<td>0xFFE0 0200</td>
<td>DCPLB_DATA0</td>
<td>“Data CPLB Data (DCPLB_DATAx) Registers” on page 6-53</td>
</tr>
<tr>
<td>0xFFE0 0204</td>
<td>DCPLB_DATA1</td>
<td>“Data CPLB Data (DCPLB_DATAx) Registers” on page 6-53</td>
</tr>
<tr>
<td>0xFFE0 0208</td>
<td>DCPLB_DATA2</td>
<td>“Data CPLB Data (DCPLB_DATAx) Registers” on page 6-53</td>
</tr>
<tr>
<td>0xFFE0 020C</td>
<td>DCPLB_DATA3</td>
<td>“Data CPLB Data (DCPLB_DATAx) Registers” on page 6-53</td>
</tr>
<tr>
<td>0xFFE0 0210</td>
<td>DCPLB_DATA4</td>
<td>“Data CPLB Data (DCPLB_DATAx) Registers” on page 6-53</td>
</tr>
<tr>
<td>0xFFE0 0214</td>
<td>DCPLB_DATA5</td>
<td>“Data CPLB Data (DCPLB_DATAx) Registers” on page 6-53</td>
</tr>
<tr>
<td>0xFFE0 0218</td>
<td>DCPLB_DATA6</td>
<td>“Data CPLB Data (DCPLB_DATAx) Registers” on page 6-53</td>
</tr>
<tr>
<td>0xFFE0 021C</td>
<td>DCPLB_DATA7</td>
<td>“Data CPLB Data (DCPLB_DATAx) Registers” on page 6-53</td>
</tr>
<tr>
<td>0xFFE0 0220</td>
<td>DCPLB_DATA8</td>
<td>“Data CPLB Data (DCPLB_DATAx) Registers” on page 6-53</td>
</tr>
<tr>
<td>0xFFE0 0224</td>
<td>DCPLB_DATA9</td>
<td>“Data CPLB Data (DCPLB_DATAx) Registers” on page 6-53</td>
</tr>
<tr>
<td>0xFFE0 0228</td>
<td>DCPLB_DATA10</td>
<td>“Data CPLB Data (DCPLB_DATAx) Registers” on page 6-53</td>
</tr>
<tr>
<td>0xFFE0 022C</td>
<td>DCPLB_DATA11</td>
<td>“Data CPLB Data (DCPLB_DATAx) Registers” on page 6-53</td>
</tr>
<tr>
<td>0xFFE0 0230</td>
<td>DCPLB_DATA12</td>
<td>“Data CPLB Data (DCPLB_DATAx) Registers” on page 6-53</td>
</tr>
<tr>
<td>0xFFE0 0234</td>
<td>DCPLB_DATA13</td>
<td>“Data CPLB Data (DCPLB_DATAx) Registers” on page 6-53</td>
</tr>
</tbody>
</table>
Table A-1. L1 Data Memory Controller Registers (Cont’d)

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFE0 0238</td>
<td>DCPLB_DATA14</td>
<td>“Data CPLB Data (DCPLB_DATAx) Registers” on page 6-53</td>
</tr>
<tr>
<td>0xFFE0 023C</td>
<td>DCPLB_DATA15</td>
<td>“Data CPLB Data (DCPLB_DATAx) Registers” on page 6-53</td>
</tr>
<tr>
<td>0xFFE0 0300</td>
<td>DTEST_COMMAND</td>
<td>“Data Test Command (DTEST_COMMAND) Register” on page 6-40</td>
</tr>
<tr>
<td>0xFFE0 0400</td>
<td>DTEST_DATA0</td>
<td>“Data Test Data (DTEST_DATA0) Register” on page 6-42</td>
</tr>
<tr>
<td>0xFFE0 0404</td>
<td>DTEST_DATA1</td>
<td>“Data Test Data (DTEST_DATA1) Register” on page 6-41</td>
</tr>
</tbody>
</table>

L1 Instruction Memory Controller Registers

L1 instruction memory controller registers (0xFFE0 1004 – 0xFFE0 1404)

Table A-2. L1 Instruction Memory Controller Registers

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFE0 1004</td>
<td>IMEM_CONTROL</td>
<td>“Instruction Memory Control (IMEM_CONTROL) Register” on page 6-6</td>
</tr>
<tr>
<td>0xFFE0 1008</td>
<td>ICPLB_STATUS</td>
<td>“Instruction and Data CPLB Status (ICPLB_STATUS, DCPLB_STATUS) Registers” on page 6-59</td>
</tr>
<tr>
<td>0xFFE0 100C</td>
<td>ICPLB_FAULT_ADDR</td>
<td>“Instruction and Data CPLB Fault Address (ICPLB_FAULT_ADDR, DCPLB_FAULT_ADDR) Registers” on page 6-60</td>
</tr>
</tbody>
</table>
Table A-2. L1 Instruction Memory Controller Registers (Cont’d)

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFE0 1100</td>
<td>ICPLB_ADDR0</td>
<td>“Instruction CPLB Address (ICPLB_ADDRx) Registers” on page 6-57</td>
</tr>
<tr>
<td>0xFFE0 1104</td>
<td>ICPLB_ADDR1</td>
<td>“Instruction CPLB Address (ICPLB_ADDRx) Registers” on page 6-57</td>
</tr>
<tr>
<td>0xFFE0 1108</td>
<td>ICPLB_ADDR2</td>
<td>“Instruction CPLB Address (ICPLB_ADDRx) Registers” on page 6-57</td>
</tr>
<tr>
<td>0xFFE0 110C</td>
<td>ICPLB_ADDR3</td>
<td>“Instruction CPLB Address (ICPLB_ADDRx) Registers” on page 6-57</td>
</tr>
<tr>
<td>0xFFE0 1110</td>
<td>ICPLB_ADDR4</td>
<td>“Instruction CPLB Address (ICPLB_ADDRx) Registers” on page 6-57</td>
</tr>
<tr>
<td>0xFFE0 1114</td>
<td>ICPLB_ADDR5</td>
<td>“Instruction CPLB Address (ICPLB_ADDRx) Registers” on page 6-57</td>
</tr>
<tr>
<td>0xFFE0 1118</td>
<td>ICPLB_ADDR6</td>
<td>“Instruction CPLB Address (ICPLB_ADDRx) Registers” on page 6-57</td>
</tr>
<tr>
<td>0xFFE0 111C</td>
<td>ICPLB_ADDR7</td>
<td>“Instruction CPLB Address (ICPLB_ADDRx) Registers” on page 6-57</td>
</tr>
<tr>
<td>0xFFE0 1120</td>
<td>ICPLB_ADDR8</td>
<td>“Instruction CPLB Address (ICPLB_ADDRx) Registers” on page 6-57</td>
</tr>
<tr>
<td>0xFFE0 1124</td>
<td>ICPLB_ADDR9</td>
<td>“Instruction CPLB Address (ICPLB_ADDRx) Registers” on page 6-57</td>
</tr>
<tr>
<td>0xFFE0 1128</td>
<td>ICPLB_ADDR10</td>
<td>“Instruction CPLB Address (ICPLB_ADDRx) Registers” on page 6-57</td>
</tr>
<tr>
<td>0xFFE0 112C</td>
<td>ICPLB_ADDR11</td>
<td>“Instruction CPLB Address (ICPLB_ADDRx) Registers” on page 6-57</td>
</tr>
<tr>
<td>0xFFE0 1130</td>
<td>ICPLB_ADDR12</td>
<td>“Instruction CPLB Address (ICPLB_ADDRx) Registers” on page 6-57</td>
</tr>
<tr>
<td>0xFFE0 1134</td>
<td>ICPLB_ADDR13</td>
<td>“Instruction CPLB Address (ICPLB_ADDRx) Registers” on page 6-57</td>
</tr>
<tr>
<td>0xFFE0 1138</td>
<td>ICPLB_ADDR14</td>
<td>“Instruction CPLB Address (ICPLB_ADDRx) Registers” on page 6-57</td>
</tr>
</tbody>
</table>
### Table A-2. L1 Instruction Memory Controller Registers (Cont’d)

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFE0 113C</td>
<td>ICPLB_ADDR15</td>
<td>“Instruction CPLB Address (ICPLB_ADDRx) Registers” on page 6-57</td>
</tr>
<tr>
<td>0xFFE0 1200</td>
<td>ICPLB_DATA0</td>
<td>“Instruction CPLB Data (ICPLB_DATAx) Registers” on page 6-52</td>
</tr>
<tr>
<td>0xFFE0 1204</td>
<td>ICPLB_DATA1</td>
<td>“Instruction CPLB Data (ICPLB_DATAx) Registers” on page 6-52</td>
</tr>
<tr>
<td>0xFFE0 1208</td>
<td>ICPLB_DATA2</td>
<td>“Instruction CPLB Data (ICPLB_DATAx) Registers” on page 6-52</td>
</tr>
<tr>
<td>0xFFE0 120C</td>
<td>ICPLB_DATA3</td>
<td>“Instruction CPLB Data (ICPLB_DATAx) Registers” on page 6-52</td>
</tr>
<tr>
<td>0xFFE0 1210</td>
<td>ICPLB_DATA4</td>
<td>“Instruction CPLB Data (ICPLB_DATAx) Registers” on page 6-52</td>
</tr>
<tr>
<td>0xFFE0 1214</td>
<td>ICPLB_DATA5</td>
<td>“Instruction CPLB Data (ICPLB_DATAx) Registers” on page 6-52</td>
</tr>
<tr>
<td>0xFFE0 1218</td>
<td>ICPLB_DATA6</td>
<td>“Instruction CPLB Data (ICPLB_DATAx) Registers” on page 6-52</td>
</tr>
<tr>
<td>0xFFE0 121C</td>
<td>ICPLB_DATA7</td>
<td>“Instruction CPLB Data (ICPLB_DATAx) Registers” on page 6-52</td>
</tr>
<tr>
<td>0xFFE0 1220</td>
<td>ICPLB_DATA8</td>
<td>“Instruction CPLB Data (ICPLB_DATAx) Registers” on page 6-52</td>
</tr>
<tr>
<td>0xFFE0 1224</td>
<td>ICPLB_DATA9</td>
<td>“Instruction CPLB Data (ICPLB_DATAx) Registers” on page 6-52</td>
</tr>
<tr>
<td>0xFFE0 1228</td>
<td>ICPLB_DATA10</td>
<td>“Instruction CPLB Data (ICPLB_DATAx) Registers” on page 6-52</td>
</tr>
<tr>
<td>0xFFE0 122C</td>
<td>ICPLB_DATA11</td>
<td>“Instruction CPLB Data (ICPLB_DATAx) Registers” on page 6-52</td>
</tr>
<tr>
<td>0xFFE0 1230</td>
<td>ICPLB_DATA12</td>
<td>“Instruction CPLB Data (ICPLB_DATAx) Registers” on page 6-52</td>
</tr>
<tr>
<td>0xFFE0 1234</td>
<td>ICPLB_DATA13</td>
<td>“Instruction CPLB Data (ICPLB_DATAx) Registers” on page 6-52</td>
</tr>
</tbody>
</table>
Table A-2. L1 Instruction Memory Controller Registers (Cont’d)

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFFE0 1238</td>
<td>ICPLB_DATA14</td>
<td>“Instruction CPLB Data (ICPLB_DATAx) Registers” on page 6-52</td>
</tr>
<tr>
<td>0xFFFFE0 123C</td>
<td>ICPLB_DATA15</td>
<td>“Instruction CPLB Data (ICPLB_DATAx) Registers” on page 6-52</td>
</tr>
<tr>
<td>0xFFFFE0 1300</td>
<td>ITEST_COMMAND</td>
<td>“Instruction Test Command (ITEST_COMMAND) Register” on page 6-21</td>
</tr>
<tr>
<td>0xFFFFE0 1400</td>
<td>ITEST_DATA0</td>
<td>“Instruction Test Data 0 (ITEST_DATA0) Register” on page 6-23</td>
</tr>
<tr>
<td>0xFFFFE0 1404</td>
<td>ITEST_DATA1</td>
<td>“Instruction Test Data (ITEST_DATA1) Register” on page 6-22</td>
</tr>
</tbody>
</table>

Table A-3. Interrupt Controller Registers

Interrupt controller registers (0xFFFFE0 2000 – 0xFFFFE0 2110)

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFFE0 2000</td>
<td>EVT0 (EMU)</td>
<td>“Core Event Vector Table” on page 4-43</td>
</tr>
<tr>
<td>0xFFFFE0 2004</td>
<td>EVT1 (RST)</td>
<td>“Core Event Vector Table” on page 4-43</td>
</tr>
<tr>
<td>0xFFFFE0 2008</td>
<td>EVT2 (NMI)</td>
<td>“Core Event Vector Table” on page 4-43</td>
</tr>
<tr>
<td>0xFFFFE0 200C</td>
<td>EVT3 (EVX)</td>
<td>“Core Event Vector Table” on page 4-43</td>
</tr>
<tr>
<td>0xFFFFE0 2010</td>
<td>EVT4</td>
<td>“Core Event Vector Table” on page 4-43</td>
</tr>
<tr>
<td>0xFFFFE0 2014</td>
<td>EVT5 (IVHW)</td>
<td>“Core Event Vector Table” on page 4-43</td>
</tr>
</tbody>
</table>
## Interrupt Controller Registers

### Table A-3. Interrupt Controller Registers (Cont’d)

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFE0 2018</td>
<td>EVT6 (TMR)</td>
<td>“Core Event Vector Table” on page 4-43</td>
</tr>
<tr>
<td>0xFFE0 201C</td>
<td>EVT7 (IVG7)</td>
<td>“Core Event Vector Table” on page 4-43</td>
</tr>
<tr>
<td>0xFFE0 2020</td>
<td>EVT8 (IVG8)</td>
<td>“Core Event Vector Table” on page 4-43</td>
</tr>
<tr>
<td>0xFFE0 2024</td>
<td>EVT9 (IVG9)</td>
<td>“Core Event Vector Table” on page 4-43</td>
</tr>
<tr>
<td>0xFFE0 2028</td>
<td>EVT10 (IVG10)</td>
<td>“Core Event Vector Table” on page 4-43</td>
</tr>
<tr>
<td>0xFFE0 202C</td>
<td>EVT11 (IVG11)</td>
<td>“Core Event Vector Table” on page 4-43</td>
</tr>
<tr>
<td>0xFFE0 2030</td>
<td>EVT12 (IVG12)</td>
<td>“Core Event Vector Table” on page 4-43</td>
</tr>
<tr>
<td>0xFFE0 2034</td>
<td>EVT13 (IVG13)</td>
<td>“Core Event Vector Table” on page 4-43</td>
</tr>
<tr>
<td>0xFFE0 2038</td>
<td>EVT14 (IVG14)</td>
<td>“Core Event Vector Table” on page 4-43</td>
</tr>
<tr>
<td>0xFFE0 203C</td>
<td>EVT15 (IVG15)</td>
<td>“Core Event Vector Table” on page 4-43</td>
</tr>
<tr>
<td>0xFFE0 2104</td>
<td>IMASK</td>
<td>“System Interrupt Mask (SIC_IMASKx) Registers” on page 4-32</td>
</tr>
<tr>
<td>0xFFE0 2108</td>
<td>IPEND</td>
<td>“Core Interrupts Pending (IPEND) Register” on page 4-41</td>
</tr>
<tr>
<td>0xFFE0 2110</td>
<td>IPRIIO</td>
<td>“Interrupt Priority Register and Write Buffer Depth” on page 6-36</td>
</tr>
<tr>
<td>0xFFE0 210C</td>
<td>ILAT</td>
<td>“Core Interrupt Latch (ILAT) Register” on page 4-40</td>
</tr>
</tbody>
</table>
Core Timer Registers

Core timer registers (0xFFE0 3000 – 0xFFE0 300C)

Table A-4. Core Timer Registers

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFE0 3000</td>
<td>TCNTL</td>
<td>“TCNTL Register” on page 16-45</td>
</tr>
<tr>
<td>0xFFE0 3004</td>
<td>TPERIOD</td>
<td>“TPERIOD Register” on page 16-47</td>
</tr>
<tr>
<td>0xFFE0 3008</td>
<td>TSCALE</td>
<td>“TSCALE Register” on page 16-48</td>
</tr>
<tr>
<td>0xFFE0 300C</td>
<td>TCOUNT</td>
<td>“TCOUNT Register” on page 16-46</td>
</tr>
</tbody>
</table>

Debug, MP, and Emulation Unit Registers

Debug, MP, and emulation unit registers (0xFFE0 5000 – 0xFFE0 5008)

Table A-5. Debug and Emulation Unit Registers

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFE0 5000</td>
<td>DSPID</td>
<td>“DSP Device ID (DSPID) Register” on page 22-25</td>
</tr>
</tbody>
</table>
Trace Unit Registers

Trace unit registers (0xFFE0 6000 – 0xFFE0 6100)

Table A-6. Trace Unit Registers

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFE0 6000</td>
<td>TBUFCTL</td>
<td>“Trace Buffer Control (TBUFCTL) Register” on page 22-16</td>
</tr>
<tr>
<td>0xFFE0 6004</td>
<td>TBUFSTAT</td>
<td>“Trace Buffer Status (TBUFSTAT) Register” on page 22-17</td>
</tr>
<tr>
<td>0xFFE0 6100</td>
<td>TBUF</td>
<td>“Trace Buffer (TBUF) Register” on page 22-17</td>
</tr>
</tbody>
</table>

Watchpoint and Patch Registers

Watchpoint and patch registers (0xFFE0 7000 – 0xFFE0 7200)

Table A-7. Watchpoint and Patch Registers

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFE0 7000</td>
<td>WPIACTL</td>
<td>“Instruction Watchpoint Address Control (WPIACTL) Register” on page 22-7</td>
</tr>
<tr>
<td>0xFFE0 7040</td>
<td>WPIA0</td>
<td>“Instruction Watchpoint Address (WPIAn) Registers” on page 22-5</td>
</tr>
<tr>
<td>0xFFE0 7044</td>
<td>WPIA1</td>
<td>“Instruction Watchpoint Address (WPIAn) Registers” on page 22-5</td>
</tr>
<tr>
<td>0xFFE0 7048</td>
<td>WPIA2</td>
<td>“Instruction Watchpoint Address (WPIAn) Registers” on page 22-5</td>
</tr>
<tr>
<td>0xFFE0 704C</td>
<td>WPIA3</td>
<td>“Instruction Watchpoint Address (WPIAn) Registers” on page 22-5</td>
</tr>
</tbody>
</table>
### Table A-7. Watchpoint and Patch Registers (Cont'd)

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFE0 7050</td>
<td>WPIA4</td>
<td>“Instruction Watchpoint Address (WPIAn) Registers” on page 22-5</td>
</tr>
<tr>
<td>0xFFE0 7054</td>
<td>WPIA5</td>
<td>“Instruction Watchpoint Address (WPIAn) Registers” on page 22-5</td>
</tr>
<tr>
<td>0xFFE0 7080</td>
<td>WPIACNT0</td>
<td>“Instruction Watchpoint Address Count (WPIACNTn) Registers” on page 22-6</td>
</tr>
<tr>
<td>0xFFE0 7084</td>
<td>WPIACNT1</td>
<td>“Instruction Watchpoint Address Count (WPIACNTn) Registers” on page 22-6</td>
</tr>
<tr>
<td>0xFFE0 7088</td>
<td>WPIACNT2</td>
<td>“Instruction Watchpoint Address Count (WPIACNTn) Registers” on page 22-6</td>
</tr>
<tr>
<td>0xFFE0 708C</td>
<td>WPIACNT3</td>
<td>“Instruction Watchpoint Address Count (WPIACNTn) Registers” on page 22-6</td>
</tr>
<tr>
<td>0xFFE0 7090</td>
<td>WPIACNT4</td>
<td>“Instruction Watchpoint Address Count (WPIACNTn) Registers” on page 22-6</td>
</tr>
<tr>
<td>0xFFE0 7094</td>
<td>WPIACNT5</td>
<td>“Instruction Watchpoint Address Count (WPIACNTn) Registers” on page 22-6</td>
</tr>
<tr>
<td>0xFFE0 7100</td>
<td>WPDACTL</td>
<td>“Data Watchpoint Address Control (WPDACTL) Register” on page 22-12</td>
</tr>
<tr>
<td>0xFFE0 7140</td>
<td>WPDA0</td>
<td>“Data Watchpoint Address (WPDAAn) Registers” on page 22-11</td>
</tr>
<tr>
<td>0xFFE0 7144</td>
<td>WPDA1</td>
<td>“Data Watchpoint Address (WPDAAn) Registers” on page 22-11</td>
</tr>
<tr>
<td>0xFFE0 7180</td>
<td>WPDACNT0</td>
<td>“Data Watchpoint Address Count Value (WPDACNTn) Registers” on page 22-11</td>
</tr>
<tr>
<td>0xFFE0 7184</td>
<td>WPDACNT1</td>
<td>“Data Watchpoint Address Count Value (WPDACNTn) Registers” on page 22-11</td>
</tr>
<tr>
<td>0xFFE0 7200</td>
<td>WPSTAT</td>
<td>“Watchpoint Status (WPSTAT) Register” on page 22-12</td>
</tr>
</tbody>
</table>
Performance Monitor Registers

Performance monitor registers (0xFFE0 8000 – 0xFFE0 8104)

Table A-8. Performance Monitor Registers

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFE0 8000</td>
<td>PFCTL</td>
<td>“Performance Monitor Control (PFCTL) Register” on page 22-20</td>
</tr>
<tr>
<td>0xFFE0 8100</td>
<td>PFCNTR0</td>
<td>“Performance Monitor Counter (PFCNTRn) Registers” on page 22-19</td>
</tr>
<tr>
<td>0xFFE0 8104</td>
<td>PFCNTR1</td>
<td>“Performance Monitor Counter (PFCNTRn) Registers” on page 22-19</td>
</tr>
</tbody>
</table>
B  SYSTEM MMR ASSIGNMENTS

These notes provide general information about the system memory-mapped registers (MMRs):

- The system MMR address range is 0xFFC0 0000 – 0xFFFF FFFF.

- All system MMRs are either 16 bits or 32 bits wide. MMRs that are 16 bits wide must be accessed with 16-bit read or write operations. MMRs that are 32 bits wide must be accessed with 32-bit read or write operations. Check the description of the MMR to determine whether a 16-bit or a 32-bit access is required.

- All system MMR space that is not defined in this appendix is reserved for internal use only.

This appendix lists MMR addresses and register names. To find more information about an MMR, refer to the page shown in the “See Section” column. When viewing the PDF version of this document, click a reference in the “See Section” column to jump to additional information about the MMR.
Dynamic Power Management Registers

Dynamic power management registers (0xFFC0 0000 – 0xFFC0 00FF)

Table B-1. Dynamic Power Management Registers

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0 0000</td>
<td>PLL_CTL</td>
<td>“PLL Control (PLL_CTL) Register” on page 8-7</td>
</tr>
<tr>
<td>0xFFC0 0004</td>
<td>PLL_DIV</td>
<td>“PLL Divide (PLL_DIV) Register” on page 8-6</td>
</tr>
<tr>
<td>0xFFC0 0008</td>
<td>VR_CTL</td>
<td>“Voltage Regulator Control (VR_CTL) Register” on page 8-25</td>
</tr>
<tr>
<td>0xFFC0 000C</td>
<td>PLL_STAT</td>
<td>“PLL Status (PLL_STAT) Register” on page 8-9</td>
</tr>
<tr>
<td>0xFFC0 0010</td>
<td>PLL_LOCKCNT</td>
<td>“PLL Lock Count (PLL_LOCKCNT) Register” on page 8-10</td>
</tr>
<tr>
<td>0xFFC0 0014</td>
<td>CHIPID</td>
<td>NA</td>
</tr>
</tbody>
</table>

System Reset and Interrupt Control Registers

System reset and interrupt controller registers (0xFFC0 0100 – 0xFFC0 01FF)

Table B-2. System Interrupt Controller Registers

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0 0100</td>
<td>SWRST</td>
<td>“SWRST Register” on page 3-16</td>
</tr>
<tr>
<td>0xFFC0 0104</td>
<td>SYSCR</td>
<td>“SYSCR Register” on page 3-14</td>
</tr>
<tr>
<td>0xFFC0 010C 0xFFC0 0128</td>
<td>SIC_IMASK0  SIC_IMASK1</td>
<td>“System Interrupt Mask (SIC_IMASKx) Registers” on page 4-32</td>
</tr>
</tbody>
</table>
System MMR Assignments

Table B-2. System Interrupt Controller Registers (Cont’d)

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0 0110</td>
<td>SIC_IAR0</td>
<td></td>
</tr>
<tr>
<td>0xFFC0 0114</td>
<td>SIC_IAR1</td>
<td></td>
</tr>
<tr>
<td>0xFFC0 0118</td>
<td>SIC_IAR2</td>
<td></td>
</tr>
<tr>
<td>0xFFC0 011C</td>
<td>SIC_IAR3</td>
<td></td>
</tr>
<tr>
<td>0xFFC0 0134</td>
<td>SIC_IAR4</td>
<td></td>
</tr>
<tr>
<td>0xFFC0 0138</td>
<td>SIC_IAR5</td>
<td></td>
</tr>
<tr>
<td>0xFFC0 013C</td>
<td>SIC_IAR6</td>
<td></td>
</tr>
<tr>
<td>0xFFC0 0140</td>
<td>SIC_IAR7</td>
<td></td>
</tr>
<tr>
<td>0xFFC0 0120</td>
<td>SIC_ISR0</td>
<td>“System Interrupt Status (SIC_ISRx) Registers” on page 4-29</td>
</tr>
<tr>
<td>0xFFC0 012C</td>
<td>SIC_ISR1</td>
<td></td>
</tr>
<tr>
<td>0xFFC0 0124</td>
<td>SIC_IWR0</td>
<td>“System Interrupt Wake-Up Enable (SIC_IWRx) Registers” on page 4-27</td>
</tr>
<tr>
<td>0xFFC0 0130</td>
<td>SIC_IWR1</td>
<td></td>
</tr>
</tbody>
</table>

Watchdog Timer Registers

Watchdog timer registers (0xFFC0 0200 – 0xFFC0 02FF)

Table B-3. Watchdog Timer Registers

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0 0200</td>
<td>WDOG_CTL</td>
<td>“WDOG_CTL Register” on page 16-52</td>
</tr>
<tr>
<td>0xFFC0 0204</td>
<td>WDOG_CNT</td>
<td>“WDOG_CNT Register” on page 16-49</td>
</tr>
<tr>
<td>0xFFC0 0208</td>
<td>WDOG_STAT</td>
<td>“WDOG_STAT Register” on page 16-50</td>
</tr>
</tbody>
</table>
Real-Time Clock Registers

Real-time clock registers (0xFFFFC0 0300 – 0xFFFFC0 03FF)

Table B-4. Real-Time Clock Registers

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFFC0 0300</td>
<td>RTC_STAT</td>
<td>“RTC Status (RTC_STAT) Register” on page 17-13</td>
</tr>
<tr>
<td>0xFFFFC0 0304</td>
<td>RTC_ICTL</td>
<td>“RTC Interrupt Control (RTC_ICTL) Register” on page 17-14</td>
</tr>
<tr>
<td>0xFFFFC0 0308</td>
<td>RTC_ISTAT</td>
<td>“RTC Interrupt Status (RTC_ISTAT) Register” on page 17-15</td>
</tr>
<tr>
<td>0xFFFFC0 030C</td>
<td>RTC_SWCNT</td>
<td>“RTC Stopwatch Count (RTC_SWCNT) Register” on page 17-16</td>
</tr>
<tr>
<td>0xFFFFC0 0310</td>
<td>RTC_ALARM</td>
<td>“RTC Alarm (RTC_ALARM) Register” on page 17-18</td>
</tr>
<tr>
<td>0xFFFFC0 0314</td>
<td>RTC_PREN</td>
<td>“RTC Prescaler Enable (RTC_PREN) Register” on page 17-18</td>
</tr>
</tbody>
</table>

Parallel Peripheral Interface (PPI) Registers

Parallel peripheral interface (PPI) registers (0xFFFFC0 1000 – 0xFFFFC0 10FF)

Table B-5. Parallel Peripheral Interface (PPI) Registers

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFFC0 1000</td>
<td>PPI_CONTROL</td>
<td>“PPI_CONTROL Register” on page 11-3</td>
</tr>
<tr>
<td>0xFFFFC0 1004</td>
<td>PPI_STATUS</td>
<td>“PPI_STATUS Register” on page 11-8</td>
</tr>
</tbody>
</table>
System MMR Assignments

Table B-5. Parallel Peripheral Interface (PPI) Registers (Cont’d)

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0 1008</td>
<td>PPI_COUNT</td>
<td>“PPI_COUNT Register” on page 11-11</td>
</tr>
<tr>
<td>0xFFC0 100C</td>
<td>PPI_DELAY</td>
<td>“PPI_DELAY Register” on page 11-9</td>
</tr>
<tr>
<td>0xFFC0 1010</td>
<td>PPI_FRAME</td>
<td>“PPI_FRAME Register” on page 11-12</td>
</tr>
</tbody>
</table>

UART Controller Registers

UART0 controller registers (0xFFC0 0400 – 0xFFC0 04FF)
UART1 controller registers (0xFFC0 2000 – 0xFFC0 20FF)
UART2 controller registers (0xFFC0 2100 – 0xFFC0 21FF)

Table B-6. UART0 Controller Registers

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0 0400</td>
<td>UART0_THR</td>
<td>“UART Transmit Holding (UARTx_THR) Register” on page 12-6</td>
</tr>
<tr>
<td>0xFFC0 0400</td>
<td>UART0_RBR</td>
<td>“UART Receive Buffer (UARTx_RBR) Register” on page 12-7</td>
</tr>
<tr>
<td>0xFFC0 0400</td>
<td>UART0_DLL</td>
<td>“UARTx_DLL and UARTx_DLH Registers” on page 12-12</td>
</tr>
<tr>
<td>0xFFC0 0404</td>
<td>UART0_DLH</td>
<td>“UARTx_DLL and UARTx_DLH Registers” on page 12-12</td>
</tr>
<tr>
<td>0xFFC0 0404</td>
<td>UART0_IER</td>
<td>“UART Interrupt Enable (UARTx_IER) Register” on page 12-8</td>
</tr>
<tr>
<td>0xFFC0 0408</td>
<td>UART0_IIR</td>
<td>“UART Interrupt Identification (UARTx_IIR) Register” on page 12-10</td>
</tr>
<tr>
<td>0xFFC0 040C</td>
<td>UART0_LCR</td>
<td>“UART Line Control (UARTx_LCR) Register” on page 12-3</td>
</tr>
<tr>
<td>0xFFC0 0410</td>
<td>UART0_MCR</td>
<td>“UART Modem Control (UARTx_MCR) Register” on page 12-4</td>
</tr>
</tbody>
</table>
Table B-6. UART0 Controller Registers (Cont’d)

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0 0414</td>
<td>UART0_LSR</td>
<td>“UART Line Status (UARTx_LSR) Register” on page 12-5</td>
</tr>
<tr>
<td>0xFFC0 041C</td>
<td>UART0_SCR</td>
<td>“UART Scratch (UARTx_SCR) Register” on page 12-14</td>
</tr>
<tr>
<td>0xFFC0 0424</td>
<td>UART0_GCTL</td>
<td>“UART Global Control (UARTx_GCTL) Register” on page 12-14</td>
</tr>
</tbody>
</table>

Table B-7. UART1 Controller Registers

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC 02000</td>
<td>UART1_THR</td>
<td>“UART Transmit Holding (UARTx_THR) Register” on page 12-6</td>
</tr>
<tr>
<td>0xFFC 02000</td>
<td>UART1_RBR</td>
<td>“UART Receive Buffer (UARTx_RBR) Register” on page 12-7</td>
</tr>
<tr>
<td>0xFFC 02000</td>
<td>UART1_DLL</td>
<td>“UARTx_DLL and UARTx_DLH Registers” on page 12-12</td>
</tr>
<tr>
<td>0xFFC0 2004</td>
<td>UART1_DLH</td>
<td>“UARTx_DLL and UARTx_DLH Registers” on page 12-12</td>
</tr>
<tr>
<td>0xFFC0 2004</td>
<td>UART1_IER</td>
<td>“UART Interrupt Enable (UARTx_IER) Register” on page 12-8</td>
</tr>
<tr>
<td>0xFFC0 2008</td>
<td>UART1_IIR</td>
<td>“UART Interrupt Identification (UARTx_IIR) Register” on page 12-10</td>
</tr>
<tr>
<td>0xFFC0 200C</td>
<td>UART1_LCR</td>
<td>“UART Line Control (UARTx_LCR) Register” on page 12-3</td>
</tr>
<tr>
<td>0xFFC0 2010</td>
<td>UART1_MCR</td>
<td>“UART Modem Control (UARTx_MCR) Register” on page 12-4</td>
</tr>
<tr>
<td>0xFFC0 2014</td>
<td>UART1_LSR</td>
<td>“UART Line Status (UARTx_LSR) Register” on page 12-5</td>
</tr>
</tbody>
</table>
Table B-7. UART1 Controller Registers (Cont’d)

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFF0 201C</td>
<td>UART1_SCR</td>
<td>“UART Scratch (UARTx_SCR) Register” on page 12-14</td>
</tr>
<tr>
<td>0xFFF0 2024</td>
<td>UART1_GCTL</td>
<td>“UART Global Control (UARTx_GCTL) Register” on page 12-14</td>
</tr>
</tbody>
</table>

Table B-8. UART2 Controller Registers

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFF0 2100</td>
<td>UART2_THR</td>
<td>“UART Transmit Holding (UARTx_THR) Register” on page 12-6</td>
</tr>
<tr>
<td>0xFFF0 2100</td>
<td>UART2_RBR</td>
<td>“UART Receive Buffer (UARTx_RBR) Register” on page 12-7</td>
</tr>
<tr>
<td>0xFFF0 2100</td>
<td>UART2_DLL</td>
<td>“UARTx_DLL and UARTx_DLH Registers” on page 12-12</td>
</tr>
<tr>
<td>0xFFF0 2104</td>
<td>UART2_DLH</td>
<td>“UARTx_DLL and UARTx_DLH Registers” on page 12-12</td>
</tr>
<tr>
<td>0xFFF0 2104</td>
<td>UART2_IER</td>
<td>“UART Interrupt Enable (UARTx_IER) Register” on page 12-8</td>
</tr>
<tr>
<td>0xFFF0 2108</td>
<td>UART2_IIR</td>
<td>“UART Interrupt Identification (UARTx_IIR) Register” on page 12-10</td>
</tr>
<tr>
<td>0xFFF0 210C</td>
<td>UART2_LCR</td>
<td>“UART Line Control (UARTx_LCR) Register” on page 12-3</td>
</tr>
<tr>
<td>0xFFF0 2110</td>
<td>UART2_MCR</td>
<td>“UART Modem Control (UARTx_MCR) Register” on page 12-4</td>
</tr>
<tr>
<td>0xFFF0 2114</td>
<td>UART2_LSR</td>
<td>“UART Line Status (UARTx_LSR) Register” on page 12-5</td>
</tr>
<tr>
<td>0xFFF0 211C</td>
<td>UART2_SCR</td>
<td>“UART Scratch (UARTx_SCR) Register” on page 12-14</td>
</tr>
<tr>
<td>0xFFF0 2124</td>
<td>UART2_GCTL</td>
<td>“UART Global Control (UARTx_GCTL) Register” on page 12-14</td>
</tr>
</tbody>
</table>
## SPI Controller Registers

SPI0 controller registers (0xFFC0 0500 – 0xFFC0 05FF)
SPI1 controller registers (0xFFC0 2300 – 0xFFC0 23FF)
SPI2 controller registers (0xFFC0 2400 – 0xFFC0 024FF)

### Table B-9. SPI0 Controller Registers

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0 0500</td>
<td>SPI0_CTL</td>
<td>“SPI Control (SPIx_CTL) Register” on page 10-9</td>
</tr>
<tr>
<td>0xFFC0 0504</td>
<td>SPI0_FLG</td>
<td>“SPI Flag (SPIx_FLG) Register” on page 10-11</td>
</tr>
<tr>
<td>0xFFC0 0508</td>
<td>SPI0_STAT</td>
<td>“SPI Status (SPIx_STAT) Register” on page 10-15</td>
</tr>
<tr>
<td>0xFFC0 050C</td>
<td>SPI0_TDBR</td>
<td>“SPI Transmit Data Buffer (SPIx_TDBR) Register” on page 10-17</td>
</tr>
<tr>
<td>0xFFC0 0510</td>
<td>SPI0_RDBR</td>
<td>“SPI Receive Data Buffer (SPIx_RDBR) Register” on page 10-18</td>
</tr>
<tr>
<td>0xFFC0 0514</td>
<td>SPI0_BAUD</td>
<td>“SPI BAUD Rate (SPIx_BAUD) Register” on page 10-8</td>
</tr>
<tr>
<td>0xFFC0 0518</td>
<td>SPI0_SHADOW</td>
<td>“SPI Receive Data Buffer Shadow (SPIx_SHADOW) Register” on page 10-19</td>
</tr>
</tbody>
</table>

### Table B-10. SPI1 Controller Registers

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0 2300</td>
<td>SPI1_CTL</td>
<td>“SPI Control (SPIx_CTL) Register” on page 10-9</td>
</tr>
<tr>
<td>0xFFC0 2304</td>
<td>SPI1_FLG</td>
<td>“SPI Flag (SPIx_FLG) Register” on page 10-11</td>
</tr>
<tr>
<td>0xFFC0 2308</td>
<td>SPI1_STAT</td>
<td>“SPI Status (SPIx_STAT) Register” on page 10-15</td>
</tr>
<tr>
<td>0xFFC0 230C</td>
<td>SPI1_TDBR</td>
<td>“SPI Transmit Data Buffer (SPIx_TDBR) Register” on page 10-17</td>
</tr>
</tbody>
</table>
Table B-10. SPI1 Controller Registers (Cont’d)

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0 2310</td>
<td>SPI1_RDBR</td>
<td>“SPI Receive Data Buffer (SPIx_RDBR) Register” on page 10-18</td>
</tr>
<tr>
<td>0xFFC0 2314</td>
<td>SPI1_BAUD</td>
<td>“SPI BAUD Rate (SPIx_BAUD) Register” on page 10-8</td>
</tr>
<tr>
<td>0xFFC0 2318</td>
<td>SPI1_SHADOW</td>
<td>“SPI Receive Data Buffer Shadow (SPIx_SHADOW) Register” on page 10-19</td>
</tr>
</tbody>
</table>

Table B-11. SPI2 Controller Registers

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0 2400</td>
<td>SPI2_CTL</td>
<td>“SPI Control (SPIx_CTL) Register” on page 10-9</td>
</tr>
<tr>
<td>0xFFC0 2404</td>
<td>SPI2_FLG</td>
<td>“SPI Flag (SPIx_FLG) Register” on page 10-11</td>
</tr>
<tr>
<td>0xFFC0 2408</td>
<td>SPI2_STAT</td>
<td>“SPI Status (SPIx_STAT) Register” on page 10-15</td>
</tr>
<tr>
<td>0xFFC0 240C</td>
<td>SPI2_TDBR</td>
<td>“SPI Transmit Data Buffer (SPIx_TDBR) Register” on page 10-17</td>
</tr>
<tr>
<td>0xFFC0 2410</td>
<td>SPI2_RDBR</td>
<td>“SPI Receive Data Buffer (SPIx_RDBR) Register” on page 10-18</td>
</tr>
<tr>
<td>0xFFC0 2414</td>
<td>SPI2_BAUD</td>
<td>“SPI BAUD Rate (SPIx_BAUD) Register” on page 10-8</td>
</tr>
<tr>
<td>0xFFC0 2418</td>
<td>SPI2_SHADOW</td>
<td>“SPI Receive Data Buffer Shadow (SPIx_SHADOW) Register” on page 10-19</td>
</tr>
</tbody>
</table>
# Timer Registers

Timer registers (0xFFC0 0600 – 0xFFC0 06FF)

Table B-12. Timer Registers

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0 0600</td>
<td>TIMER0_CONFIG</td>
<td>“TIMERx_CONFIG Registers” on page 16-8</td>
</tr>
<tr>
<td>0xFFC0 0604</td>
<td>TIMER0_COUNTER</td>
<td>“TIMERx_COUNTER Registers” on page 16-9</td>
</tr>
<tr>
<td>0xFFC0 0608</td>
<td>TIMER0_PERIOD</td>
<td>“TIMERx_PERIOD and TIMERx_WIDTH Registers” on page 16-10</td>
</tr>
<tr>
<td>0xFFC0 060C</td>
<td>TIMER0_WIDTH</td>
<td>“TIMERx_PERIOD and TIMERx_WIDTH Registers” on page 16-10</td>
</tr>
<tr>
<td>0xFFC0 0610</td>
<td>TIMER1_CONFIG</td>
<td>“TIMERx_CONFIG Registers” on page 16-8</td>
</tr>
<tr>
<td>0xFFC0 0614</td>
<td>TIMER1_COUNTER</td>
<td>“TIMERx_COUNTER Registers” on page 16-9</td>
</tr>
<tr>
<td>0xFFC0 0618</td>
<td>TIMER1_PERIOD</td>
<td>“TIMERx_PERIOD and TIMERx_WIDTH Registers” on page 16-10</td>
</tr>
<tr>
<td>0xFFC0 061C</td>
<td>TIMER1_WIDTH</td>
<td>“TIMERx_PERIOD and TIMERx_WIDTH Registers” on page 16-10</td>
</tr>
<tr>
<td>0xFFC0 0620</td>
<td>TIMER2_CONFIG</td>
<td>“TIMERx_CONFIG Registers” on page 16-8</td>
</tr>
<tr>
<td>0xFFC0 0624</td>
<td>TIMER2_COUNTER</td>
<td>“TIMERx_COUNTER Registers” on page 16-9</td>
</tr>
<tr>
<td>0xFFC0 0628</td>
<td>TIMER2_PERIOD</td>
<td>“TIMERx_PERIOD and TIMERx_WIDTH Registers” on page 16-10</td>
</tr>
<tr>
<td>0xFFC0 062C</td>
<td>TIMER2_WIDTH</td>
<td>“TIMERx_PERIOD and TIMERx_WIDTH Registers” on page 16-10</td>
</tr>
<tr>
<td>0xFFC0 0640</td>
<td>TIMER_ENABLE</td>
<td>“TIMER_ENABLE Register” on page 16-4</td>
</tr>
<tr>
<td>0xFFC0 0644</td>
<td>TIMER_DISABLE</td>
<td>“TIMER_DISABLE Register” on page 16-5</td>
</tr>
<tr>
<td>0xFFC0 0648</td>
<td>TIMER_STATUS</td>
<td>“TIMER_STATUS Register” on page 16-6</td>
</tr>
</tbody>
</table>
## GPIO Port C, D, and E Registers

GPIO port C, D, E registers (0xFFFF0 1500 – 0xFFFF0 15FF)

### Table B-13. GPIO Port C Registers

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFFC0 1500</td>
<td>PORTCIO_FER</td>
<td>“GPIO Function Enable (PORTxIO_FER) Register” on page 15-5</td>
</tr>
<tr>
<td>0xFFFFC0 1550</td>
<td>PORTCIO_DIR</td>
<td>“GPIO Direction (PORTxIO_DIR) Register” on page 15-6</td>
</tr>
<tr>
<td>0xFFFFC0 1560</td>
<td>PORTCIO_INEN</td>
<td>“GPIO Input Enable (PORTxIO_INEN) Register” on page 15-9</td>
</tr>
<tr>
<td>0xFFFFC0 1510</td>
<td>PORTCIO</td>
<td>“GPIO Data (PORTxIO) Register” on page 15-12</td>
</tr>
<tr>
<td>0xFFFFC0 1520</td>
<td>PORTCIO_CLEAR,</td>
<td>“GPIO Set (PORTxIO_SET), GPIO Clear (PORTxIO_CLEAR), and GPIO Toggle (PORTxIO_TOGGLE) Registers” on page 15-13</td>
</tr>
<tr>
<td>0xFFFFC0 1530</td>
<td>PORTCIO_SET,</td>
<td></td>
</tr>
<tr>
<td>0xFFFFC0 1540</td>
<td>PORTCIO_TOGGLE</td>
<td></td>
</tr>
</tbody>
</table>

### Table B-14. GPIO Port D Registers

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFFC0 1504</td>
<td>PORTDIO_FER</td>
<td>“GPIO Function Enable (PORTxIO_FER) Register” on page 15-5</td>
</tr>
<tr>
<td>0xFFFFC0 1554</td>
<td>PORTDIO_DIR</td>
<td>“GPIO Direction (PORTxIO_DIR) Register” on page 15-6</td>
</tr>
<tr>
<td>0xFFFFC0 1564</td>
<td>PORTDIO_INEN</td>
<td>“GPIO Input Enable (PORTxIO_INEN) Register” on page 15-9</td>
</tr>
</tbody>
</table>
### GPIO Port C, D, and E Registers

#### Table B-14. GPIO Port D Registers (Cont'd)

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0 1514</td>
<td>PORTDIO</td>
<td>“GPIO Data (PORTxIO) Register” on page 15-12</td>
</tr>
<tr>
<td>0xFFC0 1524 0xFFC0 1534 0xFFC0 1544</td>
<td>PORTDIO_CLEAR, PORTDIO_SET, PORTDIO_TOGGLE</td>
<td>“GPIO Set (PORTxIO_SET), GPIO Clear (PORTxIO_CLEAR), and GPIO Toggle (PORTxIO_TOGGLE) Registers” on page 15-13</td>
</tr>
</tbody>
</table>

#### Table B-15. GPIO Port E Registers

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0 1508</td>
<td>PORTEIO_FER</td>
<td>“GPIO Function Enable (PORTxIO_FER) Register” on page 15-5</td>
</tr>
<tr>
<td>0xFFC0 1558</td>
<td>PORTEIO_DIR</td>
<td>“GPIO Direction (PORTxIO_DIR) Register” on page 15-6</td>
</tr>
<tr>
<td>0xFFC0 1568</td>
<td>PORTEIO_INEN</td>
<td>“GPIO Input Enable (PORTxIO_INEN) Register” on page 15-9</td>
</tr>
<tr>
<td>0xFFC0 1518</td>
<td>PORTEIO</td>
<td>“GPIO Data (PORTxIO) Register” on page 15-12</td>
</tr>
<tr>
<td>0xFFC0 1528 0xFFC0 1538 0xFFC0 1548</td>
<td>PORTEIO_CLEAR, PORTEIO_SET, PORTEIO_TOGGLE</td>
<td>“GPIO Set (PORTxIO_SET), GPIO Clear (PORTxIO_CLEAR), and GPIO Toggle (PORTxIO_TOGGLE) Registers” on page 15-13</td>
</tr>
</tbody>
</table>
## GPIO Port F Registers

GPIO port F registers (0xFFC0 0700 – 0xFFC0 07FF)

### Table B-16. GPIO Port F Registers

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0 0700</td>
<td>PORTFIO_FLAG_D</td>
<td>“GPIO Port F Data (PORTFIO) Register” on page 14-8</td>
</tr>
<tr>
<td>0xFFC0 0704</td>
<td>PORTFIO_FLAG_CLEAR</td>
<td>“GPIO Port F Set (PORTFIO_SET), GPIO Port F Clear (PORTFIO_CLEAR), and GPIO Port F Toggle (PORTFIO_TOGGLE) Registers” on page 14-8</td>
</tr>
<tr>
<td>0xFFC0 0708</td>
<td>PORTFIO_FLAG_SET</td>
<td>“GPIO Port F Set (PORTFIO_SET), GPIO Port F Clear (PORTFIO_CLEAR), and GPIO Port F Toggle (PORTFIO_TOGGLE) Registers” on page 14-8</td>
</tr>
<tr>
<td>0xFFC0 070C</td>
<td>PORTFIO_FLAG_TOGGLE</td>
<td>“GPIO Port F Set (PORTFIO_SET), GPIO Port F Clear (PORTFIO_CLEAR), and GPIO Port F Toggle (PORTFIO_TOGGLE) Registers” on page 14-8</td>
</tr>
<tr>
<td>0xFFC0 0710</td>
<td>PORTFIO_MASKA_D</td>
<td>“GPIO Port F Mask Interrupt Registers Overview” on page 14-11</td>
</tr>
<tr>
<td>0xFFC0 0714</td>
<td>PORTFIO_MASKA_CLEAR</td>
<td>“GPIO Port F Mask Interrupt Registers Overview” on page 14-11</td>
</tr>
<tr>
<td>0xFFC0 0718</td>
<td>PORTFIO_MASKA_SET</td>
<td>“GPIO Port F Mask Interrupt Registers Overview” on page 14-11</td>
</tr>
<tr>
<td>0xFFC0 071C</td>
<td>PORTFIO_MASKA_TOGGLE</td>
<td>“GPIO Port F Mask Interrupt Registers Overview” on page 14-11</td>
</tr>
<tr>
<td>0xFFC0 0720</td>
<td>PORTFIO_MASKB_D</td>
<td>“GPIO Port F Mask Interrupt Registers Overview” on page 14-11</td>
</tr>
<tr>
<td>0xFFC0 0724</td>
<td>PORTFIO_MASKB_CLEAR</td>
<td>“GPIO Port F Mask Interrupt Registers Overview” on page 14-11</td>
</tr>
</tbody>
</table>
### SPORT Controller Registers

#### SPORT Controller Registers

**SPORT0 registers (0xFFC0 0800 – 0xFFC0 08FF)**

**SPORT1 registers (0xFFC0 0900 – 0xFFC0 09FF)**

**SPORT2 registers (0xFFC0 2500 – 0xFFC0 25FF)**

**SPORT3 registers (0xFFC0 2600 – 0xFFC0 26FF)**

#### Table B-16. GPIO Port F Registers (Cont’d)

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0 0728</td>
<td>PORTFIO_MASKB_SET</td>
<td>“GPIO Port F Mask Interrupt Registers Overview” on page 14-11</td>
</tr>
<tr>
<td>0xFFC0 072C</td>
<td>PORTFIO_MASKB_TOGGLE</td>
<td>“GPIO Port F Mask Interrupt Registers Overview” on page 14-11</td>
</tr>
<tr>
<td>0xFFC0 0730</td>
<td>PORTFIO_DIR</td>
<td>“GPIO Port F Direction (PORTFIO_DIR) Register” on page 14-5</td>
</tr>
<tr>
<td>0xFFC0 0734</td>
<td>PORTFIO_POLAR</td>
<td>“GPIO Port F Polarity (PORTFIO_POLAR) Register” on page 14-18</td>
</tr>
<tr>
<td>0xFFC0 0738</td>
<td>PORTFIO_EDGE</td>
<td>“GPIO Port F Interrupt Sensitivity (PORTFIO_EDGE) Register” on page 14-19</td>
</tr>
<tr>
<td>0xFFC0 073C</td>
<td>PORTFIO_BOTH</td>
<td>“GPIO Port F Set on Both Edges (PORTFIO_BOTH) Register” on page 14-20</td>
</tr>
<tr>
<td>0xFFC0 0740</td>
<td>PORTFIO_INEN</td>
<td>“GPIO Port F Input Enable (PORTFIO_INEN) Register” on page 14-21</td>
</tr>
</tbody>
</table>
Table B-17. SPORT0 Controller Registers

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0 0800</td>
<td>SPORT0_TCR1</td>
<td>“SPORT Transmit Configuration (SPORTx_TCR1, SPORTx_TCR2) Registers” on page 13-11</td>
</tr>
<tr>
<td>0xFFC0 0804</td>
<td>SPORT0_TCR2</td>
<td>“SPORT Transmit Configuration (SPORTx_TCR1, SPORTx_TCR2) Registers” on page 13-11</td>
</tr>
<tr>
<td>0xFFC0 0808</td>
<td>SPORT0_TCLKDIV</td>
<td>“SPORT Transmit Serial Clock Divider (SPORTx_TCLKDIV, SPORTx_RCLKDIV) Registers” on page 13-31</td>
</tr>
<tr>
<td>0xFFC0 080C</td>
<td>SPORT0_TFSDIV</td>
<td>“SPORT Transmit Frame Sync Divider (SPORTx_TFSDIV, SPORTx_RFSDIV) Register” on page 13-32</td>
</tr>
<tr>
<td>0xFFC0 0810</td>
<td>SPORT0_TX</td>
<td>“SPORT Transmit Data (SPORTx_TX) Register” on page 13-23</td>
</tr>
<tr>
<td>0xFFC0 0818</td>
<td>SPORT0_RX</td>
<td>“SPORT Receive Data (SPORTx_RX) Register” on page 13-25</td>
</tr>
<tr>
<td>0xFFC0 0820</td>
<td>SPORT0_RCR1</td>
<td>“SPORT Receive Configuration (SPORTx_RCR1, SPORTx_RCR2) Registers” on page 13-18</td>
</tr>
<tr>
<td>0xFFC0 0824</td>
<td>SPORT0_RCR2</td>
<td>“SPORT Receive Configuration (SPORTx_RCR1, SPORTx_RCR2) Registers” on page 13-18</td>
</tr>
<tr>
<td>0xFFC0 0828</td>
<td>SPORT0_RCLKDIV</td>
<td>“SPORT Transmit Serial Clock Divider (SPORTx_TCLKDIV, SPORTx_RCLKDIV) Registers” on page 13-31</td>
</tr>
<tr>
<td>0xFFC0 082C</td>
<td>SPORT0_RFSDIV</td>
<td>“SPORT Transmit Frame Sync Divider (SPORTx_TFSDIV, SPORTx_RFSDIV) Register” on page 13-32</td>
</tr>
<tr>
<td>0xFFC0 0830</td>
<td>SPORT0_STAT</td>
<td>“SPORT Status (SPORTx_STAT) Register” on page 13-28</td>
</tr>
<tr>
<td>0xFFC0 0834</td>
<td>SPORT0_CHNL</td>
<td>“SPORT Current Channel (SPORTx_CHNL) Register” on page 13-59</td>
</tr>
</tbody>
</table>
### SPORT Controller Registers

Table B-17. SPORT0 Controller Registers (Cont’d)

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0 0838</td>
<td>SPORT0_MCMC1</td>
<td>“SPORT Multichannel Configuration (SPORTx_MCMCn) Registers” on page 13-53</td>
</tr>
<tr>
<td>0xFFC0 083C</td>
<td>SPORT0_MCMC2</td>
<td>“SPORT Multichannel Configuration (SPORTx_MCMCn) Registers” on page 13-53</td>
</tr>
<tr>
<td>0xFFC0 0840</td>
<td>SPORT0_MTCS0</td>
<td>“SPORT Multichannel Transmit Selection (SPORTx_MTCSn) Registers” on page 13-64</td>
</tr>
<tr>
<td>0xFFC0 0844</td>
<td>SPORT0_MTCS1</td>
<td>“SPORT Multichannel Transmit Selection (SPORTx_MTCSn) Registers” on page 13-64</td>
</tr>
<tr>
<td>0xFFC0 0848</td>
<td>SPORT0_MTCS2</td>
<td>“SPORT Multichannel Transmit Selection (SPORTx_MTCSn) Registers” on page 13-64</td>
</tr>
<tr>
<td>0xFFC0 084C</td>
<td>SPORT0_MTCS3</td>
<td>“SPORT Multichannel Transmit Selection (SPORTx_MTCSn) Registers” on page 13-64</td>
</tr>
<tr>
<td>0xFFC0 0850</td>
<td>SPORT0_MRC50</td>
<td>“SPORT Multichannel Receive Selection (SPORTx_MRCSn) Registers” on page 13-62</td>
</tr>
<tr>
<td>0xFFC0 0854</td>
<td>SPORT0_MRC51</td>
<td>“SPORT Multichannel Receive Selection (SPORTx_MRCSn) Registers” on page 13-62</td>
</tr>
<tr>
<td>0xFFC0 0858</td>
<td>SPORT0_MRC52</td>
<td>“SPORT Multichannel Receive Selection (SPORTx_MRCSn) Registers” on page 13-62</td>
</tr>
<tr>
<td>0xFFC0 085C</td>
<td>SPORT0_MRC53</td>
<td>“SPORT Multichannel Receive Selection (SPORTx_MRCSn) Registers” on page 13-62</td>
</tr>
</tbody>
</table>

Table B-18. SPORT1 Controller Registers

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0 0900</td>
<td>SPORT1_TCR1</td>
<td>“SPORT Transmit Configuration (SPORTx_TCR1, SPORTx_TCR2) Registers” on page 13-11</td>
</tr>
<tr>
<td>0xFFC0 0904</td>
<td>SPORT1_TCR2</td>
<td>“SPORT Transmit Configuration (SPORTx_TCR1, SPORTx_TCR2) Registers” on page 13-11</td>
</tr>
</tbody>
</table>
Table B-18. SPORT1 Controller Registers (Cont’d)

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0 0908</td>
<td>SPORT1_TCLKDIV</td>
<td>“SPORT Transmit Serial Clock Divider (SPORTx_TCLKDIV, SPORTx_RCLKDIV) Registers” on page 13-31</td>
</tr>
<tr>
<td>0xFFC0 090C</td>
<td>SPORT1_TFSDIV</td>
<td>“SPORT Transmit Frame Sync Divider (SPORTx_TFSDIV, SPORTx_RFSDIV) Register” on page 13-32</td>
</tr>
<tr>
<td>0xFFC0 0910</td>
<td>SPORT1_TX</td>
<td>“SPORT Transmit Data (SPORTx_TX) Register” on page 13-23</td>
</tr>
<tr>
<td>0xFFC0 0918</td>
<td>SPORT1_RX</td>
<td>“SPORT Receive Data (SPORTx_RX) Register” on page 13-25</td>
</tr>
<tr>
<td>0xFFC0 0920</td>
<td>SPORT1_RCR1</td>
<td>“SPORT Receive Configuration (SPORTx_RCR1, SPORTx_RCR2) Registers” on page 13-18</td>
</tr>
<tr>
<td>0xFFC0 0924</td>
<td>SPORT1_RCR2</td>
<td>“SPORT Receive Configuration (SPORTx_RCR1, SPORTx_RCR2) Registers” on page 13-18</td>
</tr>
<tr>
<td>0xFFC0 0928</td>
<td>SPORT1_RCLKDIV</td>
<td>“SPORT Transmit Serial Clock Divider (SPORTx_TCLKDIV, SPORTx_RCLKDIV) Registers” on page 13-31</td>
</tr>
<tr>
<td>0xFFC0 092C</td>
<td>SPORT1_RFSDIV</td>
<td>“SPORT Transmit Frame Sync Divider (SPORTx_TFSDIV, SPORTx_RFSDIV) Register” on page 13-32</td>
</tr>
<tr>
<td>0xFFC0 0930</td>
<td>SPORT1_STAT</td>
<td>“SPORT Status (SPORTx_STAT) Register” on page 13-28</td>
</tr>
<tr>
<td>0xFFC0 0934</td>
<td>SPORT1_CHNL</td>
<td>“SPORT Current Channel (SPORTx_CHNL) Register” on page 13-59</td>
</tr>
<tr>
<td>0xFFC0 0938</td>
<td>SPORT1_MCMC1</td>
<td>“SPORT Multichannel Configuration (SPORTx_MCMCn) Registers” on page 13-53</td>
</tr>
<tr>
<td>0xFFC0 093C</td>
<td>SPORT1_MCMC2</td>
<td>“SPORT Multichannel Configuration (SPORTx_MCMCn) Registers” on page 13-53</td>
</tr>
<tr>
<td>0xFFC0 0940</td>
<td>SPORT1_MTCS0</td>
<td>“SPORT Multichannel Transmit Selection (SPORTx_MTCSn) Registers” on page 13-64</td>
</tr>
</tbody>
</table>
Table B-18. SPORT1 Controller Registers (Cont’d)

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0 0944</td>
<td>SPORT1_MTCS1</td>
<td>“SPORT Multichannel Transmit Selection (SPORTx_MTCSn) Registers” on page 13-64</td>
</tr>
<tr>
<td>0xFFC0 0948</td>
<td>SPORT1_MTCS2</td>
<td>“SPORT Multichannel Transmit Selection (SPORTx_MTCSn) Registers” on page 13-64</td>
</tr>
<tr>
<td>0xFFC0 094C</td>
<td>SPORT1_MTCS3</td>
<td>“SPORT Multichannel Transmit Selection (SPORTx_MTCSn) Registers” on page 13-64</td>
</tr>
<tr>
<td>0xFFC0 0950</td>
<td>SPORT1_MRCS0</td>
<td>“SPORT Multichannel Receive Selection (SPORTx_MRCSn) Registers” on page 13-62</td>
</tr>
<tr>
<td>0xFFC0 0954</td>
<td>SPORT1_MRCS1</td>
<td>“SPORT Multichannel Receive Selection (SPORTx_MRCSn) Registers” on page 13-62</td>
</tr>
<tr>
<td>0xFFC0 0958</td>
<td>SPORT1_MRCS2</td>
<td>“SPORT Multichannel Receive Selection (SPORTx_MRCSn) Registers” on page 13-62</td>
</tr>
<tr>
<td>0xFFC0 095C</td>
<td>SPORT1_MRCS3</td>
<td>“SPORT Multichannel Receive Selection (SPORTx_MRCSn) Registers” on page 13-62</td>
</tr>
</tbody>
</table>

Table B-19. SPORT2 Controller Registers

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0 2500</td>
<td>SPORT2_TCR1</td>
<td>“SPORT Transmit Configuration (SPORTx_TCR1, SPORTx_TCR2) Registers” on page 13-11</td>
</tr>
<tr>
<td>0xFFC0 2504</td>
<td>SPORT2_TCR2</td>
<td>“SPORT Transmit Configuration (SPORTx_TCR1, SPORTx_TCR2) Registers” on page 13-11</td>
</tr>
<tr>
<td>0xFFC0 2508</td>
<td>SPORT2_TCLKDIV</td>
<td>“SPORT Transmit Serial Clock Divider (SPORTx_TCLKDIV, SPORTx_RCLKDIV) Registers” on page 13-31</td>
</tr>
<tr>
<td>0xFFC0 250C</td>
<td>SPORT2_TFSDIV</td>
<td>“SPORT Transmit Frame Sync Divider (SPORTx_TFSDIV, SPORTx_RFSDIV) Register” on page 13-32</td>
</tr>
</tbody>
</table>
### System MMR Assignments

Table B-19. SPORT2 Controller Registers (Cont’d)

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0 2510</td>
<td>SPORT2_TX</td>
<td>“SPORT Transmit Data (SPORTx_TX) Register” on page 13-23</td>
</tr>
<tr>
<td>0xFFC0 2518</td>
<td>SPORT2_RX</td>
<td>“SPORT Receive Data (SPORTx_RX) Register” on page 13-25</td>
</tr>
<tr>
<td>0xFFC0 2520</td>
<td>SPORT2_RCR1</td>
<td>“SPORT Receive Configuration (SPORTx_RCR1, SPORTx_RCR2) Registers” on page 13-18</td>
</tr>
<tr>
<td>0xFFC0 2524</td>
<td>SPORT2_RCR2</td>
<td>“SPORT Receive Configuration (SPORTx_RCR1, SPORTx_RCR2) Registers” on page 13-18</td>
</tr>
<tr>
<td>0xFFC0 2528</td>
<td>SPORT2_RCLKDIV</td>
<td>“SPORT Transmit Serial Clock Divider (SPORTx_TCLKDIV, SPORTx_RCLKDIV) Registers” on page 13-31</td>
</tr>
<tr>
<td>0xFFC0 252C</td>
<td>SPORT2_RFSDIV</td>
<td>“SPORT Transmit Frame Sync Divider (SPORTx_TFSDIV, SPORTx_RFSDIV) Register” on page 13-32</td>
</tr>
<tr>
<td>0xFFC0 2530</td>
<td>SPORT2_STAT</td>
<td>“SPORT Status (SPORTx_STAT) Register” on page 13-28</td>
</tr>
<tr>
<td>0xFFC0 2534</td>
<td>SPORT2_CHNL</td>
<td>“SPORT Current Channel (SPORTx_CHNL) Register” on page 13-59</td>
</tr>
<tr>
<td>0xFFC0 2538</td>
<td>SPORT2_MCMC1</td>
<td>“SPORT Multichannel Configuration (SPORTx_MCMCn) Registers” on page 13-53</td>
</tr>
<tr>
<td>0xFFC0 253C</td>
<td>SPORT2_MCMC2</td>
<td>“SPORT Multichannel Configuration (SPORTx_MCMCn) Registers” on page 13-53</td>
</tr>
<tr>
<td>0xFFC0 2540</td>
<td>SPORT2_MTCS0</td>
<td>“SPORT Multichannel Transmit Selection (SPORTx_MTCSn) Registers” on page 13-64</td>
</tr>
<tr>
<td>0xFFC0 2544</td>
<td>SPORT2_MTCS1</td>
<td>“SPORT Multichannel Transmit Selection (SPORTx_MTCSn) Registers” on page 13-64</td>
</tr>
<tr>
<td>0xFFC0 2548</td>
<td>SPORT2_MTCS2</td>
<td>“SPORT Multichannel Transmit Selection (SPORTx_MTCSn) Registers” on page 13-64</td>
</tr>
<tr>
<td>0xFFC0 254C</td>
<td>SPORT2_MTCS3</td>
<td>“SPORT Multichannel Transmit Selection (SPORTx_MTCSn) Registers” on page 13-64</td>
</tr>
</tbody>
</table>
### Table B-19. SPORT2 Controller Registers (Cont’d)

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFFC0 2550</td>
<td>SPORT2_MRCS0</td>
<td>“SPORT Multichannel Receive Selection (SPORTx_MRCSn) Registers” on page 13-62</td>
</tr>
<tr>
<td>0xFFFFC0 2554</td>
<td>SPORT2_MRCS1</td>
<td>“SPORT Multichannel Receive Selection (SPORTx_MRCSn) Registers” on page 13-62</td>
</tr>
<tr>
<td>0xFFFFC0 2558</td>
<td>SPORT2_MRCS2</td>
<td>“SPORT Multichannel Receive Selection (SPORTx_MRCSn) Registers” on page 13-62</td>
</tr>
<tr>
<td>0xFFFFC0 255C</td>
<td>SPORT2_MRCS3</td>
<td>“SPORT Multichannel Receive Selection (SPORTx_MRCSn) Registers” on page 13-62</td>
</tr>
</tbody>
</table>

### Table B-20. SPORT3 Controller Registers

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFFC0 2600</td>
<td>SPORT3_TCR1</td>
<td>“SPORT Transmit Configuration (SPORTx_TCR1, SPORTx_TCR2) Registers” on page 13-11</td>
</tr>
<tr>
<td>0xFFFFC0 2604</td>
<td>SPORT3_TCR2</td>
<td>“SPORT Transmit Configuration (SPORTx_TCR1, SPORTx_TCR2) Registers” on page 13-11</td>
</tr>
<tr>
<td>0xFFFFC0 2608</td>
<td>SPORT3_TCLKDIV</td>
<td>“SPORT Transmit Serial Clock Divider (SPORTx_TCLKDIV, SPORTx_RCLKDIV) Registers” on page 13-31</td>
</tr>
<tr>
<td>0xFFFFC0 260C</td>
<td>SPORT3_TFSDIV</td>
<td>“SPORT Transmit Frame Sync Divider (SPORTx_TFSDIV, SPORTx_RFSDIV) Register” on page 13-32</td>
</tr>
<tr>
<td>0xFFFFC0 2610</td>
<td>SPORT3_TX</td>
<td>“SPORT Transmit Data (SPORTx_TX) Register” on page 13-23</td>
</tr>
<tr>
<td>0xFFFFC0 2618</td>
<td>SPORT3_RX</td>
<td>“SPORT Receive Data (SPORTx_RX) Register” on page 13-25</td>
</tr>
<tr>
<td>0xFFFFC0 2620</td>
<td>SPORT3_RCR1</td>
<td>“SPORT Receive Configuration (SPORTx_RCR1, SPORTx_RCR2) Registers” on page 13-18</td>
</tr>
</tbody>
</table>
Table B-20. SPORT3 Controller Registers (Cont’d)

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFF0 2624</td>
<td>SPORT3_RCR2</td>
<td>“SPORT Receive Configuration (SPORTx_RCR1, SPORTx_RCR2) Registers” on page 13-18</td>
</tr>
<tr>
<td>0xFFF0 2628</td>
<td>SPORT3_RCLKDIV</td>
<td>“SPORT Transmit Serial Clock Divider (SPORTx_TCLKDIV, SPORTx_RCLKDIV) Registers” on page 13-31</td>
</tr>
<tr>
<td>0xFFF0 262C</td>
<td>SPORT3_RFSDIV</td>
<td>“SPORT Transmit Frame Sync Divider (SPORTx_TFSDIV, SPORTx_RFSDIV) Register” on page 13-32</td>
</tr>
<tr>
<td>0xFFF0 2630</td>
<td>SPORT3_STAT</td>
<td>“SPORT Status (SPORTx_STAT) Register” on page 13-28</td>
</tr>
<tr>
<td>0xFFF0 2634</td>
<td>SPORT3_CHNL</td>
<td>“SPORT Current Channel (SPORTx_CHNL) Register” on page 13-59</td>
</tr>
<tr>
<td>0xFFF0 2638</td>
<td>SPORT3_MCMC1</td>
<td>“SPORT Multichannel Configuration (SPORTx_MCMCn) Registers” on page 13-53</td>
</tr>
<tr>
<td>0xFFF0 263C</td>
<td>SPORT3_MCMC2</td>
<td>“SPORT Multichannel Configuration (SPORTx_MCMCn) Registers” on page 13-53</td>
</tr>
<tr>
<td>0xFFF0 2640</td>
<td>SPORT3_MTCS0</td>
<td>“SPORT Multichannel Transmit Selection (SPORTx_MTCSn) Registers” on page 13-64</td>
</tr>
<tr>
<td>0xFFF0 2644</td>
<td>SPORT3_MTCS1</td>
<td>“SPORT Multichannel Transmit Selection (SPORTx_MTCSn) Registers” on page 13-64</td>
</tr>
<tr>
<td>0xFFF0 2648</td>
<td>SPORT3_MTCS2</td>
<td>“SPORT Multichannel Transmit Selection (SPORTx_MTCSn) Registers” on page 13-64</td>
</tr>
<tr>
<td>0xFFF0 264C</td>
<td>SPORT3_MTCS3</td>
<td>“SPORT Multichannel Transmit Selection (SPORTx_MTCSn) Registers” on page 13-64</td>
</tr>
<tr>
<td>0xFFF0 2650</td>
<td>SPORT3_MRCS0</td>
<td>“SPORT Multichannel Receive Selection (SPORTx_MRCSn) Registers” on page 13-62</td>
</tr>
<tr>
<td>0xFFF0 2654</td>
<td>SPORT3_MRCS1</td>
<td>“SPORT Multichannel Receive Selection (SPORTx_MRCSn) Registers” on page 13-62</td>
</tr>
</tbody>
</table>
DMA controller 0 registers (0xFFC0 0C00 – 0xFFC0 0FFF)
DMA controller 1 registers (0xFFC0 1C00 – 0xFFC0 1FFF)

Table B-21. DMA Traffic Control Registers

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
</table>
| 0xFFC0 0B0C           | DMAC0_TC_PER   | “DMA Traffic Control Counter Period (DMACx_TC_PER) and Counter
DMAC1_TC_PER          |              | (DMACx_TC_CNT) Registers” on page 9-61                                     |
| 0xFFC0 0B10           | DMAC0_TC_CNT   | “DMA Traffic Control Counter Period (DMACx_TC_PER) and Counter
DMAC1_TC_CNT          |              | (DMACx_TC_CNT) Registers” on page 9-61                                     |

Since each DMA channel has an identical MMR set, with fixed offsets from the base address associated with that DMA channel, it is convenient to view the MMR information as provided in Table B-22 and Table B-23. Table B-22 identifies the base address of each DMA channel, as well as the register prefix that identifies the channel. Table B-24 then lists the register suffix and provides its offset from the Base Address.

As an example, the DMA channel 0 Y modify register is called DMA0_Y_MODIFY, and its address is 0xFFC0 0C1C. Likewise, the memory
DMA stream 0 source current address register is called MDMA_S0_CURR_ADDR, and its address is 0xFFC0 0E64.

Table B-22. DMA Channel Base Addresses for DMA Controller 0

<table>
<thead>
<tr>
<th>DMA Channel Identifier</th>
<th>MMR Base Address</th>
<th>Register Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA0</td>
<td>0xFFC0 0C00</td>
<td>DMA0_</td>
</tr>
<tr>
<td>DMA1</td>
<td>0xFFC0 0C40</td>
<td>DMA1_</td>
</tr>
<tr>
<td>DMA2</td>
<td>0xFFC0 0C80</td>
<td>DMA2_</td>
</tr>
<tr>
<td>DMA3</td>
<td>0xFFC0 0CC0</td>
<td>DMA3_</td>
</tr>
<tr>
<td>DMA4</td>
<td>0xFFC0 0D00</td>
<td>DMA4_</td>
</tr>
<tr>
<td>DMA5</td>
<td>0xFFC0 0D40</td>
<td>DMA5_</td>
</tr>
<tr>
<td>DMA6</td>
<td>0xFFC0 0D80</td>
<td>DMA6_</td>
</tr>
<tr>
<td>DMA7</td>
<td>0xFFC0 0DC0</td>
<td>DMA7_</td>
</tr>
<tr>
<td>Mem DMA 0 Stream 0 Destination</td>
<td>0xFFC0 0E00</td>
<td>MDMA0_D0_</td>
</tr>
<tr>
<td>Mem DMA 0 Stream 0 Source</td>
<td>0xFFC0 0E40</td>
<td>MDMA0_S0_</td>
</tr>
<tr>
<td>Mem DMA 0 Stream 1 Destination</td>
<td>0xFFC0 0E80</td>
<td>MDMA0_D1_</td>
</tr>
<tr>
<td>Mem DMA 0 Stream 1 Source</td>
<td>0xFFC0 0EC0</td>
<td>MDMA0_S1_</td>
</tr>
</tbody>
</table>

Table B-23. DMA Channel Base Addresses for DMA Controller 1

<table>
<thead>
<tr>
<th>DMA Channel Identifier</th>
<th>MMR Base Address</th>
<th>Register Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA8</td>
<td>0xFFC0 1C00</td>
<td>DMA8_</td>
</tr>
<tr>
<td>DMA9</td>
<td>0xFFC0 1C40</td>
<td>DMA9_</td>
</tr>
<tr>
<td>DMA10</td>
<td>0xFFC0 1C80</td>
<td>DMA10_</td>
</tr>
<tr>
<td>DMA11</td>
<td>0xFFC0 1CC0</td>
<td>DMA11_</td>
</tr>
<tr>
<td>DMA12</td>
<td>0xFFC0 1D00</td>
<td>DMA12_</td>
</tr>
<tr>
<td>DMA13</td>
<td>0xFFC0 1D40</td>
<td>DMA13_</td>
</tr>
<tr>
<td>DMA14</td>
<td>0xFFC0 1D80</td>
<td>DMA14_</td>
</tr>
</tbody>
</table>
### DMA/Memory DMA Control Registers

Table B-23. DMA Channel Base Addresses for DMA Controller 1 (Cont’d)

<table>
<thead>
<tr>
<th>DMA Channel Identifier</th>
<th>MMR Base Address</th>
<th>Register Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA15</td>
<td>0xFFC0 1DC0</td>
<td>DMA15_</td>
</tr>
<tr>
<td>DMA16</td>
<td>0xFFC0 1E00</td>
<td>DMA16_</td>
</tr>
<tr>
<td>DMA17</td>
<td>0xFFC0 1E40</td>
<td>DMA17_</td>
</tr>
<tr>
<td>DMA18</td>
<td>0xFFC0 1E80</td>
<td>DMA18_</td>
</tr>
<tr>
<td>DMA19</td>
<td>0xFFC0 1EC0</td>
<td>DMA19_</td>
</tr>
<tr>
<td>Mem DMA 1 Stream 0 Destination</td>
<td>0xFFC0 1F00</td>
<td>MDMA1_D0_</td>
</tr>
<tr>
<td>Mem DMA 1 Stream 0 Source</td>
<td>0xFFC0 1F40</td>
<td>MDMA1_S0_</td>
</tr>
<tr>
<td>Mem DMA 1 Stream 1 Destination</td>
<td>0xFFC0 1F80</td>
<td>MDMA1_D1_</td>
</tr>
<tr>
<td>Mem DMA 1 Stream 1 Source</td>
<td>0xFFC0 1FC0</td>
<td>MDMA1_S1_</td>
</tr>
</tbody>
</table>

Table B-24. DMA Register Suffix and Offset

<table>
<thead>
<tr>
<th>Register Suffix</th>
<th>Offset From Base</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEXT_DESC_PTR</td>
<td>0x00</td>
<td>“Next Descriptor Pointer (DMAx_NEXT_DESC_PTR, MDMAx_yy_NEXT_DESC_PTR) Registers” on page 9-8</td>
</tr>
<tr>
<td>START_ADDR</td>
<td>0x04</td>
<td>“Start Address (DMAx_START_ADDR, MDMAx_yy_START_ADDR) Registers” on page 9-10</td>
</tr>
<tr>
<td>CONFIG</td>
<td>0x08</td>
<td>“DMA Configuration (DMAx_CONFIG, MDMAx_yy_CONFIG) Registers” on page -11</td>
</tr>
<tr>
<td>X_COUNT</td>
<td>0x10</td>
<td>“Inner Loop Count (DMAx_X_COUNT, MDMAx_yy_X_COUNT) Registers” on page 9-14</td>
</tr>
<tr>
<td>X_MODIFY</td>
<td>0x14</td>
<td>“Inner Loop Address Increment (DMAx_X_MODIFY, MDMAx_yy_X_MODIFY) Registers” on page 9-14</td>
</tr>
<tr>
<td>Y_COUNT</td>
<td>0x18</td>
<td>“Outer Loop Count (DMAx_Y_COUNT, MDMAx_yy_Y_COUNT) Registers” on page 9-15</td>
</tr>
<tr>
<td>Y_MODIFY</td>
<td>0x1C</td>
<td>“Outer Loop Address Increment (DMAx_Y_MODIFY, MDMAx_yy_Y_MODIFY) Registers” on page 9-17</td>
</tr>
</tbody>
</table>
Table B-24. DMA Register Suffix and Offset (Cont’d)

<table>
<thead>
<tr>
<th>Register Suffix</th>
<th>Offset From Base</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURR_DESC_PTR</td>
<td>0x20</td>
<td>“Current Descriptor Pointer (DMAx_CURR_DESC_PTR, MDMAx_yy_CURR_DESC_PTR) Registers” on page 9-17</td>
</tr>
<tr>
<td>CURR_ADDR</td>
<td>0x24</td>
<td>“Current Address (DMAx_CURR_ADDR, MDMAx_yy_CURR_ADDR) Registers” on page 9-18</td>
</tr>
<tr>
<td>IRQ_STATUS</td>
<td>0x28</td>
<td>“Interrupt Status (DMAx_IRQ_STATUS, MDMAx_yy_IRQ_STATUS) Registers” on page 9-26</td>
</tr>
<tr>
<td>PERIPHERAL_MAP</td>
<td>0x2C</td>
<td>“Peripheral Map (DMAx_PERIPHERAL_MAP, MDMAx_yy_PERIPHERAL_MAP) Registers” on page 9-21</td>
</tr>
<tr>
<td>CURR_X_COUNT</td>
<td>0x30</td>
<td>“Current Inner Loop Count (DMAx_CURR_X_COUNT, MDMAx_yy_CURR_X_COUNT) Registers” on page 9-19</td>
</tr>
<tr>
<td>CURR_Y_COUNT</td>
<td>0x38</td>
<td>“Current Outer Loop Count (DMAx_CURR_Y_COUNT, MDMAx_yy_CURR_Y_COUNT) Registers” on page 9-20</td>
</tr>
</tbody>
</table>

External Bus Interface Unit Registers

External bus interface unit registers (0xFFC0 0A00 – 0xFFC0 0AFF)

Table B-25. External Bus Interface Unit Registers

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0 0A00</td>
<td>EBIU_AMGCTL</td>
<td>“EBIU_AMGCTL Register” on page 18-10</td>
</tr>
<tr>
<td>0xFFC0 0A04</td>
<td>EBIU_AMBCTL0</td>
<td>“EBIU_AMBCTL0 and EBIU_AMBCTL1 Registers” on page 18-11</td>
</tr>
</tbody>
</table>
CAN Registers

Table B-25. External Bus Interface Unit Registers (Cont’d)

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0 0A08</td>
<td>EBIU_AMBCTL1</td>
<td>&quot;EBIU_AMBCTL0 and EBIU_AMBCTL1 Registers&quot; on page 18-11</td>
</tr>
<tr>
<td>0xFFC0 0A10</td>
<td>EBIU_SDGCTL</td>
<td>&quot;EBIU_SDGCTL Register&quot; on page 18-33</td>
</tr>
<tr>
<td>0xFFC0 0A14</td>
<td>EBIU_SDBCTL</td>
<td>&quot;EBIU_SDBCTL Register&quot; on page 18-44</td>
</tr>
<tr>
<td>0xFFC0 0A18</td>
<td>EBIU_SDRRC</td>
<td>&quot;EBIU_SDRRC Register&quot; on page 18-47</td>
</tr>
<tr>
<td>0xFFC0 0A1C</td>
<td>EBIU_SDSTAT</td>
<td>&quot;EBIU_SDSTAT Register&quot; on page 18-47</td>
</tr>
</tbody>
</table>

Table B-26. CAN Control and Configuration Registers

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0 2A00</td>
<td>CAN_MC1</td>
<td>&quot;CAN Mailbox Configuration (CAN_MCx) and Direction (CAN_MDx) Registers&quot; on page 19-35</td>
</tr>
<tr>
<td>0xFFC0 2A04</td>
<td>CAN_MD1</td>
<td>&quot;CAN Mailbox Configuration (CAN_MCx) and Direction (CAN_MDx) Registers&quot; on page 19-35</td>
</tr>
<tr>
<td>0xFFC0 2A08</td>
<td>CAN_TRS1</td>
<td>&quot;CAN Transmission Request Set (CAN_TRSx) Registers&quot; on page 19-51</td>
</tr>
<tr>
<td>0xFFC0 2A0C</td>
<td>CAN_TRR1</td>
<td>&quot;CAN Transmission Request Reset (CAN_TRRx) Registers&quot; on page 19-53</td>
</tr>
<tr>
<td>0xFFC0 2A10</td>
<td>CAN_TA1</td>
<td>&quot;CAN Transmission Acknowledge (CAN_TAx) Register&quot; on page 19-57</td>
</tr>
<tr>
<td>0xFFC0 2A14</td>
<td>CAN_AA1</td>
<td>&quot;CAN Abort Acknowledge (CAN_AAx) Register&quot; on page 19-56</td>
</tr>
<tr>
<td>0xFFC0 2A18</td>
<td>CAN_RMP1</td>
<td>&quot;CAN Receive Message Pending (CAN_RMPx) Register&quot; on page 19-45</td>
</tr>
</tbody>
</table>
Table B-26. CAN Control and Configuration Registers (Cont’d)

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0 2A1C</td>
<td>CAN_RML1</td>
<td>“CAN Receive Message Lost (CAN_RMLx) Register” on page 19-46</td>
</tr>
<tr>
<td>0xFFC0 2A20</td>
<td>CAN_MBTIF1</td>
<td>“CAN Mailbox Interrupt Mask Flag (CAN_MBTIFx) Registers” on page 19-67</td>
</tr>
<tr>
<td>0xFFC0 2A24</td>
<td>CAN_MBRIF1</td>
<td>“CAN Mailbox Receive Interrupt Flag (CAN_MBRIFx) Registers” on page 19-68</td>
</tr>
<tr>
<td>0xFFC0 2A28</td>
<td>CAN_MBIM1</td>
<td>“CAN Mailbox Interrupt Mask (CAN_MBIMx) Registers” on page 19-65</td>
</tr>
<tr>
<td>0xFFC0 2A2C</td>
<td>CAN_RFH1</td>
<td>“CAN Remote Frame Handling (CAN_RFHx) Registers” on page 19-60</td>
</tr>
<tr>
<td>0xFFC0 2A30</td>
<td>CAN_OPSS1</td>
<td>“CAN Overwrite Protection/Single Shot Transmission (CAN_OPSSx) Register” on page 19-48</td>
</tr>
<tr>
<td>0xFFC0 2A40</td>
<td>CAN_MC2</td>
<td>“CAN Mailbox Configuration (CAN_MCx) and Direction (CAN_MDx) Registers” on page 19-35</td>
</tr>
<tr>
<td>0xFFC0 2A44</td>
<td>CAN_MD2</td>
<td>“CAN Mailbox Configuration (CAN_MCx) and Direction (CAN_MDx) Registers” on page 19-35</td>
</tr>
<tr>
<td>0xFFC0 2A48</td>
<td>CAN_TRS2</td>
<td>“CAN Transmission Request Set (CAN_TRSx) Registers” on page 19-51</td>
</tr>
<tr>
<td>0xFFC0 2A4C</td>
<td>CAN_TRR2</td>
<td>“CAN Transmission Request Reset (CAN_TRRx) Registers” on page 19-53</td>
</tr>
<tr>
<td>0xFFC0 2A50</td>
<td>CAN_TA2</td>
<td>“CAN Transmission Acknowledge (CAN_TAx) Register” on page 19-57</td>
</tr>
<tr>
<td>0xFFC0 2A54</td>
<td>CAN_AA2</td>
<td>“CAN Abort Acknowledge (CAN_AAx) Register” on page 19-56</td>
</tr>
<tr>
<td>0xFFC0 2A58</td>
<td>CAN_RMP2</td>
<td>“CAN Receive Message Pending (CAN_RMPx) Register” on page 19-45</td>
</tr>
<tr>
<td>0xFFC0 2A5C</td>
<td>CAN_RML2</td>
<td>“CAN Receive Message Lost (CAN_RMLx) Register” on page 19-46</td>
</tr>
<tr>
<td>0xFFC0 2A60</td>
<td>CAN_MBTIF2</td>
<td>“CAN Mailbox Interrupt Mask Flag (CAN_MBTIFx) Registers” on page 19-67</td>
</tr>
</tbody>
</table>
### Table B-26. CAN Control and Configuration Registers (Cont’d)

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0 2A64</td>
<td>CAN_MBRIF2</td>
<td>“CAN Mailbox Receive Interrupt Flag (CAN_MBRIFx) Registers” on page 19-68</td>
</tr>
<tr>
<td>0xFFC0 2A68</td>
<td>CAN_MBIM2</td>
<td>“CAN Mailbox Interrupt Mask (CAN_MBIMx) Registers” on page 19-65</td>
</tr>
<tr>
<td>0xFFC0 2A6C</td>
<td>CAN_RFH2</td>
<td>“CAN Remote Frame Handling (CAN_RFHx) Registers” on page 19-60</td>
</tr>
<tr>
<td>0xFFC0 2A70</td>
<td>CAN_OPSS2</td>
<td>“CAN Overwrite Protection/Single Shot Transmission (CAN_OPSSx) Register” on page 19-48</td>
</tr>
<tr>
<td>0xFFC0 2A80</td>
<td>CAN_CLOCK</td>
<td>“CAN Clock (CAN_CLOCK) Register” on page 19-14</td>
</tr>
<tr>
<td>0xFFC0 2A84</td>
<td>CAN_TIMING</td>
<td>“CAN Timing (CAN_TIMING) Register” on page 19-15</td>
</tr>
<tr>
<td>0xFFC0 2A88</td>
<td>CAN_DEBUG</td>
<td>“CAN Debug (CAN_DEBUG) Register” on page 19-17</td>
</tr>
<tr>
<td>0xFFC0 2A8C</td>
<td>CAN_Status</td>
<td>“CAN Status (CAN_STATUS) Register” on page 19-11</td>
</tr>
<tr>
<td>0xFFC0 2A90</td>
<td>CAN_CEC</td>
<td>“CAN Error Status (CAN_ESR) Register” on page 19-86</td>
</tr>
<tr>
<td>0xFFC0 2A94</td>
<td>CAN_GIS</td>
<td>“CAN Global Interrupt Status (CAN_GIS) Register” on page 19-75</td>
</tr>
<tr>
<td>0xFFC0 2A98</td>
<td>CAN_GIM</td>
<td>“CAN Global Interrupt Mask (CAN_GIM) Register” on page 19-74</td>
</tr>
<tr>
<td>0xFFC0 2A9C</td>
<td>CAN_GIF</td>
<td>“CAN Global Interrupt Flag (CAN_GIF) Register” on page 19-75</td>
</tr>
<tr>
<td>0xFFC0 2AA0</td>
<td>CAN_CONTROL</td>
<td>“CAN Error Status (CAN_ESR) Register” on page 19-86</td>
</tr>
<tr>
<td>0xFFC0 2AA4</td>
<td>CAN_INTR</td>
<td>“CAN Interrupt (CAN_INTR) Register” on page 19-62</td>
</tr>
<tr>
<td>0xFFC0 2AAC</td>
<td>CAN_MBTD</td>
<td>“CAN Mailbox Temporary Disable (CAN_MBTD) Register” on page 19-59</td>
</tr>
</tbody>
</table>
Table B-26. CAN Control and Configuration Registers (Cont’d)

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0 2AB0</td>
<td>CAN_EWR</td>
<td>“CAN Error Counter Warning Level (CAN_EWR) Register” on page 19-88</td>
</tr>
<tr>
<td>0xFFC0 2AB4</td>
<td>CAN_ESR</td>
<td>“CAN Error Status (CAN_ESR) Register” on page 19-86</td>
</tr>
<tr>
<td>0xFFC0 2AC4</td>
<td>CAN_UCCNT</td>
<td>“CAN Universal Counter (CAN_UCCNT) Register” on page 19-83</td>
</tr>
<tr>
<td>0xFFC0 2AC8</td>
<td>CAN_UCRC</td>
<td>“CAN Universal Counter Reload/Capture (CAN_UCRC) Register” on page 19-84</td>
</tr>
<tr>
<td>0xFFC0 2ACC</td>
<td>CAN_UCCNF</td>
<td>“CAN Universal Counter Configuration (CAN_UCCNF) Register” on page 19-80</td>
</tr>
</tbody>
</table>

Table B-27. CAN Mailbox Acceptance Mask Registers

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0 2B00</td>
<td>CAN_AM00L</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFC0 2B04</td>
<td>CAN_AM00H</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFC0 2B08</td>
<td>CAN_AM01L</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFC0 2B0C</td>
<td>CAN_AM01H</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFC0 2B10</td>
<td>CAN_AM02L</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFC0 2B14</td>
<td>CAN_AM02H</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFC0 2B18</td>
<td>CAN_AM03L</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFC0 2B1C</td>
<td>CAN_AM03H</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
</tbody>
</table>
Table B-27. CAN Mailbox Acceptance Mask Registers (Cont’d)

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0 2B20</td>
<td>CAN_AM04L</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFC0 2B24</td>
<td>CAN_AM04H</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFC0 2B28</td>
<td>CAN_AM05L</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFC0 2B2C</td>
<td>CAN_AM05H</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFC0 2B30</td>
<td>CAN_AM06L</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFC0 2B34</td>
<td>CAN_AM06H</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFC0 2B38</td>
<td>CAN_AM07L</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFC0 2B3C</td>
<td>CAN_AM07H</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFC0 2B40</td>
<td>CAN_AM08L</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFC0 2B44</td>
<td>CAN_AM08H</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFC0 2B48</td>
<td>CAN_AM09L</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFC0 2B4C</td>
<td>CAN_AM09H</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFC0 2B50</td>
<td>CAN_AM10L</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFC0 2B54</td>
<td>CAN_AM10H</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFC0 2B58</td>
<td>CAN_AM11L</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
</tbody>
</table>
### Table B-27. CAN Mailbox Acceptance Mask Registers (Cont'd)

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFFC0 2B5C</td>
<td>CAN_AM11H</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFFFC0 2B60</td>
<td>CAN_AM12L</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFFFC0 2B64</td>
<td>CAN_AM12H</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFFFC0 2B68</td>
<td>CAN_AM13L</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFFFC0 2B6C</td>
<td>CAN_AM13H</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFFFC0 2B70</td>
<td>CAN_AM14L</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFFFC0 2B74</td>
<td>CAN_AM14H</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFFFC0 2B78</td>
<td>CAN_AM15L</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFFFC0 2B7C</td>
<td>CAN_AM15H</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFFFC0 2B80</td>
<td>CAN_AM16L</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFFFC0 2B84</td>
<td>CAN_AM16H</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFFFC0 2B88</td>
<td>CAN_AM17L</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFFFC0 2B8C</td>
<td>CAN_AM17H</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFFFC0 2B90</td>
<td>CAN_AM18L</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFFFC0 2B94</td>
<td>CAN_AM18H</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
</tbody>
</table>
## Table B-27. CAN Mailbox Acceptance Mask Registers (Cont’d)

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFFC0 2B98</td>
<td>CAN_AM19L</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFFFC0 2B9C</td>
<td>CAN_AM19H</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFFFC0 2BA0</td>
<td>CAN_AM20L</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFFFC0 2BA4</td>
<td>CAN_AM20H</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFFFC0 2BA8</td>
<td>CAN_AM21L</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFFFC0 2BAC</td>
<td>CAN_AM21H</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFFFC0 2BB0</td>
<td>CAN_AM22L</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFFFC0 2BB4</td>
<td>CAN_AM22H</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFFFC0 2BB8</td>
<td>CAN_AM23L</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFFFC0 2BBC</td>
<td>CAN_AM23H</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFFFC0 2BC0</td>
<td>CAN_AM24L</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFFFC0 2BC4</td>
<td>CAN_AM24H</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFFFC0 2BC8</td>
<td>CAN_AM25L</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFFFC0 2BCC</td>
<td>CAN_AM25H</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
<tr>
<td>0xFFFFC0 2BD0</td>
<td>CAN_AM26L</td>
<td>“CAN Acceptance Mask (CAN_AMxx) Registers” on page 19-40</td>
</tr>
</tbody>
</table>
Since each CAN mailbox has an identical MMR set, with fixed offsets from the base address associated with that mailbox, it is convenient to view the MMR information as provided in Table B-28 and Table B-29 on page B-35. Table B-28 identifies the base address of each CAN mailbox, as well as the register prefix that identifies mailbox. Table B-29 then lists the register suffix and provides its offset from the base address.
As an example, the CAN mailbox 2 length register is called `CAN_MB02_LENGTH`, and its address is 0xFFC0 2C50. Likewise, the CAN mailbox 17 time stamp register is called `CAN_MB17_TIMESTAMP`, and its address is 0xFFC0 2E34.

Table B-28. CAN Mailbox Base Addresses

<table>
<thead>
<tr>
<th>Mailbox Identifier</th>
<th>MMR Base Address</th>
<th>Register Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xFFC0 2C00</td>
<td>CAN_MB00_</td>
</tr>
<tr>
<td>1</td>
<td>0xFFC0 2C20</td>
<td>CAN_MB01_</td>
</tr>
<tr>
<td>2</td>
<td>0xFFC0 2C40</td>
<td>CAN_MB02_</td>
</tr>
<tr>
<td>3</td>
<td>0xFFC0 2C60</td>
<td>CAN_MB03_</td>
</tr>
<tr>
<td>4</td>
<td>0xFFC0 2C80</td>
<td>CAN_MB04_</td>
</tr>
<tr>
<td>5</td>
<td>0xFFC0 2CA0</td>
<td>CAN_MB05_</td>
</tr>
<tr>
<td>6</td>
<td>0xFFC0 2CC0</td>
<td>CAN_MB06_</td>
</tr>
<tr>
<td>7</td>
<td>0xFFC0 2CE0</td>
<td>CAN_MB07_</td>
</tr>
<tr>
<td>8</td>
<td>0xFFC0 2D00</td>
<td>CAN_MB08_</td>
</tr>
<tr>
<td>9</td>
<td>0xFFC0 2D20</td>
<td>CAN_MB09_</td>
</tr>
<tr>
<td>10</td>
<td>0xFFC0 2D40</td>
<td>CAN_MB10_</td>
</tr>
<tr>
<td>11</td>
<td>0xFFC0 2D60</td>
<td>CAN_MB11_</td>
</tr>
<tr>
<td>12</td>
<td>0xFFC0 2D80</td>
<td>CAN_MB12_</td>
</tr>
<tr>
<td>13</td>
<td>0xFFC0 2DA0</td>
<td>CAN_MB13_</td>
</tr>
<tr>
<td>14</td>
<td>0xFFC0 2DC0</td>
<td>CAN_MB14_</td>
</tr>
<tr>
<td>15</td>
<td>0xFFC0 2DE0</td>
<td>CAN_MB15_</td>
</tr>
<tr>
<td>16</td>
<td>0xFFC0 2E00</td>
<td>CAN_MB16_</td>
</tr>
<tr>
<td>17</td>
<td>0xFFC0 2E20</td>
<td>CAN_MB17_</td>
</tr>
<tr>
<td>18</td>
<td>0xFFC0 2E40</td>
<td>CAN_MB18_</td>
</tr>
<tr>
<td>19</td>
<td>0xFFC0 2E60</td>
<td>CAN_MB19_</td>
</tr>
<tr>
<td>20</td>
<td>0xFFC0 2E80</td>
<td>CAN_MB20_</td>
</tr>
<tr>
<td>21</td>
<td>0xFFC0 2EA0</td>
<td>CAN_MB21_</td>
</tr>
</tbody>
</table>
### System MMR Assignments

Table B-28. CAN Mailbox Base Addresses (Cont’d)

<table>
<thead>
<tr>
<th>Mailbox Identifier</th>
<th>MMR Base Address</th>
<th>Register Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>0xFFC0 2EC0</td>
<td>CAN_MB22_</td>
</tr>
<tr>
<td>23</td>
<td>0xFFC0 2EE0</td>
<td>CAN_MB23_</td>
</tr>
<tr>
<td>24</td>
<td>0xFFC0 2F00</td>
<td>CAN_MB24_</td>
</tr>
<tr>
<td>25</td>
<td>0xFFC0 2F20</td>
<td>CAN_MB25_</td>
</tr>
<tr>
<td>26</td>
<td>0xFFC0 2F40</td>
<td>CAN_MB26_</td>
</tr>
<tr>
<td>27</td>
<td>0xFFC0 2F60</td>
<td>CAN_MB27_</td>
</tr>
<tr>
<td>28</td>
<td>0xFFC0 2F80</td>
<td>CAN_MB28_</td>
</tr>
<tr>
<td>29</td>
<td>0xFFC0 2FA0</td>
<td>CAN_MB29_</td>
</tr>
<tr>
<td>30</td>
<td>0xFFC0 2FC0</td>
<td>CAN_MB30_</td>
</tr>
<tr>
<td>31</td>
<td>0xFFC0 2FE0</td>
<td>CAN_MB31_</td>
</tr>
</tbody>
</table>

Table B-29. CAN Mailbox Register Suffix and Offset

<table>
<thead>
<tr>
<th>Register Suffix</th>
<th>Offset From Base</th>
<th>See Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA0</td>
<td>0x00</td>
<td>Figure 19-15 on page 19-32 and Table 19-10 on page 19-32</td>
</tr>
<tr>
<td>DATA1</td>
<td>0x04</td>
<td>Figure 19-14 on page 19-31 and Table 19-9 on page 19-31</td>
</tr>
<tr>
<td>DATA2</td>
<td>0x08</td>
<td>Figure 19-13 on page 19-30 and Table 19-8 on page 19-30</td>
</tr>
<tr>
<td>DATA3</td>
<td>0x0C</td>
<td>Figure 19-12 on page 19-29 and Table 19-7 on page 19-29</td>
</tr>
<tr>
<td>LENGTH</td>
<td>0x10</td>
<td>Figure 19-11 on page 19-27 and Table 19-6 on page 19-28</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>0x14</td>
<td>Figure 19-10 on page 19-26 and Table 19-5 on page 19-26</td>
</tr>
</tbody>
</table>
Two-Wire Interface Registers

Table B-30. TWI 0 Registers

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0 1400</td>
<td>TWI0_CLKDIV</td>
<td>“TWI Clock Divider (TWIx_CLKDIV) Registers” on page 20-6</td>
</tr>
<tr>
<td>0xFFC0 1404</td>
<td>TWI0_CONTROL</td>
<td>“TWI Control (TWIx_CONTROL) Registers” on page 20-4</td>
</tr>
<tr>
<td>0xFFC0 1408</td>
<td>TWI0_SLAVE_CTRL</td>
<td>“TWI Slave Mode Control (TWIx_SLAVE_CTRL) Registers” on page 20-7</td>
</tr>
<tr>
<td>0xFFC0 140C</td>
<td>TWI0_SLAVE_STAT</td>
<td>“TWI Slave Mode Status (TWIx_SLAVE_STAT) Registers” on page 20-9</td>
</tr>
<tr>
<td>0xFFC0 1410</td>
<td>TWI0_SLAVE_ADDR</td>
<td>“TWI Slave Mode Address (TWIx_SLAVE_ADDR) Registers” on page 20-9</td>
</tr>
<tr>
<td>0xFFC0 1414</td>
<td>TWI0_MASTER_CTRL</td>
<td>“TWI Master Mode Control (TWIx_MASTER_CTRL) Registers” on page 20-11</td>
</tr>
</tbody>
</table>
Table B-30. TWI 0 Registers (Cont’d)

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0 1418</td>
<td>TWI0_MASTER_STAT</td>
<td>“TWI Master Mode Status (TWIx_MASTER_STAT) Registers” on page 20-14</td>
</tr>
<tr>
<td>0xFFC0 141C</td>
<td>TWI0_MASTER_ADDR</td>
<td>“TWI Master Mode Address (TWIx_MASTER_ADDR) Registers” on page 20-14</td>
</tr>
<tr>
<td>0xFFC0 1420</td>
<td>TWI0_INT_STAT</td>
<td>“TWI Interrupt Status (TWIx_INT_STAT) Registers” on page 20-24</td>
</tr>
<tr>
<td>0xFFC0 1428</td>
<td>TWI0_FIFO_CTRL</td>
<td>“TWI FIFO Control (TWIx_FIFO_CTRL) Registers” on page 20-18</td>
</tr>
<tr>
<td>0xFFC0 142C</td>
<td>TWI0_FIFO_STAT</td>
<td>“TWI FIFO Status (TWIx_FIFO_STAT) Registers” on page 20-20</td>
</tr>
<tr>
<td>0xFFC0 1480</td>
<td>TWI0_XMT_DATA8</td>
<td>“TWI FIFO Transmit Data Single Byte (TWIx_XMT_DATA8) Registers” on page 20-27</td>
</tr>
<tr>
<td>0xFFC0 1484</td>
<td>TWI0_XMT_DATA16</td>
<td>“TWI FIFO Transmit Data Double Byte (TWIx_XMT_DATA16) Registers” on page 20-27</td>
</tr>
<tr>
<td>0xFFC0 1488</td>
<td>TWI0_RCV_DATA8</td>
<td>“TWI FIFO Receive Data Single Byte (TWIx_RCV_DATA8) Registers” on page 20-28</td>
</tr>
<tr>
<td>0xFFC0 148C</td>
<td>TWI0_RCV_DATA16</td>
<td>“TWI FIFO Receive Data Double Byte (TWIx_RCV_DATA16) Registers” on page 20-29</td>
</tr>
</tbody>
</table>
### Table B-31. TWI 1 Registers

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFF0 2200</td>
<td>TWI1_CLKDIV</td>
<td>“TWI Clock Divider (TWIx_CLKDIV) Registers” on page 20-6</td>
</tr>
<tr>
<td>0xFFF0 2204</td>
<td>TWI1_CONTROL</td>
<td>“TWI Control (TWIx_CONTROL) Registers” on page 20-4</td>
</tr>
<tr>
<td>0xFFF0 2208</td>
<td>TWI1_SLAVE_CTRL</td>
<td>“TWI Slave Mode Control (TWIx_SLAVE_CTRL) Registers” on page 20-7</td>
</tr>
<tr>
<td>0xFFF0 220C</td>
<td>TWI1_SLAVE_STAT</td>
<td>“TWI Slave Mode Status (TWIx_SLAVE_STAT) Registers” on page 20-9</td>
</tr>
<tr>
<td>0xFFF0 2210</td>
<td>TWI1_SLAVE_ADDR</td>
<td>“TWI Slave Mode Address (TWIx_SLAVE_ADDR) Registers” on page 20-9</td>
</tr>
<tr>
<td>0xFFF0 2214</td>
<td>TWI1_MASTER_CTRL</td>
<td>“TWI Master Mode Control (TWIx_MASTER_CTRL) Registers” on page 20-11</td>
</tr>
<tr>
<td>0xFFF0 2218</td>
<td>TWI1_MASTER_STAT</td>
<td>“TWI Master Mode Status (TWIx_MASTER_STAT) Registers” on page 20-14</td>
</tr>
<tr>
<td>0xFFF0 221C</td>
<td>TWI1_MASTER_ADDR</td>
<td>“TWI Master Mode Address (TWIx_MASTER_ADDR) Registers” on page 20-14</td>
</tr>
<tr>
<td>0xFFF0 2220</td>
<td>TWI1_INT_STAT</td>
<td>“TWI Interrupt Status (TWIx_INT_STAT) Registers” on page 20-24</td>
</tr>
<tr>
<td>0xFFF0 2228</td>
<td>TWI1_FIFO_CTRL</td>
<td>“TWI FIFO Control (TWIx_FIFO_CTRL) Registers” on page 20-18</td>
</tr>
<tr>
<td>0xFFF0 222C</td>
<td>TWI1_FIFO_STAT</td>
<td>“TWI FIFO Status (TWIx_FIFO_STAT) Registers” on page 20-20</td>
</tr>
<tr>
<td>0xFFF0 2280</td>
<td>TWI1_XMT_DATA8</td>
<td>“TWI FIFO Transmit Data Single Byte (TWIx_XMT_DATA8) Registers” on page 20-27</td>
</tr>
</tbody>
</table>
Table B-31. TWI 1 Registers (Cont’d)

<table>
<thead>
<tr>
<th>Memory-Mapped Address</th>
<th>Register Name</th>
<th>See Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0 2284</td>
<td>TWI1_XMT_DATA16</td>
<td>“TWI FIFO Transmit Data Double Byte (TWIx_XMT_DATA16) Registers” on page 20-27</td>
</tr>
<tr>
<td>0xFFC0 2288</td>
<td>TWI1_RCV_DATA8</td>
<td>“TWI FIFO Receive Data Single Byte (TWIx_RCV_DATA8) Registers” on page 20-28</td>
</tr>
<tr>
<td>0xFFC0 228C</td>
<td>TWI1_RCV_DATA16</td>
<td>“TWI FIFO Receive Data Double Byte (TWIx_RCV_DATA16) Registers” on page 20-29</td>
</tr>
</tbody>
</table>
Two-Wire Interface Registers
This chapter discusses the test features of the processor.

**JTAG Standard**

The processor is fully compatible with the IEEE 1149.1 standard, also known as the Joint Test Action Group (JTAG) standard.

The JTAG standard defines circuitry that may be built to assist in the test, maintenance, and support of assembled printed circuit boards. The circuitry includes a standard interface through which instructions and test data are communicated. A set of test features is defined, including a boundary-scan register, such that the component can respond to a minimum set of instructions designed to help test printed circuit boards.

The standard defines test logic that can be included in an integrated circuit to provide standardized approaches to:

- Testing the interconnections between integrated circuits once they have been assembled onto a printed circuit board
- Testing the integrated circuit itself
- Observing or modifying circuit activity during normal component operation

The test logic consists of a boundary-scan register and other building blocks. The test logic is accessed through a test access port (TAP).

### Boundary-Scan Architecture

The boundary-scan test logic consists of:

- A TAP comprised of five pins (see Table C-1)
- A TAP controller that controls all sequencing of events through the test registers
- An instruction register (IR) that interprets 5-bit instruction codes to select the test mode that performs the desired test operation
- Several data registers defined by the JTAG standard

#### Table C-1. Test Access Port Pins

<table>
<thead>
<tr>
<th>Pin Name</th>
<th>Input/Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDI</td>
<td>Input</td>
<td>Test data input</td>
</tr>
<tr>
<td>TMS</td>
<td>Input</td>
<td>Test mode select</td>
</tr>
<tr>
<td>TCK</td>
<td>Input</td>
<td>Test clock</td>
</tr>
<tr>
<td>TRST</td>
<td>Input</td>
<td>Test reset</td>
</tr>
<tr>
<td>TDO</td>
<td>Output</td>
<td>Test data out</td>
</tr>
</tbody>
</table>

The TAP controller is a synchronous, 16-state, finite-state machine controlled by the TCK and TMS pins. Transitions to the various states in the diagram occur on the rising edge of TCK and are defined by the state of the TMS pin, here denoted by either a logic 1 or logic 0 state. For full details of the operation, see the JTAG standard.
Figure C-1 shows the state diagram for the TAP controller.

Note:

- The TAP controller enters the test-logic-reset state when $TMS$ is held high after five $TCK$ cycles.
- The TAP controller enters the test-logic-reset state when $TRST$ is asynchronously asserted.
• An external system reset does not affect the state of the TAP controller, nor does the state of the TAP controller affect an external system reset.

Instruction Register

The instruction register is five bits wide and accommodates up to 32 boundary-scan instructions.

The instruction register holds both public and private instructions. The JTAG standard requires some of the public instructions; other public instructions are optional. Private instructions are reserved for the manufacturer’s use.

The binary decode column of Table C-2 lists the decode for the public instructions. The register column lists the serial scan paths.

Table C-2. Decode for Public JTAG-Scan Instructions

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Binary Decode 01234</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXTEST</td>
<td>00000</td>
<td>Boundary-scan</td>
</tr>
<tr>
<td>SAMPLE/PRELOAD</td>
<td>10000</td>
<td>Boundary-scan</td>
</tr>
<tr>
<td>BYPASS</td>
<td>11111</td>
<td>Bypass</td>
</tr>
</tbody>
</table>
Figure C-2 shows the instruction bit scan ordering for the paths shown in Table C-2.

Figure C-2. Serial Scan Paths
Boundary-Scan Architecture

Public Instructions

The following sections describe the public JTAG scan instructions.

**EXTEST – Binary Code 00000**

The **EXTEST** instruction selects the boundary-scan register to be connected between the **TDI** and **TDO** pins. This instruction allows testing of on-board circuitry external to the device.

**EXTEST** instruction allows internal data to be driven to the boundary outputs and external data to be captured on the boundary inputs.

🚫 To protect the internal logic when the boundary outputs are overdriven or signals are received on the boundary inputs, make sure that nothing else drives data on the processor’s output pins.

**SAMPLE/PRELOAD – Binary Code 10000**

The **SAMPLE/PRELOAD** instruction performs two functions and selects the boundary-scan register to be connected between **TDI** and **TDO**. The instruction has no effect on internal logic.

**SAMPLE** part of the instruction allows a snapshot of the inputs and outputs captured on the boundary-scan cells. Data is sampled on the rising edge of **TCK**.

**PRELOAD** part of the instruction allows data to be loaded on the device pins and driven out on the board with the **EXTEST** instruction. Data is pre-loaded on the pins on the falling edge of **TCK**.

**BYPASS – Binary Code 11111**

The **BYPASS** instruction selects the **BYPASS** register to be connected to **TDI** and **TDO**. The instruction has no effect on the internal logic. No data inversion should occur between **TDI** and **TDO**.
Test Features

Boundary-Scan Register

The boundary-scan register is selected by the **EXTEST** and **SAMPLE/PRELOAD** instructions. These instructions allow the pins of the processor to be controlled and sampled for board-level testing.
Boundary-Scan Architecture
D NUMERIC FORMATS

Blackfin family processors support 8-, 16-, 32-, and 40-bit fixed-point data in hardware. Special features in the computation units allow support of other formats in software. This appendix describes various aspects of these data formats. It also describes how to implement a block floating-point format in software.

Unsigned or Signed: Two’s-Complement Format

Unsigned integer numbers are positive, and no sign information is contained in the bits. Therefore, the value of an unsigned integer is interpreted in the usual binary sense. The least significant words of multiple-precision numbers are treated as unsigned numbers.

Signed numbers supported by the Blackfin family are in two’s-complement format. Signed-magnitude, one’s-complement, binary-coded decimal (BCD) or excess-n formats are not supported.

Integer or Fractional

The Blackfin family supports both fractional and integer data formats. In an integer, the radix point is assumed to lie to the right of the least significant bit (LSB), so that all magnitude bits have a weight of 1 or greater. This format is shown in Figure D-1. Note in two’s-complement format, the sign bit has a negative weight.
In a fractional format, the assumed radix point lies within the number, so that some or all of the magnitude bits have a weight of less than 1. In the format shown in Figure D-2, the assumed radix point lies to the left of the three LSBs, and the bits have the weights indicated.

The native formats for the Blackfin processor family are a signed fractional 1.M format and an unsigned fractional 0.N format, where N is the number of bits in the data word and M = N – 1.

The notation used to describe a format consists of two numbers separated by a period (.), the first number is the number of bits to the left of the radix point, the second is the number of bits to the right of the radix point. For example, 16.0 format is an integer format; all bits lie to the left of the radix point. The format in Figure D-2 is 13.3.

**Figure D-1. Integer Format**

<table>
<thead>
<tr>
<th>Bit</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight</td>
<td>- (2^{15})</td>
<td>(2^{14})</td>
<td>(2^{13})</td>
<td>...</td>
<td>(2^{2})</td>
<td>(2^{1})</td>
</tr>
<tr>
<td>Signed Bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight</td>
<td>(2^{15})</td>
<td>(2^{14})</td>
<td>(2^{13})</td>
<td>...</td>
<td>(2^{2})</td>
<td>(2^{1})</td>
</tr>
<tr>
<td>Signed Bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Radix Point | | | | | | |
Table D-1 shows the ranges of signed numbers representable in the fractional formats that are possible with 16 bits.

Figure D-2. Example of Fractional Format
Table D-1. Fractional Formats and Their Ranges

<table>
<thead>
<tr>
<th>Format</th>
<th># of Integer Bits</th>
<th># of Fractional Bits</th>
<th>Max Positive Value (0x7FFF) In Decimal</th>
<th>Max Negative Value (0x8000) In Decimal</th>
<th>Value of 1 LSB (0x0001) In Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.15</td>
<td>1</td>
<td>15</td>
<td>0.999969482421875</td>
<td>−1.0</td>
<td>0.000030517578125</td>
</tr>
<tr>
<td>2.14</td>
<td>2</td>
<td>14</td>
<td>1.999938964843750</td>
<td>−2.0</td>
<td>0.000061035156250</td>
</tr>
<tr>
<td>3.13</td>
<td>3</td>
<td>13</td>
<td>3.999877929687500</td>
<td>−4.0</td>
<td>0.000122070312500</td>
</tr>
<tr>
<td>4.12</td>
<td>4</td>
<td>12</td>
<td>7.9975585937500000</td>
<td>−8.0</td>
<td>0.0002441406250000</td>
</tr>
<tr>
<td>5.11</td>
<td>5</td>
<td>11</td>
<td>15.9995117187500000</td>
<td>−16.0</td>
<td>0.000488281250000000</td>
</tr>
<tr>
<td>6.10</td>
<td>6</td>
<td>10</td>
<td>31.9990234375000000</td>
<td>−32.0</td>
<td>0.000976562500000000</td>
</tr>
<tr>
<td>7.9</td>
<td>7</td>
<td>9</td>
<td>63.99804687500000000</td>
<td>−64.0</td>
<td>0.001953125000000000</td>
</tr>
<tr>
<td>8.8</td>
<td>8</td>
<td>8</td>
<td>127.9960937500000000000</td>
<td>−128.0</td>
<td>0.00390625000000000000</td>
</tr>
<tr>
<td>9.7</td>
<td>9</td>
<td>7</td>
<td>255.9921875000000000000</td>
<td>−256.0</td>
<td>0.00781250000000000000</td>
</tr>
<tr>
<td>10.6</td>
<td>10</td>
<td>6</td>
<td>511.98437500000000000000000000</td>
<td>−512.0</td>
<td>0.0156250000000000000000000000</td>
</tr>
<tr>
<td>11.5</td>
<td>11</td>
<td>5</td>
<td>1023.96875000000000000000000000000000</td>
<td>−1024.0</td>
<td>0.031250000000000000000000000000000</td>
</tr>
<tr>
<td>12.4</td>
<td>12</td>
<td>4</td>
<td>2047.93750000000000000000000000000000</td>
<td>−2048.0</td>
<td>0.062500000000000000000000000000000</td>
</tr>
<tr>
<td>13.3</td>
<td>13</td>
<td>3</td>
<td>4095.87500000000000000000000000000000</td>
<td>−4096.0</td>
<td>0.125000000000000000000000000000000</td>
</tr>
<tr>
<td>14.2</td>
<td>14</td>
<td>2</td>
<td>8191.7500000000000000000000000000000000</td>
<td>−8192.0</td>
<td>0.2500000000000000000000000000000000</td>
</tr>
<tr>
<td>15.1</td>
<td>15</td>
<td>1</td>
<td>16383.50000000000000000000000000000000</td>
<td>−16384.0</td>
<td>0.5000000000000000000000000000000000</td>
</tr>
<tr>
<td>16.0</td>
<td>16</td>
<td>0</td>
<td>32767.000000000000000000000000000000000000</td>
<td>−32768.0</td>
<td>1.0000000000000000000000000000000000</td>
</tr>
</tbody>
</table>

Binary Multiplication

In addition and subtraction, both operands must be in the same format (signed or unsigned, radix point in the same location), and the result format is the same as the input format. Addition and subtraction are performed the same way whether the inputs are signed or unsigned.
Numeric Formats

In multiplication, however, the inputs can have different formats, and the result depends on their formats. The Blackfin family assembly language allows you to specify whether the inputs are both signed, both unsigned, or one of each (mixed-mode). The location of the radix point in the result can be derived from its location in each of the inputs. This is shown in Figure D-3. The product of two 16-bit numbers is a 32-bit number. If the inputs’ formats are M.N and P.Q, the product has the format (M + P).(N + Q). For example, the product of two 13.3 numbers is a 26.6 number. The product of two 1.15 numbers is a 2.30 number.

<table>
<thead>
<tr>
<th>General Rule</th>
<th>4-bit Example</th>
<th>16-bit Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>M.N</td>
<td>1.111 (1.3 Format)</td>
<td>5.3</td>
</tr>
<tr>
<td>x P.Q</td>
<td>x 11.11 (2.2 Format)</td>
<td>x 5.3</td>
</tr>
<tr>
<td>(M + P).(N + Q)</td>
<td>1111</td>
<td>10.6</td>
</tr>
<tr>
<td></td>
<td>1111</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1111</td>
<td>1.15</td>
</tr>
<tr>
<td></td>
<td>1111</td>
<td>x 1.15</td>
</tr>
<tr>
<td></td>
<td>111.00001 (3.5 Format = (1 + 2).(2 + 3))</td>
<td>2.30</td>
</tr>
</tbody>
</table>

Figure D-3. Format of Multiplier Result

**Fractional Mode and Integer Mode**

A product of 2 two’s-complement numbers has two sign bits. Since one of these bits is redundant, you can shift the entire result left one bit. Additionally, if one of the inputs was a 1.15 number, the left shift causes the result to have the same format as the other input (with 16 bits of additional precision). For example, multiplying a 1.15 number by a 5.11 number yields a 6.26 number. When shifted left one bit, the result is a 5.27 number, or a 5.11 number plus 16 LSBs.
The Blackfin family provides a means (a signed fractional mode) by which the multiplier result is always shifted left one bit before being written to the result register. This left shift eliminates the extra sign bit when both operands are signed, yielding a result that is correctly formatted.

When both operands are in 1.15 format, the result is 2.30 (30 fractional bits). A left shift causes the multiplier result to be 1.31 which can be rounded to 1.15. Thus, if you use a signed fractional data format, it is most convenient to use the 1.15 format.

For more information about data formats, see the data formats listed in Table 2-2 on page 2-12.
Block Floating-Point Format

A block floating-point format enables a fixed-point processor to gain some of the increased dynamic range of a floating-point format without the overhead needed to do floating-point arithmetic. However, some additional programming is required to maintain a block floating-point format.

A floating-point number has an exponent that indicates the position of the radix point in the actual value. In block floating-point format, a set (block) of data values share a common exponent. A block of fixed-point values can be converted to block floating-point format by shifting each value left by the same amount and storing the shift value as the block exponent.

Typically, block floating-point format allows you to shift out non-significant MSBs (most significant bits), increasing the precision available in each value. Block floating-point format can also be used to eliminate the possibility of a data value overflowing. See Figure D-4. Each of the three data samples shown has at least two non-significant, redundant sign bits. Each data value can grow by these two bits (two orders of magnitude) before overflowing. These bits are called guard bits.

![Figure D-4. Data With Guard Bits](image)

To detect bit growth into two guard bits, set $SB = -2$.
Block Floating-Point Format

If it is known that a process will not cause any value to grow by more than the two guard bits, then the process can be run without loss of data. Later, however, the block must be adjusted to replace the guard bits before the next process.

Figure D-5 shows the data after processing but before adjustment. The block floating-point adjustment is performed as follows.

- Assume the output of the `SIGNBITS` instruction is $SB$ and $SB$ is used as an argument in the `EXPADJ` instruction (see Blackfin Processor Programming Reference for the usage and syntax of these instructions). Initially, the value of $SB$ is +2, corresponding to the two guard bits. During processing, each resulting data value is inspected by the `EXPADJ` instruction, which counts the number of redundant sign bits and adjusts $SB$ if the number of redundant sign bits is less than two. In this example, $SB = +1$ after processing, indicating the block of data must be shifted right one bit to maintain the two guard bits.

- If $SB$ were 0 after processing, the block would have to be shifted two bits right. In either case, the block exponent is updated to reflect the shift.
1. Check for bit growth

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Sign Bit</th>
<th>Exponent</th>
<th>EXPADJ Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1FFF</td>
<td>0 0 0 1  1 1 1 1  1 1 1 1  1 1 1 1</td>
<td>+2, +2</td>
<td>Exponent = +2, SB = +2</td>
</tr>
<tr>
<td>0x3FFF</td>
<td>0 0 1 1  1 1 1 1  1 1 1 1  1 1 1 1</td>
<td>+1, +1</td>
<td>Exponent = +1, SB = +1</td>
</tr>
<tr>
<td>0x07FF</td>
<td>0 0 0 0  0 1 1 1  1 1 1 1  1 1 1 1</td>
<td>+4, +1</td>
<td>Exponent = +4, SB = +1</td>
</tr>
</tbody>
</table>

2. Shift right to restore guard bits

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Sign Bit</th>
<th>Exponent</th>
<th>EXPADJ Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0FFF</td>
<td>0 0 0 0  1 1 1 1  1 1 1 1  1 1 1 1</td>
<td>+2, +2</td>
<td>Exponent = +2, SB = +2</td>
</tr>
<tr>
<td>0x1FFF</td>
<td>0 0 0 1  1 1 1 1  1 1 1 1  1 1 1 1</td>
<td>+1, +1</td>
<td>Exponent = +1, SB = +1</td>
</tr>
<tr>
<td>0x03FF</td>
<td>0 0 0 0  0 0 1 1  1 1 1 1  1 1 1 1</td>
<td>+4, +1</td>
<td>Exponent = +4, SB = +1</td>
</tr>
</tbody>
</table>

Figure D-5. Block Floating-Point Adjustment
Block Floating-Point Format
INDEX

Numerics

16-bit operations, 2-26, 2-35
16-bit SDRAM Bank, 18-51
32-bit operations, 2-28

A

A10 pin of SDRAM, 18-33
Abort Acknowledge Register (CANAA), 19-56
AC (Address Calculation), 4-7
Acceptance Filter / Data Acceptance Filter, 19-39
Acceptance Mask Register, 19-40
accumulator result registers A[1: 0], 2-6, 2-38, 2-45
active low/high frame syncs, serial port, 13-41
Active mode, 1-26, 8-13
ACTIVE_PLLDISABLED bit, 8-9
ACTIVE_PLLENABLED bit, 8-9
Active Video Only mode, PPI, 11-19
address bus, 18-62
Address Calculation (AC), 4-7
address compare block, 20-3
addressing
  modes, 5-16
  transfers supported (table), 5-14
addressing (continued)
  See also auto-decrement; auto-increment;
  bit-reversed; circular-buffer; indexed;
  indirect; modified; post-increment;
  post-modify; pre-modify; data address
  generators
  address mapping, SDRAM, 18-50
  Address Not Acknowledged bit, 20-17
  address pointer registers. See pointer
  registers
  address-tag compare operation, 6-14
  alarm clock, RTC, 17-2
  A-law companding, 13-2, 13-37, 13-62
  alignment exceptions, 6-69
  alignment of memory operations, 6-68
  allocating system stack, 4-66
  alternate frame sync mode, 13-44
  alternate timing, serial port, 13-43
  ALU, 2-1, 2-25 to 2-37
    arithmetic, 2-13
    arithmetic formats, 2-15
    data flow, 2-33
    data types, 2-13
    functions, 2-25
    inputs and outputs, 2-25
    instructions, 2-25, 2-29, 2-37
    operations, 2-25 to 2-28
    status, 2-23
    status signals, 2-36
  AMC
    EBIU block diagram, 18-2
    timing parameters, 18-11
AMCKEN bit, 18-10
AMS, 18-9
ANAK, 20-17
AND, logical, 2-25
arbitration
  DAB, 7-7
  EAB, 7-11
  latency, 7-10
architecture, processor core, 2-2
ARDY pin, 18-16, 18-21
arithmetic formats summary, 2-15 to 2-16
Arithmetic Logic Unit (ALU), 1-5
arithmetic logic unit (ALU)
  instructions, 2-29
Arithmetic Logic Unit. See ALU
arithmetic operations, 2-25
arithmetic shift (ASHIFT) instruction, 2-50
arithmetic shifts, 2-1, 2-15
Arithmetic Status register (ASTAT), 2-23
ASIC/FPGA designs, 18-1
assembly language, 2-1
ASTAT (Arithmetic Status register), 2-23
asynchronous accesses, by core, 18-17
asynchronous controller, 1-15
asynchronous interfaces supported, 18-1
asynchronous memory, 18-2, 18-9
Asynchronous Memory Bank Address
  Range (table), 18-9
Asynchronous Memory Bank Control
  registers (EBIU_AMBCTLx), 18-11
asynchronous memory controller. See AMC
Asynchronous Memory Global Control
  register (EBIU_AMGCTL), 18-10
asynchronous read, 18-17
asynchronous serial communications, 12-2
asynchronous write, 18-19
ASYNC memory banks, 18-3
atomic operations, 6-69
autobaud, and general-purpose timers, 16-34
autobaud detection, 12-2, 16-34
auto-decrement addressing, 5-11
auto-increment addressing, 5-11
Auto-Refresh
  command, 18-59
  timing, 18-47
avoiding bus contention, 18-15

B
bank
  address, 18-46
  size, 18-46
  size encodings (table), 18-50
Bank Activate command, 18-24, 18-57
  selecting delay for, 18-41
bank size, 18-30
bank width, 18-30
barrel-shifter. See shifter
Base registers (B[3:0]), 2-8, 5-2, 5-7
baud rate, UART, 12-6, 12-7, 12-13
baud rate values, SPI, 10-8
B (Base) registers, 5-7
biased rounding, 2-18
BI bit, 12-6
binary decode, C-4
binary multiplication, D-5
binary numbers, 2-3
Bit Configuration Register 0 (CANCR0), 19-14
Bit Configuration Register 1 (CANCR1), 19-15
bit manipulation
  bit clear, 2-53
  bit set, 2-53
  bit test, 2-53
  bit toggle, 2-53
bit order, selecting, 13-36
bit-reversed addressing, 5-9
bit-reversed carry addressing, 5-1
Blackfin processor family
  core architecture, 1-1
  dynamic power management, 1-1
  memory architecture, 1-9
  native formats, D-2
block diagrams
  bus hierarchy, 7-1
  core, 7-3
  core timer, 16-44, 16-45
  EBIU, 18-2
  interrupt processing, 4-22
  PLL, 8-3
  processor, 1-5
  RTC, 17-2
  SDRAM, 18-31
  SPI, 10-2
  SPORT, 13-5
block floating point format, D-7
BMODE
  bits, 3-14
  state, 3-14
BMODE pins, 4-45
booting, 21-2
boot kernel, 3-19
boot modes, 1-28
boot ROM
  loading user code, 3-19
  reading in user code, 3-19
boundary-scan architecture, C-2
Boundary-Scan register, C-7
branch, 4-9
branch, conditional, 4-13
branch latency, 4-10
  conditional branches, 4-14
  unconditional branches, 4-15
branch prediction, 4-14
branch target, 4-12
branch target address
  unconditional branches, 4-15
B-registers (Base), 2-8, 5-2, 5-7
broadcast mode, 10-2, 10-14
Buffer Read Error bit, 20-17
buffers
  Cacheability Protection Lookaside
    Buffers (CPLBs), 6-11, 6-43, 6-44
    timing, external, 18-61
  Buffer Write Error bit, 20-16
  BUFRDERR, 20-17
  BUFWRERR, 20-16
  burst length, 18-24, 18-52
  burst stop command, 18-25
  burst type, 18-25
  bus agents
    DAB, 7-10
    PAB, 7-6
  BUSBUSY, 20-15
  Bus Busy bit, 20-15
  bus contention, avoiding, 18-15, 21-10
  bus error, EBIU, 18-8
buses
  hierarchy, 7-1
  loading, 21-12
  on-chip, 7-1
  peripheral, 7-5
  See also DAB, EAB, EMB, PAB
bus request and grant, 18-62
BYPASS field, 8-8
BYPASS instruction, C-6
Bypass mode, 3-19
Bypass register, C-6
byte
  address, 18-46
  byte enables, 18-21
  byte order, 2-13
C

cache
  coherency support, 6-69
  mapping into data banks, 6-31
  validity of cache lines, 6-12
Cacheability Protection Lookaside Buffers (CPLBs), 6-11, 6-43, 6-44
cache block (definition), 6-71
cache hit
  address-tag compare, 6-14
  data cache access, 6-34
  definition, 6-14, 6-71
cache inhibited accesses, 6-69
cache line
  components, 6-11
  definition, 6-71
  states, 6-34
cache miss
  definition, 6-34, 6-71
  replacement policy, 6-15
CALL instruction, 4-9, 4-11
  range, 4-11
CAN
  debug and test modes, 19-17
  test modes, 19-19
CAN_CEC (CAN error counter) register, 19-19
CAN_DEBUG (CAN debug) register, 19-17, 19-18
CAN Error Counter Register (CANCEC), 19-85
CAN Module Registers, 19-6
capacitive loads, 18-23, 21-12
capacitors, 21-12, 21-13
carry status, 2-36
CAS before RAS, 18-26
CAS latency, 18-25
  selecting, 18-40
CAW, 18-46
CBR refresh, 18-26

CCA CAN Configuration Mode
  Acknowledge, 19-12
CC flag bit, 4-12
CCIR-656. See ITU-R 656
CCITT G.711 specification, 13-37
CLK (core clock), 8-4
  disabling, 8-29
  status by operating mode, 8-12
CC status bit, 4-10
CDDBG bit, 18-37
CDE bit, 19-17
channels
  defined, serial, 13-61
  serial port TDM, 13-61
  serial select offset, 13-61
CHNL bit, 13-59
circuit board testing, C-1, C-6
circular buffer addressing, 5-6
  registers, 5-6
  wraparound, 5-9
CKELOW bit, 8-25
clean (definition), 6-72
CL field, 18-35, 18-40
CLKHI, 20-7
CLKIN (input clock), 8-1, 8-2
CLKIN to VCO, changing the multiplier, 8-19
CLKLO, 20-7
CLK_SEL bit, 16-15
clock
  EBIU, 18-1
  frequency for SPORT, 13-31
  managing, 21-4
  RTC, 17-2
  setting up, 13-35
  source for general-purpose timers, 16-2
types, 21-4
clock divide modulus register, 13-31
clock generation module, 20-4
Clock High bit, 20-7
Index

clocking, 8-1 to 8-10
  clock input (CLkin) pin, 21-4
  Clock Low bit, 20-7
  clock phase, SPI, 10-20, 10-22
  clock polarity, SPI, 10-20
  clock rate
    core timer, 16-44
    SPORT, 13-2
  clock signal, SPI, 10-4
  code examples
    Active mode to Full On mode, 8-22
    Epilog code for nested ISR, 4-59
    Execution Trace recreation, 22-18
    Full On mode to Active mode, 8-22
    interrupt enabling and disabling, 6-71
    load base of MMRs, 6-71
    loop, 4-16
    modification of PLL, 8-19
    Prolog code for nested ISR, 4-59
    restoration of the control register, 6-71
  code patching, 22-5
  column address, 18-46
  strobe latency, 18-25
  Command Inhibit command, 18-60
  commands
    Auto-Refresh, 18-48, 18-59
    Bank Activate, 18-24, 18-41, 18-57
    Burst Stop, 18-25
    Command Inhibit, 18-60
    Load Mode Register, 18-57
    No Operation, 18-60
    parallel refresh, 18-53
    Precharge, 18-27, 18-42, 18-56
    Read/Write, 18-58
    SDC, 18-55
    Self-Refresh, 18-28, 18-59
    Transfer Initiate, 10-25
  companding, 13-51, 13-62
    defined, 13-37
    lengths supported, 13-37
    multichannel operations, 13-62
  computational instructions, 2-1
  computational status, 2-23
  computational units, 2-1 to 2-57
  conditional
    branches, 4-13
    branch latency, 4-14
    instructions, 4-3
    JUMP instruction, 4-10
  conditional branches, 6-66, 6-67
  conditional instructions, 2-23
  condition code (CC) flag bit, 4-12
  configuration
    L1 Instruction Memory, 6-11
    L1 SRAM, 6-1
    SDC, 18-53
    SDRAM, 18-23
    SPORT, 13-10
    Content-Addressable Memory (CAM), 6-43
  contention, bus, avoiding, 18-15
  control bit summary, general-purpose timers, 16-42
  control register
    data memory, 6-24
    EBIU, 18-7
    instruction memory, 6-6
    restoration, 6-71
  convergent rounding, 2-19
  core
    access to flag configuration, 14-5, 15-4
    architecture, 2-2
    core clock/system clock ratio control, 8-4
    double-fault condition, 4-44
    double-fault reset, 3-13
    powering down, 8-29
    core and system reset, code example, 3-17
Index

core architecture, 1-5 to 1-8
core clock (CCLK), 8-4
core event
  in EVT, 4-43
  MMR location, 4-43
Core Event Controller (CEC), 4-18
Core Event Vector Table (table), 4-43
core instructions, asynchronous accesses, 18-17
Core Interrupt Latch register (ILAT), 4-40
Core Interrupt Mask register (IMASK), 4-39
Core Inturruptions Pending register (IPEND), 3-1, 4-41
core-only software reset, 3-13, 3-17, 3-19
core timer, 16-44 to 16-48
  block diagram, 16-45
  clock rate, 16-44
  scaling, 16-48
Core Timer Control register (TCNTL), 16-45
Core Timer Count register (TCOUNT), 16-46
Core Timer Period register (TPERIOD), 16-47
Core Timer Scale register (TSCALE), 16-48
counter
cycle, 4-4, 22-23
  RTC, 17-1
CROSSCORE software, 1-31
cross options, 2-35
crosstalk, 21-12
CSYNC, 6-66
Current Address registers
  (DMAx_CURR_ADDR), 9-18
Current Address registers
  (MDMA_yy_CURR_ADDR), 9-18
Current Descriptor Pointer registers
  (DMAx_CURR_DESC_PTR), 9-17
Current Descriptor Pointer registers
  (MDMA_yy_CURR_DESC_PTR), 9-17
Current Inner Loop Count registers
  (DMAx_CURR_X_COUNT), 9-19
Current Inner Loop Count registers
  (MDMA_yy_CURR_X_COUNT), 9-19
Current Outer Loop Count registers
  (DMAx_CURR_Y_COUNT), 9-20
Current Outer Loop Count registers
  (MDMA_yy_CURR_Y_COUNT), 9-20
cycle counters, 4-4, 22-23, 22-24
CYCLES and CYCLES2 (Execution Cycle Count registers), 22-24

D
DAB (DMA Access bus)
  arbitration, 7-7
  arbitration priority (table), 7-8
  bus agents (masters), 7-10
  latencies (table), 7-10
  performance, 7-9
DAG0 CPLB Miss, 4-50
DAG0 Misaligned Access, 4-50
DAG0 Multiple CPLB Hits, 4-50
DAG0 Protection Violation, 4-50
DAG1 CPLB Miss, 4-50
DAG1 Misaligned Access, 4-50
DAG1 Multiple CPLB Hits, 4-50
DAG1 Protection Violation, 4-50
DAG. See data address generators (DAGs)
Data Address Generators (DAGs)
  support for branches, 4-3
data address generators (DAGs), 5-1 to 5-21
  addressing modes, 5-16
  instructions, 5-17
  register modification, 5-13
  registers, 2-5, 2-7
data bursts, DMA, 7-9
data bus, 18-62
data cache control instructions, 6-37
data corruption, avoiding with SPI, 10-23
data-driven interrupts, 9-26
data flow, 2-1
data formats, 2-3 to 2-4, 2-11, 2-12
  binary multiplication, D-5
  SPORT, 13-36
data input modes for PPI, 11-22 to 11-25
data mask encodings, 18-52
data memory, L1, 6-24 to 6-38
Data Memory Control register
  (DMEM_CONTROL), 6-24, 6-44
data move, serial port operations, 13-46
Data Not Acknowledged bit, 20-17
data operations, CPLB, 6-44
data output modes for PPI, 11-25 to 11-27
data overflow, 13-39
data register file, 2-5, 2-6
data registers, 2-5, 3-4
data sampling, serial, 13-41
Data SRAM
  L1, 6-27
Data Storage, 19-21
data store format, 6-72
Data Test Command register
  (DTEST_COMMAND), 6-40
Data Test Data registers
  (DTEST_DATAx), 6-41
Data Test registers, 6-39 to 6-42
Data Transfer Count bit, 20-12
data transfers
  data register file, 2-6
  SPI, 10-2
data types, 2-11 to 2-22
data underflow, 13-39
Data Watch Point Address Control register
  (WPDACTL), 22-12
Data Watch Point Address Count Value registers (WPDACNTn), 22-11
Data Watch Point Address registers (WPDAAn), 22-11
data word
  serial data formats, 13-22
  UART, 12-6
DBGCTL (Debug Control register), 3-17
DCBS bit, 6-25
DCBS (L1 Data Cache Bank Select) bit, 6-32
DCNT, 20-12
DCPLB Address registers (DCPLB_ADDRx), 6-56
DCPLB_ADDRx (DCPLB Address registers), 6-56
DCPLB Data registers (DCPLB_DATAx), 6-53
DCPLB_DATAx (DCPLB Data registers), 6-53
DCPLB_FAULT_ADDR (DCPLB Fault Address register), 6-60
DCPLB Fault Address register (DCPLB_FAULT_ADDR), 6-60
DCPLB_STATUS (DCPLB Status register), 6-59
DCPLB Status register (DCPLB_STATUS), 6-59
Debug Control register (DBGCTL), 3-17
deg; debug features, 22-1
DEC bit, 19-19
DEC (Instruction Decode), 4-7
Deep Sleep mode, 1-27, 8-14
Index

deferring exception processing, 4-61
Delay Count register (PPI_DELAY), 11-9
development tools, 1-31
DF bit, 8-3, 8-9
DF (Divide Frequency), 8-3
DI_EN bit, 9-13
DIL bit, 19-18
direct-mapped (definition), 6-71
direct memory access (DMA), 1-13, 9-1 to 9-50
dirty (definition), 6-72
Disable Interrupts (CLI) instruction, 6-71
disabling
general-purpose timers, 16-5, 16-13
interrupts, global, 4-42
PLL, 8-15
RTC prescaler, 17-19
DISALGNEXPT instruction, 5-14, 5-15
DI_SEL bit, 9-13
DITFS bit, 13-16, 13-29, 13-45
divide primitives (DIVS, DIVQ), 2-13, 2-36
divisor, UART, 12-12
divisor reset, UART, 12-13
DIVQ instruction, 2-36
DIVS instruction, 2-36
DLAB bit, 12-8, 12-9
DLAN field, 11-3
DMA, 9-1 to 9-50
and SPI, 10-32 to 10-38
and SPORT, 13-3
and synchronization with PPI, 11-22
and UART, 12-9, 12-16
autobuffer mode, 9-28, 9-38
buffer size, multichannel SPORT, 13-65
channel registers, 9-28
channels, 9-39
continuous transfers using autobuffering, 9-42
descriptor array, 9-37
DMA (continued)
descriptor elements, 9-6
descriptor lists, 9-37
descriptor queue management, 9-45
descriptor structures, 9-44
direction, 9-14
DMA-capable peripherals, 9-2
DMA Error interrupt, 9-26
flex descriptor structure, 9-28
flow chart, 9-32
for SPI transmit, 10-17
memory DMA, 9-50
operation flow, 9-32
polling registers, 9-40
refresh, 9-37
register naming conventions, 9-5
serial port block transfers, 13-46
single-buffer transfers, 9-42
software management, 9-39
SPI data transmission, 10-17, 10-18
SPI master, 10-33
SPI slave, 10-36
startup, 9-35
stopping, 9-38
synchronization, 9-39, 9-40
two-dimensional (2D), 9-49
with PPI, 11-31
DMA2D bit, 9-13
DMA Bus. See DAB
DMA Configuration registers (DMAx_CONFIG), 9-11
DMA Configuration registers (MDMA_yy_CONFIG), 9-11
DMA_EN bit, 9-14
DMAx_CONFIG (DMA Configuration registers), 9-11
DMAx_CURR_ADDR (Current Address registers), 9-18
DMAx_CURR_DESC_PTR (Current Descriptor Pointer registers), 9-17
Index

DMAx_CURR_X_COUNT (Current Inner Loop Count registers), 9-19
DMAx_CURR_Y_COUNT (Current Outer Loop Count registers), 9-20
DMAx_IRQ_STATUS (Interrupt Status registers), 9-26
DMAx_NEXT_DESC_PTR (Next Descriptor Pointer registers), 9-8
DMAx_PERIPHERAL_MAP (Peripheral Map registers), 9-21
DMAx_START_ADDR (Start Address registers), 9-10
DMAx_X_COUNT (Inner Loop Count registers), 9-14
DMAx_X_MODIFY (Inner Loop Address Increment registers), 9-15
DMAx_Y_COUNT (Outer Loop Count registers), 9-16
DMAx_Y_MODIFY (Outer Loop Address Increment registers), 9-17
DMC field, 6-26
DMEM_CONTROL (Data Memory Control register), 6-24, 6-44
DNM, 20-17
double-fault condition, 4-44
DPMC (Dynamic Power Management Controller), 8-2, 8-11 to 8-31
dynamic power management, 1-26, 8-1 to 8-31
Dynamic Power Management Controller (DPMC), 8-2, 8-11 to 8-31

E
EAB (External Access bus), 7-10
and EBIU, 18-4
arbitration, 7-11
frequency, 7-11
performance, 7-11
early frame sync, 13-43
EBCAW field, 18-45
EBE bit, 18-45, 18-54
EBIU, 18-1 to 18-62
as slave, 18-4
asynchronous interfaces supported, 18-1
block diagram, 18-2
bus error, 18-8
byte enables, 18-21
clock, 18-1
clocking, 8-1
control registers, 18-7
overview, 18-1
programmable timing characteristics, 18-17
request priority, 18-1
SDRAM devices supported, 18-45
status register, 18-7
EBIU_AMBCTL0 (Asynchronous Memory Bank Control 0 register), 18-11
EBIU_AMBCTL1 (Asynchronous Memory Bank Control 1 register), 18-11
EBIU_AMGCTL (Asynchronous Memory Global Control register), 18-10
EBIU_SDBCTL (SDRAM Memory Bank Control register), 18-44
EBIU_SDGCTL (SDRAM Memory Global Control register), 18-33
EBIU_SDRRC (SDRAM Refresh Rate Control register), 18-47
EBIU_SDSSTAT (SDRAM Control Status register), 18-47
EBSZ field, 18-50
EBUFE bit, 18-36
setting, 18-39
ELSI bit, 12-10
EMISO bit, 10-9
EMREN bit, 18-36
EMU core event, 4-19
emulation, and timer counter, 16-10
emulation event, 4-44
Emulation mode, 3-9, 4-44
Enable Interrupts (STI) instruction, 6-71, 8-21
enabling
general-purpose timers, 16-4, 16-13
enabling interrupts, global, 4-42
ENDCPLB bit, 6-26
endianess, 2-13
endian format
data and instruction storage, 6-62
entire field mode, PPI, 11-18
EPROM, 1-10
ERBFI bit, 12-8
ERR_DET bit, 11-8
ERR_NCOR bit, 11-8
errors
bus parity, 4-52
bus timeout, 4-52
hardware, 4-52
hardware conditions causing, 4-52
internal core, 4-52
errors (continued)
misalignment of data, 6-68
multiple hardware, 4-52
peripheral, 4-52
error signals, SPI, 10-28 to 10-30
ERR_TYP field, 16-8
ETBEI bit, 12-8
evaluation of loop conditions, 4-5
event
definition, 4-18
exception, 4-46
latency in servicing, 4-66
nested, 4-41
event controller, 3-1, 4-18
MMRs, 4-39
sequencer, 4-3
event handling, 1-11, 4-18
event monitor, 22-22
event processing, 4-3
events by priority (table), 4-19
Event Vector Table (EVT), 4-43
EVT (Event Vector Table), 4-43
EVX core event, 4-19
EX1 (Execute 1), 4-7
EX2 (Execute 2), 4-7
EX3 (Execute 3), 4-7
EX4 (Execute 4), 4-7
exception, 4-1
deferring, 4-61
events, 4-46
events causing, 4-47
handler, executing, 4-51
handling instructions in pipeline, 4-61
multiple, 4-50
while exception handler executing, 4-51
exception events, 3-4
exception routine, example code, 4-64
Exceptions by Descending Priority (table), 4-50
exclusive (definition), 6-72
Index

EXCEPT instruction, 4-50
Execute 1 (EX1), 4-7
Execute 2 (EX2), 4-7
Execute 3 (EX3), 4-7
Execute 4 (EX4), 4-7
Execution Cycle Count registers (CYCLES and CYCLES2), 22-24
Execution Unit, components, 4-9
EXT_CLK mode, 16-35 to 16-37
external, 1-10
External Access Bus. See EAB
external buffer timing, 18-61
External Bus Interface Unit, 1-14
External Bus Interface Unit. See EBIU
External Event mode. See EXT_CLK mode
e external memory, 1-10, 6-43
design issues, 21-7
interfaces, 18-5
external memory interfaces, 18-5
External Memory Map (figure), 18-3
e external SDRAM memory, 18-50
EXTEST instruction, C-6

F
FAST, 20-13
Fast Fourier Transform, 2-35, 5-9
Fast Mode bit, 20-13
FBBRW bit, 18-36
FE bit, 12-6
d fetch address, 4-8
incrementation, 4-8
d fetched address, 4-3
FFE bit, 12-14
FIFO, 18-1
Flag Clear register (PORTCIO_CLEAR), 15-13
Flag Clear register (PORTFIO_CLEAR), 14-8
Flag Configuration register, core access to, 14-5, 15-4
Flag Data register (PORTFIO), 14-8
Flag Data register (PORTxIO), 15-12
Flag Direction register (PORTFIO_DIR), 14-5, 15-5, 15-6
Flag Input Enable register (PORTFIO_INEN), 14-21, 15-9
Flag Interrupt Sensitivity register (PORTFIO_EDGE), 14-19
Flag Mask Interrupt A Clear register (PORTFIO_MASKA_CLEAR), 14-11
Flag Mask Interrupt A Data register (PORTFIO_MASKA), 14-15
Flag Mask Interrupt A Set register (PORTFIO_MASKA_SET), 14-11
Flag Mask Interrupt A Toggle register (PORTFIO_MASKA_TOGGLE), 14-15
Flag Mask Interrupt B Clear register (PORTFIO_MASKB_CLEAR), 14-11
Flag Mask Interrupt B Data register (PORTFIO_MASKB), 14-17
Flag Mask Interrupt B Set register (PORTFIO_MASKB_SET), 14-11
Flag Mask Interrupt B Toggle register (PORTFIO_MASKB_TOGGLE), 14-17
Flag Mask Interrupt registers, 14-11
Flag Polarity register (PORTFIO_POLAR), 14-18
flags, interrupt generation, 14-1, 14-13
flags, overflow, 2-13
Flag Set on Both Edges register (PORTFIO_BOTH), 14-20
Flag Set register (PORTFIO_SET), 14-8, 15-13
Flag Toggle register (PORTFIO_TOGGLE), 14-9, 15-17
Flag Value registers, 14-6
Index

flash memory, 1-10, 18-1
FLD bit, 11-9
FLD_SEL bit, 11-6
flex descriptor structure, 9-28
FLOW[2:0] field, 9-12
FLSx bits, 10-12
FLUSH instruction, 6-38
FLUSHINV instruction, 6-38
Force Interrupt / Reset (RAISE) instruction, 3-11
FPE bit, 12-14
fractional data format, D-1
fractional mode, 2-14, D-6
fractional multiplier results format, 2-17
fractional representation, 2-4
fractions, multiplication, 2-47
framed serial transfers, characteristics, 13-40
framed/unframed data, 13-39
Frame Pointer (FP), 4-4
Frame Pointer (FP) registers, 5-5
frame start detect, PPI, 11-12
frame sync, 13-1
  active high/low, 13-41
  early, 13-43
  early/late, 13-43
  external/internal, 13-40
  internal, 13-34
  internally generated, 13-32
  late, 13-43
  multichannel mode, 13-55
  sampling edge, 13-41
  setting up, 13-35
  SPORT options, 13-39
frame synchronization and SPORT, 13-3
  PPI in GP modes, 11-28
frame sync polarity, PPI and Timer, 11-29
frame sync pulse
  use of, 13-16
  when issued, 13-16
frame sync signal, control of, 13-16, 13-21
Frame Track Error, 11-9
FREQ field, 8-26, 8-27
frequencies, clock and frame sync, 13-34
frequency
  EAB, 7-11
FSDR bit, 13-60
F signal, 11-9
FT_ERR bit, 11-9, 11-12
full duplex, 13-1, 13-4
full-duplex
  SPI, 10-1
FULL_ON bit, 8-9
Full On mode, 1-26, 8-12

G
GAIN field, 8-26, 8-28
GCALL, 20-10
GEN, 20-8
General Call bit, 20-10
General Call Enable bit, 20-8
general-purpose interrupt, 4-18, 4-54
  with multiple peripheral interrupts, 4-34
general-purpose I/O (GPIO)
pins, 14-1
general-purpose timers, 16-1 to 16-42
  and PPI, 16-37
  autobaud mode, 16-34
  clock source, 16-2
  control bit summary, 16-42
  disabling, 16-5, 16-13
  enabling, 16-4, 16-13
  enabling simultaneously, 16-3
  interrupts, 16-3, 16-6, 16-18, 16-37
  modes, 16-1
  output pad disable, 16-17
  PULSE_HI toggle mode, 16-21
Index

general-purpose timers  (continued)
   registers, 16-2
   single pulse generation, 16-17
   size of register accesses, 16-4
   stopping in PWM_OUT mode, 16-19
   waveform generation, 16-17
   glitch filtering, UART, 12-20
   global enabling and disabling interrupts, 4-42
Global Interrupt, 19-70
Global Interrupt Flag Register (CANGIF), 19-75
Global Interrupt Logic, 19-74
Global Status Register (CANGSR), 19-11
glueless connection, 21-7
GM bit, 10-27
ground plane, 21-12
GSM speech-compression routines, 2-22
GSM speech vocoder algorithms, 2-43

H
H.100, 13-2, 13-60, 13-66
hardware conditions and error interrupts, 4-53
Hardware Conditions Causing Hardware Error Interrupts (table), 4-53
hardware-error interrupt (HWE), 4-52
   causes, 4-52
hardware error interrupts, 4-53
   causes, 4-53
hardware errors, multiple, 4-52
hardware reset, 3-13
Harvard architecture, 6-4
Hibernate state, 1-28, 8-14, 8-29
   high frequency design considerations, 21-11
HMVIP, 13-66
   hold, for EBIU asynchronous memory controller, 18-11
HWE (hardware-error interrupt), 4-52

I
I2C bus standard, 20-1
I²S serial devices, 13-2
ICPLB Address registers
   (ICPLB_ADDRx), 6-57
ICPLB_ADDRx (ICPLB Address registers), 6-57
ICPLB Data registers (ICPLB_DATAx), 6-52
ICPLB_DATAx (ICPLB Data registers), 6-52
ICPLB Fault Address register
   (ICPLB_FAULT_ADDR), 6-60
ICPLB_FAULT_ADDR (ICPLB Fault Address register), 6-60
ICPLB_STATUS (ICPLB Status register), 6-60
ICPLB Status register (ICPLB_STATUS), 6-60
ICTL field, 16-52
Idle state, 3-10, 4-1
IEEE 1149.1 standard. See JTAG standard
IF1 (Instruction Fetch 1), 4-7
IF2 (Instruction Fetch 2), 4-7
IF3 (Instruction Fetch 3), 4-7
I-Fetch Access Exception, 4-50
I-Fetch CPLB Miss, 4-50
I-Fetch Misaligned Access, 4-50
I-Fetch Multiple CPLB Hits, 4-50
I-Fetch Protection Violation, 4-50
ILAT (Core Interrupt Latch register), 4-40
   illegal combination, 4-50
   illegal use protected resource, 4-50
IMASK (Core Interrupt Mask register), 4-39
IMEM_CONTROL (Instruction Memory Control register), 6-6, 6-44
   immediate overflow status, 2-36
   index (definition), 6-72
indexed addressing, 5-10
with immediate offset, 5-12
Index registers ([I[3:0]), 2-8, 5-2, 5-6
inductance (run length), 21-12
Inner Loop Address Increment registers (DMAx_X_MODIFY), 9-15
Inner Loop Address Increment registers (MDMA_yy_X_MODIFY), 9-15
Inner Loop Count registers (DMAx_X_COUNT), 9-14
Inner Loop Count registers (MDMA_yy_X_COUNT), 9-14
input clock (CLKin), 8-1
inputs and outputs, 2-25
instruction address, 4-3
Instruction Alignment Unit, 4-8
instruction-bit scan ordering, C-5
instruction cache
coherency, 6-17
invalidation, 6-18
management, 6-16 to 6-19
Instruction Decode (DEC), 4-7
Instruction Fetch 1 (IF1), 4-7
Instruction Fetch 2 (IF2), 4-7
Instruction Fetch 3 (IF3), 4-7
instruction fetches, 6-44
instruction fetch time loop, 4-17
instruction in pipeline when interrupt occurs, 4-61
instruction loop buffer, 4-17
Instruction Memory Control register (IMEM_CONTROL), 6-6, 6-44
Instruction Memory Unit, 4-8
instruction pipeline, 4-3, 4-7
Instruction register, C-2, C-4
instructions
ALU, 2-29, 2-31
conditional, 2-23, 4-3
DAG, 5-17
instruction set, 1-30
instruction (continued)
interlocked pipeline, 6-64
load store, 6-63
multiplier, 2-40
protected, 3-4
register file, 2-8
shifter, 2-54
stored in memory, 6-62
synchronizing, 6-66
instruction set, 1-30
Instruction Test Command register (ITEST_COMMAND), 6-21
Instruction Test Data registers (ITEST_DATAx), 6-22
Instruction Test registers, 6-20 to 6-23
Instruction Watch Point Address Control register (WPIACTL), 22-7
Instruction Watch Point Address Count registers (WPIACNTn), 22-5, 22-6
Instruction Watch Point Address registers (WPIAn), 22-5
instruction watch points, 22-4
instruction width, 4-8
integer data format, D-1
integer mode, 2-14, D-6
integer multiplier results format, 2-17
integers, multiplication, 2-47
interfaces, 7-5
external memory, 18-5
internal, 7-1
internal memory, 18-4
RTC, 17-2
Inter IC bus, 20-1
interleaving
of data in SPORT FIFO, 13-23
SPORT data, 13-5
Internal Address Mapping (table), 18-45
internal bank, 18-26
internal/external frame syncs. See frame sync
Index

internal memory, 1-10, 6-2
   interfaces, 18-4
internal supply regulator, shutting off, 8-29
interrupt, 4-1
   assigning priority for UART, 12-11
   buffer completion, 9-27
   control of system, 4-18
   definition, 4-18
   DMA error, 9-26
   enabling and disabling, 6-71
   general-purpose, 4-18, 4-54
   general-purpose timers, 16-3
   generated by peripheral, 4-21
   global enabling and disabling, 4-42
   hardware-error, 4-52
   multiple sources, 4-22
   nested, 4-41
   non-nested, 4-55
   peripheral, 4-18, 9-27
   PF pins, 14-1
   priority watermark, 6-36
   processing, 4-3, 4-21
   row completion, 9-27
   RTC, 17-12
   servicing, 4-54
   shared, 4-34
   sources, peripheral, 4-29
   SPI Error, 10-7
   SPORT Error, 13-30
   SPORT RX, 13-26, 13-30
   SPORT TX, 13-24, 13-30
interrupt channels, UART, 12-8
interrupt conditions, UART, 12-10
interrupt handling, instructions in pipeline, 4-61
interrupt output, SPI, 10-6
Interrupt Priority register (IPRIO), 6-36
Interrupt Register (CANINTR), 19-62
interrupts
   configuring and servicing, 21-4
   general-purpose timers, 16-18, 16-37
   hardware conditions (table), 4-53
   timers, 16-6
interrupt service routine
   determining source of interrupt, 4-29
Interrupt Status registers
   (DMAx_IRQ_STATUS), 9-26
Interrupt Status registers
   (MDMA_yy_IRQ_STATUS), 9-26
invalidating instructions, 4-8
invalidation
   of instruction cache, 6-18
   invalid cache line (definition), 6-72
I/O interface to peripheral serial device, 13-1
I/O memory space, 1-11
I/O pins, general-purpose, 14-1
IPEND (Core Interrupts Pending register), 3-1, 4-41
IPRIO (Interrupt Priority register), 6-36
IRCLK bit, 13-20
IrDA, 12-14
   receiver, 12-19
   SIR protocol, 12-1
   transmitter, 12-18
I-registers (Index), 2-8
IREN bit, 12-18
IRFS bit, 13-21, 13-40
IRPOL bit, 12-20
ISR and multiple interrupt sources, 4-22
Issue Stop Condition bit, 20-13
ITCLK bit, 13-15
ITEST_COMMAND (Instruction Test Command register), 6-21
ITEST_DATAX (Instruction Test Data registers), 6-22
ITEST registers, 6-20
ITFS bit, 13-16, 13-40, 13-56
ITU-R 656 modes, 11-6, 11-8, 11-13
and DLEN field, 11-3
frame start detect, 11-12
frame synchronization, 11-20
output, 11-20
IVG core events, 4-19
IVHW core event, 4-19
IVHW interrupt, 4-52
IVTMR core event, 4-19

J
JPEG compression, PPI, 11-32
JTAG
port, 3-17
standard, C-1, C-2, C-4
JUMP instruction, 4-9
conditional, 4-10
range, 4-10
jumps, 4-1

L
L1 Data SRAM, 6-27
L1 memory. See Level 1 (L1) memory;
Level 1 (L1) Data Memory; Level 1 (L1) Instruction Memory
LARFS bit, 13-21
latched interrupt request, 4-40
late frame sync, 13-43, 13-54
latency
DAB (table), 7-10
programmable flags, 14-22
SDRAM, 18-35
SDRAM Read command, 18-53
servicing events, 4-66
setting CAS value, 18-40
when servicing interrupts, 4-54
LATFS bit, 13-17, 13-43
LB (Loop Bottom registers), 4-5
LC (Loop Counter registers), 4-5
least recently used algorithm (LRU)
definition, 6-72
Length registers (L[3:0]), 2-8, 5-2, 5-7
Level 1 (L1) Data Memory, 6-24 to 6-38
subbanks, 6-27
traffic, 6-24
Level 1 (L1) Instruction Memory, 6-5 to 6-19
configuration, 6-11
DAG reference exception, 6-7
instruction cache, 6-11
organization, 6-11
subbank organization, 6-5
subbanks, 6-8
Level 1 (L1) memory, 6-4
address alignment, 6-7
definition, 6-72
scratchpad data SRAM, 6-5
See also Level 1 (L1) Data Memory; Level 1 (L1) Instruction Memory
Level 2 (L2) memory, 6-43
Lines Per Frame register (PPI_FRAME), 11-12
line terminations, SPORT, 13-67
little endian (definition), 6-72
load, speculative execution, 6-67
Load Mode Register command, 18-57
load operation, 6-63
load ordering, 6-64
load/store instructions, 5-5
locked transfers, DMA, 7-9
logging nested interrupt, 4-5
logical operations, 2-25
logical shift (LSHIFT) instruction, 2-50
logical shifts, 2-1, 2-15
long jump (JUMP.L) instruction, 4-11
loop, 4-1
buffer, 4-17
conditions, evaluation, 4-5
disabling, 4-17
loop

- instruction fetch time, 4-17
- registers, 4-4, 4-6
- termination, 4-3
- top and bottom addresses, 4-16

Loop Bottom registers (LB), 4-5
Loop Counter registers (LC), 4-5
Loop Top registers (LT), 4-5
LOSTARB, 20-17
Lost Arbitration bit, 20-17
L-registers (Length), 2-8
LRFS bit, 13-21, 13-40, 13-41
LTFS bit, 13-17, 13-40, 13-41
LT (Loop Top registers), 4-5

M

MACs (multiplier-accumulators), 2-37 to 2-50
- dual operations, 2-49
- multicycle 32-bit instruction, 2-48
  See also multiply without accumulate
Mailbox Area, 19-33
Mailbox Configuration (CANMC / CANMD), 19-35
Mailbox Control Logic, 19-35
Mailbox Interrupts, 19-62, 19-64
Mailbox Layout, 19-22
Mailbox Receive Interrupt Flag Register (CANMBRIF), 19-68
Mailbox Types, 19-34
Master Control Register (CANMCR), 19-6
Master Mode Enable bit, 20-13
master mode transmission (MEN), 20-32
masters
  DAB, 7-10
  PAB, 7-6
Master Transfer Complete bit, 20-23, 20-26
Master Transfer Direction bit, 20-13
Master Transfer Error bit, 20-23, 20-25
Master Transfer in Progress bit, 20-18
MBptr Mail Box Pointer, 19-12
MCMEN bit, 13-54
MCOMP, 20-23, 20-26
MDIR, 20-13
MDMA_yy_CONFIG (DMA Configuration registers), 9-11
MDMA_yy_CURR_ADDR (Current Address registers), 9-18
MDMA_yy_CURR_DESC_PTR (Current Descriptor Pointer registers), 9-17
MDMA_yy_CURR_X_COUNT (Current Inner Loop Count registers), 9-19
MDMA_yy_CURR_Y_COUNT (Current Outer Loop Count registers), 9-20
MDMA_yy_IRQ_STATUS (Interrupt Status registers), 9-26
MDMA_yy_NEXT_DESC_PTR (Next Descriptor Pointer registers), 9-8
MDMA_yy_PERIPHERAL_MAP (Peripheral Map registers), 9-21
MDMA_yy_START_ADDR (Start Address registers), 9-10
MDMA_yy_X_COUNT (Inner Loop Count registers), 9-14
MDMA_yy_X_MODIFY (Inner Loop Address Increment registers), 9-15
MDMA_yy_Y_COUNT (Outer Loop Count registers), 9-16
MDMA_yy_Y_MODIFY (Outer Loop Address Increment registers), 9-17
memory, 1-10
  address alignment, 5-14
  architecture, 1-9, 6-1 to 6-5
  asynchronous interface, 21-7
  asynchronous region, 18-2
memory (continued)

- external, 1-10, 6-43
- external memory, 18-5
- external SDRAM, 18-50
- how instructions are stored, 6-62
- internal, 1-10
- internal bank, 18-26
- L1 data, 6-24 to 6-38
- L1 Data SRAM, 6-27
- management, 6-43
- map, 6-2
- moving data between SPORT and, 13-46
- nonaligned operations, 6-68
- off-chip, 1-10
- Page Descriptor Table, 6-47
- protected, 3-5
- protection and properties, 6-43 to 6-60
- start locations of L1 Instruction Memory subbanks, 6-8
- terminology, 6-71 to 6-73
- transaction model, 6-62
- unpopulated, 18-9

See also cache; Level 1 (L1) memory;
Level 1 (L1) Data Memory; Level 1 (L1) Instruction Memory; Level 2 (L2) memory

Memory DMA, 9-50
channels, 9-50
register naming conventions, 9-7

Memory Management Unit (MMU), 6-43
memory map
- external (figure), 18-3
- memory-mapped registers (MMRs), 6-70 to 6-71
memory page, 6-45
- attributes, 6-45
MEN, 20-13
MERR, 20-23, 20-25
MFD field, 13-58

MISO pin, 10-3, 10-4, 10-5, 10-20, 10-23, 10-24, 10-27
μ-law companding, 13-2, 13-62
MMR location of core events, 4-43
mode
- boot, 1-28
- Emulation, 4-44
mode fault error, 10-7, 10-28
mode register, 18-26
modes
- boot, 3-19
- broadcast, 10-2, 10-14
- Bypass, 3-19
- multichannel, 13-50
- of general-purpose timers, 16-1
- serial port, 13-10
- SPI master, 10-2, 10-24
- SPI slave, 10-2, 10-26
- UART DMA, 12-16
- UART non-DMA, 12-15
- MODF bit, 10-28
- modified addressing, 5-4
- modified (definition), 6-72
- modify address, 5-1
- Modify registers (M[3:0]), 2-8, 5-2, 5-6
- MOSI pin, 10-3, 10-4, 10-5, 10-20, 10-23, 10-24, 10-27
- moving data, serial port, 13-46
- MPEG compression, PPI, 11-33
- MPROG, 20-18
- M-registers (Modify), 2-8
- MSEL (Multiplication Select) field, 8-8
- MSTR bit, 10-9
- multichannel frame, 13-57
- Multichannel Frame Delay field, 13-58
- multichannel mode, 13-50
- enable/disable, 13-54
- frame syncs, 13-55
- SPORT, 13-55

I-18 ADSP-BF538/ADSP-BF538F Blackfin Processor Hardware Reference
multichannel operation, SPORT, 13-50 to 13-66

multimaster environment
SPI, 10-2
multiple exception, for an instruction, 4-50
multiple interrupt sources, 4-22, 4-60
multiple slave SPI systems, 10-14
Multiplexed SDRAM Addressing Scheme (figure), 18-50

multiplier, 2-1
accumulator result registers A[1: 0], 2-38, 2-39
arithmetic integer modes formats, 2-16
data types, 2-14
fractional modes format, 2-16
instruction options, 2-42
instructions, 2-40
operands for input, 2-38
operations, 2-38
results, 2-39, 2-40, 2-44
rounding, 2-39
saturation, 2-39
status, 2-23
status bits, 2-40
Multiplier Select (MSEL) field, 8-3
multiply without accumulate, 2-46
MVIP-90, 13-66

N
NAK, 20-8
NAK bit, 20-8
NDSIZE[3:0] field, 9-13
negative status, 2-36
nested interrupt, 4-41
logging, 4-60
Nested Interrupt Handling (figure), 4-57
nested ISR
example Prolog code, 4-59
Next Descriptor Pointer registers (DMAx_NEXT_DESC_PTR), 9-8

Next Descriptor Pointer registers (MDMA_yy_NEXT_DESC_PTR), 9-8

NINT bit, 12-10
NMI, 4-45
core event, 4-19
nonaligned memory operations, 6-68
non-maskable interrupt, 4-45
non-nested interrupt, 4-55
Non-Nested Interrupt Handling (figure), 4-56

non-OS environments, 3-7
nonsequential program operation, 4-9
nonsequential program structures, 4-1
No Operation command, 18-60
NOP command, 18-60
normal frame sync mode, 13-44
normal timing, serial port, 13-43
NOT, logical, 2-25
numbers
   binary, 2-3
data formats, 2-12
   fractional representation, 2-4
two’s complement, 2-4
unsigned, 2-4
numeric formats, D-1 to D-8
   binary multiplication, D-5
   block floating point, D-7
   integer mode, D-6
two’s complement, D-1

O
OE bit, 12-6
off-chip memory, 1-10
OmniVision Serial Camera Control Bus (SCCB), 20-2
open drain drivers, 10-1
open drain outputs, 10-23
open page, 18-24
operating modes, 3-1 to 3-20, 8-11
   Active, 1-26, 8-13
   Deep Sleep, 1-27, 8-14
   Full On, 1-26, 8-12
   Hibernate state, 1-28, 8-14
   PPI, 11-6
   Sleep, 1-27, 8-13
   transition, 8-15
operating mode transitions, 8-18
OR, logical, 2-25
ordering
   loads and stores, 6-64
   weak and strong, 6-65
oscilloscope probes, 21-14
OUT_DIS bit, 16-8
Outer Loop Address Increment registers
   (DMAx_Y_MODIFY), 9-17
Outer Loop Address Increment registers
   (MDMA_yy_Y_MODIFY), 9-17
Outer Loop Count registers
   (DMAx_Y_COUNT), 9-16
Outer Loop Count registers
   (MDMA_yy_Y_COUNT), 9-16
output, PF pin configured as, 14-1
output, PPI, 1 sync mode, 11-25
output pad disable, timer, 16-17
overflow, data, 13-39
overflow, saturation of multiplier results, 2-40
overflow flags, 2-13
Overview, 19-1
Overwrite Protection / Single Shot
   Transmission Register (CANOPSS), 19-48
OVR bit, 11-9

**P**

PAB (Peripheral Access bus), 7-5
   and EBIU, 18-4
   arbitration, 7-5
   bus agents (masters, slaves), 7-6
   clocking, 8-1
   errors generated by SPORT, 13-31
   performance, 7-5
PACK_EN bit, 11-5
packing, serial port, 13-65
page size, 18-27, 18-46
Parallel Peripheral Interface (PPI). See PPI
parallel refresh command, 18-33
PC100 SDRAM standard, 18-1
PC133 SDRAM controller, 1-14
PC133 SDRAM standard, 18-1
PDWN bit, 8-8
PE bit, 12-6
PEMUSWx bits, 22-20
performance
   DAB, 7-9
   EAB, 7-11
   PAB, 7-5
   programmable flags, 14-22, 15-19
   SDRAM, 18-61
Performance Monitor Control register
   (PFCTL), 22-20
Performance Monitor Counter registers
   (PFCNTRn), 22-19
Performance Monitoring Unit, 22-19
PERIOD_CNT bit, 16-16
peripheral
   configuring for an IVG priority, 4-38
   interrupt generated by, 4-21
   interrupt sources, 4-29
Peripheral Bus. See PAB
peripheral DMA Start Address registers, 9-10
peripheral interrupts, 4-18
  relative priority, 4-34
  source masking, 4-32
Peripheral Map registers
  (DMAx_PERIPHERAL_MAP), 9-21
Peripheral Map registers
  (MDMA_yy_PERIPHERAL_MAP), 9-21
peripherals, 1-4 to 1-5
  compatible with SPI, 10-1
peripherals, timing, 7-2
PFCNTRn (Performance Monitor
  Counter registers), 22-19
PFCTL (Performance Monitor Control
  register), 22-20
PF pins, shared with PPI, 11-1
PFx pins, 10-12
Philips I2C, 20-1
pins, 21-1
  SPORT, 13-4
Pin State during SDC Commands (table), 18-56
pin terminations, SPORT, 13-67
pipeline
  instruction, 4-3, 4-7
  instructions when interrupt occurs, 4-61
  interlocked, 6-64
pipeline (figure), 4-8
pipelining, SDC supported, 18-39
PLL
  Active mode, 8-13
  Active mode, effect of programming for,
    8-20
  applying power to the PLL, 8-17
  block diagram, 8-3
  BYPASS bit, 8-13, 8-20
  CCLK derivation, 8-3
  changing CLKIN-to-VCO multiplier,
    8-17
  clock counter, 8-10
PLL (continued)
  clock dividers, 8-3
  clock frequencies, changing, 8-10
  clocking to SDRAM, 8-14
  clock multiplier ratios, 8-3
  code example, Active mode to Full On
    mode, 8-22
  code example, changing clock multiplier,
    8-23
  code example, Full On mode to Active
    mode, 8-22
  configuration, 8-3
  control bits, 8-15
  Deep Sleep mode, effect of programming
    for, 8-21
  disabled, 8-15
  Divide Frequency (DF) bit, 8-3
  DMA access, 8-12, 8-13, 8-21
  Dynamic Power Management Controller
    (DPMC), 8-11
  enabled, 8-15
  enabled but bypassed, 8-13
  Full On mode, effect of programming
    for, 8-20
  lock counter, 8-10
  maximum performance mode, 8-12
  modification, activating changes to DF
    or MSEL, 8-19
  modification in Active mode, 8-15
  Multiplier Select (MSEL) field, 8-3
  new multiplier ratio, 8-17
  operating modes, operational
    characteristics, 8-11
  operating mode transitions, 8-18
  operating mode transitions (figure), 8-17
  PDWN bit, 8-15
  PLL_LOCKED bit, 8-19
  PLL_OFF bit, 8-15
  PLL status (table), 8-11
  power domains, 8-23
PLL  (continued)
  powering down core, 8-29
  power savings by operating mode (table), 8-12
  processing during PLL programming sequence, 8-20
  programming sequence, 8-19
  relocking after changes, 8-19
  removing power to the PLL, 8-15
  RTC interrupt, 8-14, 8-21
  SCLK derivation, 8-1, 8-3
  Sleep mode, 8-13, 8-20
  STOPCK bit, 8-15
  transitions, 21-8
  voltage control, 8-11, 8-28
  wakeup signal, 8-20
PLL Control register (PLL_CTL), 8-7
PLL_CTL (PLL Control register), 8-7
PLL Divide register (PLL_DIV), 8-6
PLL_DIV (PLL Divide register), 8-6
PLL_LOCKCNT (PLL Lock Count register), 8-10
PLL_LOCKED bit, 8-9
PLL_OFF bit, 8-9
PLL_STAT (PLL Status register), 8-9
PLL_STAT (PLL status) register, 8-9
PLL Status register (PLL_STAT), 8-9
pointer register file, 2-5
pointer register modification, 5-13
pointer registers, 2-7, 3-4
point-to-point connections, 21-11
polarity, programmable flags, 14-18
POLC bit, 11-3
polling DMA registers, 9-40
POLS bit, 11-3
popping, manual, 4-4
PORT_CFG field, 11-6
PORTCIO_CLEAR (Flag Clear register), 15-13
port connection
  SPORT, 13-7
  PORT_DIR bit, 11-6
  PORT_EN bit, 11-8
  PORTFIO_BOTH (Flag Set on Both Edges register), 14-20
  PORTFIO_CLEAR (Flag Clear register), 14-8
  PORTFIO_DIR (Flag Direction register), 14-5
  PORTFIO_EDGE (Flag Interrupt Sensitivity register), 14-19
  PORTFIO (Flag Data register), 14-8
  PORTFIO_INEN (Flag Input Enable register), 14-21, 15-9
  PORTFIO_MASKA_CLEAR (Flag Mask Interrupt A Clear register), 14-11, 14-15, 14-17
  PORTFIO_MASKA (Flag Mask Interrupt A Data register), 14-15
  PORTFIO_MASKA_SET (Flag Mask Interrupt A Set register), 14-11, 14-15, 14-17
  PORTFIO_MASKA_TOGGLE (Flag Mask Interrupt A Toggle register), 14-15
  PORTFIO_MASKB_CLEAR (Flag Mask Interrupt B Clear register), 14-11
  PORTFIO_MASKB (Flag Mask Interrupt B Data register), 14-17
  PORTFIO_MASKB_SET (Flag Mask Interrupt B Set register), 14-11
  PORTFIO_MASKB_TOGGLE (Flag Mask Interrupt B Toggle register), 14-17
  PORTFIO_POLAR (Flag Polarity register), 14-18
  PORTFIO_SET (Flag Set register), 14-8, 15-13
PORTFIO_TOGGLE (Flag Toggle register), 14-9, 15-17
PORT_PREF0 bit, 6-24
PORT_PREF1 bit, 6-24
port width, PPI, 11-5
PORTxIO_DIR (Flag Direction register), 15-5, 15-6
PORTxIO (Flag Data register), 15-12
post-modify addressing, 5-1, 5-4, 5-7, 5-12
post-modify buffer access, 5-8
power dissipation, 8-23
power domains, 8-23
power-down warning, as NMI, 4-45
powering down core, 8-29
power management, 1-26, 8-1 to 8-31
power-up, 18-26
sequence, 18-57, 18-60
power-up sequence, SDRAM, 18-35
PPI, 11-1 to 11-33
Active Video Only mode, 11-19
and general-purpose timers, 16-37
and synchronization with DMA, 11-22
clock input, 11-1
clock input, 11-1
control signal polarities, 11-3
data input modes, 11-22 to 11-25
data output modes, 11-25 to 11-27
data width, 11-3
delay before starting, 11-10
DMA operation, 11-31
delay before starting, 11-10
data sensitive inputs, 11-30
delay before starting, 11-10
data width, 11-3
data input modes, 11-22 to 11-25
data output modes, 11-25 to 11-27
data width, 11-3
delay before starting, 11-10
DMA operation, 11-31
delay before starting, 11-10
data sensitive inputs, 11-30
delay before starting, 11-10
DMA operation, 11-31
delay before starting, 11-10
DMA operation, 11-31
edge-sensitive inputs, 11-30
enabling, 11-8
entire field mode, 11-18
FIFO, 11-9
frame start detect, 11-12
frame synchronization, 11-20
frame sync polarity with Timer peripherals, 11-29
Frame Track Error, 11-9
GP modes, frame synchronization, 11-28
PPI (continued)
ITU-R 656 modes, 11-13, 11-20
MMRs, 11-2
number of samples, 11-11
operating modes, 11-3, 11-6
output, 1 sync mode, 11-25
pins, 11-1
port width, 11-5
timer pins, 11-30
vertical blanking interval only mode, 11-19
video data transfer, 11-32
video processing, 11-13
when data transfer begins, 11-8
PPI_CLK signal, 11-3
PPI_CONTROL (PPI Control register), 11-3
PPI Control register (PPI_CONTROL), 11-3
PPI_COUNT (Transfer Count register), 11-11
PPI_DELAY (Delay Count register), 11-9
PPI_FRAME (Lines Per Frame register), 11-12
PPI_FS1 signal, 11-3
PPI_FS2 signal, 11-3
PPI_FS3 signal, 11-9
PPI_STATUS (PPI Status register), 11-8
PPI Status register (PPI_STATUS), 11-8
PRCENx bits, 22-20
Precharge command, 18-27, 18-56
Precharge delay, selecting, 18-42
PREFETCH instruction, 6-37
pre-modify instruction, 5-11
pre-modify stack pointer addressing, 5-11
pre-modify instruction, 5-11
pre-modify stack pointer addressing, 5-11
prescaler, RTC, 17-1
disabling, 17-19
disabling, 17-19
prescaler block, 20-4
private instructions, C-4
probes, oscilloscope, 21-14
Index

processor block diagram, 1-5
processor core architecture, 2-2
processor mode
determination, 3-1
Emulation, 3-9
figure, 3-2
identification, 3-2
IPEND interrogation, 3-1
Supervisor, 3-7
User, 3-3
processor state
Idle, 3-10
Reset, 3-11
product identification registers, 22-25
Program Counter register (PC), 4-3
PC-relative indirect JUMP and CALL, 4-12
PC-relative offset, 4-10, 4-11
program flow, 4-1
programmable flags, 14-1 to 14-22
dedge sensitive, 14-19
level sensitive, 14-19
pins, interrupt, 14-1
polarity, 14-18
system MMRs, 14-5
throughput, 14-22, 15-19
programmable timing characteristics,
EBIU, 18-17
Programmable Warning Limit for REC
and TEC, 19-84
Programming examples
Master Mode Clock Setup, 20-36
Master Mode Receive, 20-38
Master Mode Transmit, 20-37
Receive/Transmit Repeated Start
Sequence, 20-40
Repeated Start Condition, 20-39
Slave Mode, 20-35
programmable model
 cache memory, 6-2
EBIU, 18-7
program sequencer, 4-1
program structures, nonsequential, 4-1
protected instructions, 3-4
protected resources, 3-4
PSM bit, 18-35, 18-54
PSSE bit, 10-9, 18-35
public instructions, C-4, C-6
PULSE_HI bit, 16-18
PULSE_HI toggle mode, 16-21
Pulse Width Count and Capture mode. See
 WDTH_CAP mode
Pulse Width Modulation mode. See
 PWM_OUT mode
PUPSD bit, 18-35
pushing, manual, 4-4
PWM_OUT mode, 16-15 to 16-25
externally clocked, 16-20
PULSE_HI toggle mode, 16-21
stopping the timer, 16-19
Q
quad 16-bit operations, 2-27
query semaphore, 21-6
quotient status, 2-36
R
radix point, D-1, D-2
range
call instruction, 4-11
conditional branches, 4-13
JUMP instruction, 4-10
of signed numbers, D-3
RBSY flag, 10-30
RCKFE bit, 13-22
RCVFLUSH, 20-19
RCVINTLEN, 20-19, 20-25

I-24     ADSP-BF538/ADSP-BF538F Blackfin Processor Hardware Reference
Index

RCVSTAT, 20-21
RCVSTAT field, 20-19
RDIV
  field, 18-48, 18-53, 18-54
  field, equation for value, 18-48
RDTYPE field, 13-20
read, asynchronous, 18-17
read access, for EBIU asynchronous
  memory controller, 18-11
read transfers to SDRAM banks, 18-52
Read/Write command, 18-58
Real-Time Clock. See RTC
Receive Buffer Flush bit, 20-19
Receive Buffer Interrupt Length bit, 20-19
Receive Configuration registers
  (SPORTx_RCR1, SPORTx_RCR2), 13-18
Receive Control Registers, 19-45
receive FIFO, SPORT, 13-25
Receive FIFO Service bit, 20-22, 20-25
Receive FIFO Status bit, 20-21
Receive Logic, 19-38
Receive Message Pending Register
  (CANRMP), 19-45, 19-46, 19-48
receive shift register, 20-3
Receive Status (RCVSTAT) field, 20-28, 20-29
reception error, SPI, 10-30
Rec Receive Mode, 19-11, 19-24
refresh, parallel, 18-33
refresh rate, SDRAM, 21-8
register block, 20-2
register file instructions, 2-8
register files, 2-5 to 2-10
register instructions, conditional branch,
  4-10
register move, 4-14
registers
  accessible in User mode, 3-4
  core, A-1 to A-12
  memory-mapped, core, A-1 to A-12
  product identification, 22-25
  system, B-1 to B-25
Remote Frame Handling Register
  (CANRFH), 19-60
Repeat Start bit, 20-12
replacement policy, 6-34
  definition, 6-72
reserved SDRAM, 18-2
reset
  core double-fault, 3-13
  core-only software, 3-13, 3-17, 3-19
  effect on memory configuration, 6-26
  effect on SPI, 10-3
  hardware, 3-13, 8-14
  initialization sequence
    programming for interrupts, 4-32
  system software, 3-13, 3-15
  watchdog timer, 3-13, 3-15
reset interrupt (RST), 4-44
reset modes, 21-1
resets
  core and system, 3-17
RESET signal, 3-11
Reset state, 3-11
reset vector, 4-45
Reset Vector Addresses (table), 4-45
resources, protected, 3-4
resource sharing, with semaphores, 21-5
Retransmission, 19-50
RETS register, 4-11
return address, 4-3
return address for CALL instruction, 4-9
Return Address registers, 4-4
return from emulation (RTE) instruction,
  4-10
Index

return from exception (RTX) instruction, 4-10
return from interrupt (RTI) instruction, 4-10
return from nonmaskable interrupt (RTN) instruction, 4-10
return from subroutine (RTS) instruction, 4-10
return instruction, 4-10
RETx register, 3-6
RFS pins, 13-39, 13-55, 13-56
RFSR bit, 13-21, 13-39
RND_MOD bit, 2-18, 2-20
ROM, 1-10, 18-1
rounding
  biased, 2-18, 2-20
  convergent, 2-19
  instructions, 2-18, 2-22
  unbiased, 2-18
round-to-nearest, 2-20
ROVF bit, 13-27, 13-29
row address, 18-46
RPOLC bit, 12-14
RRFST bit, 13-22
RSCLKx pins, 13-38
RSFSE bit, 13-22
RSPEN bit, 13-9, 13-18, 13-20
RSTART, 20-12
RST core event, 4-19
RST (reset interrupt), 4-44
RTC, 1-23, 17-1 to 17-21
  alarm clock features, 17-2
  clock requirements, 17-2
  counters, 17-1
  digital watch features, 17-1
  interfaces, 17-2
  interrupt structure, 17-12
  prescaler, 17-1
  programming model, 17-4
RTC
  state transitions, 17-20
  stopwatch function, 17-2
RTC Alarm register (RTC_ALARM), 17-18
RTC_ALARM (RTC Alarm register), 17-18
RTC_ICTL (RTC Interrupt Control register), 17-14
RTC Interrupt Control register (RTC_ICTL), 17-14
RTC Interrupt Status register (RTC_ISTAT), 17-15
RTC_ISTAT (RTC Interrupt Status register), 17-15
RTC_PREN (RTC Prescaler Enable register), 17-18
RTC Prescaler Enable register (RTC_PREN), 17-18
RTC_STAT (RTC Status register), 17-13
RTC Status register (RTC_STAT), 17-13
RTC Stopwatch Count register (RTC_SWCNT), 17-16
RTC_SWCNT (RTC Stopwatch Count register), 17-16
RTE (return from emulation) instruction, 4-10
RTI instruction, use, 4-65
RTI (return from interrupt) instruction, 4-10
RTN (return from nonmaskable interrupt) instruction, 4-10
RTS (return from subroutine) instruction, 4-10
RTX (return from exception) instruction, 4-10
RUVF bit, 13-26, 13-30
RX Hold register, 13-26
RXS bit, 10-31
RXSE bit, 13-22

RTC (continued)
RZI modulation, 12-18

S
SA10 pin, 18-33
SAMPLE/PRELOAD instruction, C-6
sampling clock period, UART, 12-7
sampling edge, SPORT, 13-41
sampling point, UART, 12-7
SB bit, 12-3
scaling, of core timer, 16-48
scan paths, C-5
SCK signal, 10-4, 10-20, 10-23, 10-24
SCLK, 8-4
  changing frequency, 21-10
  derivation, 8-1
  disabling, 8-29
  EBIU, 18-1
  frequency, 8-12
  status by operating mode (table), 8-12
SCLOVR, 20-11
SCLSEN, 20-16
SCOMP, 20-24, 20-26
scratchpad SRAM, 6-5
SCTLE bit, 18-33, 18-37
SDAOVR, 20-12
SDASEN, 20-16
SDC, 18-2, 18-22 to 18-61
  commands, 18-55
  component configurations, 18-30
  configuration, 18-53
  glueless interface features, 18-23
  operation, 18-52
  set up, 18-53
SDEASE bit, 18-47
SDIR, 20-10
SDQM[1:0] Encodings During Writes (table), 18-52
SDQM pins, 18-52

SDRAM
A10 pin, 18-33
address mapping, 18-50
auto-refresh, 18-59
banks, 6-43, 18-28
bank size, 18-1
block diagram, 18-31
Buffering Timing Option (EBUFE), setting, 18-39
configuration, 18-23
external memory, 6-1, 18-50
interface commands, 18-55
latency, 18-35
memory banks, 18-3
no operation command, 18-60
operation parameters, initializing, 18-57
performance, 18-61
power-up sequence, 18-35
read command latency, 18-53
read transfers, 18-52
read/write, 18-58
refresh during PLL transitions, 21-8
refresh rate, 21-8
reserved, 18-2
sharing external, 18-35
size configuration, 18-44
sizes supported, 6-43, 18-23
smaller than 16M byte, 21-8
start addresses, 18-1
timing specifications, 18-60
SDRAM clock enables, setting, 18-37
SDRAM Controller. See SDC
SDRAM Control Status register (EBIU_SDSTAT), 18-47
SDRAM devices supported, 18-45
SDRAM Interface Signals (table), 18-6
SDRAM Memory Bank Control register (EBIU_SDBCTL), 18-44
SDRAM Memory Global Control register (EBIU_SDGCTL), 18-33
SDRAM Refresh Rate Control register (EBIU_SDRRC), 18-47
SDRS bit, 18-54
Self-Refresh command, 18-59
Self-Refresh mode, 18-28
  entering, 18-38
  exiting, 18-38
semaphores, 21-5
  example code, 21-6
  query, 21-6
SEN, 20-8
sensitivity, programmable flags, 14-19
SEQSTAT (Sequencer Status register), 4-4
sequencer, 4-8
sequencer registers, 3-4
Sequencer Status register (SEQSTAT), 4-4
Serial Camera Control Bus, 20-2
serial clock frequency, 10-8
Serial Clock Override bit, 20-11
Serial Clock Sense bit, 20-16
serial communications, 12-2
Serial Data (SDA) Override bit, 20-12
Serial Data Sense bit, 20-16
serial data transfer, 13-1
Serial Peripheral Interface (SPI). See SPI
serial port, 1-18
serial ports. See SPORT
serial scan paths, C-4, C-5
SERR, 20-23, 20-26
servicing interrupt, 4-54
set associative (definition), 6-73
set (definition), 6-72
set up
  for EBIU asynchronous memory controller, 18-11
SDC, 18-53
  SDRAM clock enables, 18-37
shared interrupt, 4-34, 4-60
shifter, 2-1, 2-50 to 2-57
  arithmetic formats, 2-16
  data types, 2-15
  immediate shifts, 2-51, 2-52
  instructions, 2-54
  operations, 2-50
  register shifts, 2-52, 2-53
  status flags, 2-54
  three-operand shifts, 2-52
  two-operand shifts, 2-51
short jump (JUMP.S) instruction, 4-11
SIC, 4-29
SIC_IAR0 (System Interrupt Assignment register 0), 4-34
SIC_IARx (System Interrupt Assignment registers), 4-34
SIC_IMASK (System Interrupt Mask register), 4-32
SIC_IWR (System Interrupt Wakeup Enable register), 4-27
signal integrity, 21-12
signed numbers, 2-3, D-1
  ranges, D-3
sign extending data, 2-11
SIMD video ALU operations, 2-37
single 16-bit operations, 2-26
single pulse generation, timer, 16-17
Single Shot Transmission, 19-50
Single Step exception, 4-50
SINIT, 20-24, 20-26
SIZE bit, 10-9
size of accesses, timer registers, 16-4
SKIP_EN bit, 11-3
SKIP_EO bit, 11-3
Slave Enable bit, 20-8
Slave Overflow bit, 20-23, 20-26
slaves
  EBIU, 18-4
  PAB, 7-6
slave SPI device, 10-6
Slave Transfer Complete bit, 20-24, 20-26
Slave Transfer Complete (SCOMP) bit, 20-26
Slave Transfer Direction bit, 20-10
Slave Transfer Error bit, 20-23, 20-26
Slave Transfer Initiated bit, 20-24, 20-26
Slave Transmit Data Valid bit, 20-8
Sleep mode, 1-27, 8-13
SLEN field, 13-15, 13-21
rerestrictions, 13-36
word length formula, 13-36
software interrupt handlers, 4-19
software management of DMA, 9-39
Software Reset register (SWRST), 3-16
software watchdog timer, 1-24, 16-48
SOVF, 20-23, 20-26
SPE bit, 10-9
speculative load execution, 6-67
speech compression routines, 2-22
SPI, 10-1 to 10-39
and DMA, 10-32 to 10-38
beginning and ending transfers, 10-30
block diagram, 10-2
clock phase, 10-20, 10-22, 10-24
clock polarity, 10-20, 10-24
clock signal, 10-2, 10-24
compatible peripherals, 10-1
data corruption, avoiding, 10-23
data interrupt, 10-6
data transfer, 10-2
detecting transfer complete, 10-15
effect of reset, 10-3
error interrupt, 10-6
error signals, 10-28 to 10-30
general operation, 10-23 to 10-28
interface signals, 10-4 to 10-7
interrupt outputs, 10-6
master mode, 10-2, 10-24
SPI (continued)
master mode booting, 3-20
master mode DMA operation, 10-33
mode fault error, 10-28
multimaster environment, 10-2
multiple slave systems, 10-14
ports, 1-20
reception error, 10-30
registers, table, 10-19
SCK signal, 10-4
slave device, 10-6
slave mode, 10-2, 10-26
slave mode booting, 3-19, 3-20
slave mode DMA operation, 10-36
slave-select function, 10-12
slave transfer preparation, 10-28
SPI_FLG mapping to PFx pins, 10-12
switching between transmit and receive, 10-32
timing, 10-39
transfer formats, 10-20 to 10-22
Transfer Initiate command, 10-25
transfer modes, 10-25
transmission error, 10-30
transmission/reception errors, 10-15
transmit collision error, 10-30
using DMA, 10-17
word length, 10-9
SPI Baud Rate registers (SPI_BAUD), 10-8, 10-20
SPI_BAUD (SPI Baud Rate registers), 10-8, 10-20
SPI_BAUD values, 10-8
SPI Control registers (SPI_CTL), 10-9, 10-19
SPI_CTL (SPI Control registers), 10-9, 10-19
SPIF bit, 10-31
SPI Flag registers (SPI_FLG), 10-11, 10-19
SPI_FLG (SPI Flag registers), 10-11, 10-19
SPI RDBR Shadow registers
(SPI_SHADOW), 10-19, 10-20
SPI_RDBR (SPI Receive Data Buffer registers), 10-18, 10-20
SPI Receive Data Buffer registers
(SPI_RDBR), 10-18, 10-20
SPI_SHADOW (SPI RDBR Shadow registers), 10-19, 10-20
SPISS signal, 10-4, 10-13, 10-14, 10-20
SPI_STAT (SPI Status registers), 10-15, 10-20
SPI Status registers (SPI_STAT), 10-15, 10-20
SPI_TDBR (SPI Transmit Data Buffer registers), 10-17, 10-20
SPI Transmit Data Buffer registers
(SPI_TDBR), 10-17, 10-20
SPORT, 13-1 to 13-72
active low vs. active high frame syncs, 13-41
and DMA block transfers, 13-2
channels, 13-50
clock, 13-38
clock frequency, 13-31, 13-34
clock rate, 13-2
clock rate restrictions, 13-35
clock recovery control, 13-66
companding, 13-37
configuration, 13-10
data formats, 13-36
data word formats, 13-22
disabling, 13-10, 13-38
DMA data packing, 13-65
enable/disable, 13-9
enabling multichannel mode, 13-54
framed serial transfers, 13-40
framed vs. unframed, 13-39
frame sync, 13-40, 13-43
frame sync frequencies, 13-34
SPORT (continued)
frame sync pulses, 13-1
framing signals, 13-39
general operation, 13-8
H.100 standard protocol, 13-66
initialization code, 13-20
internal memory access, 13-46
internal vs. external frame syncs, 13-40
late frame sync, 13-54
modes, 13-10
moving data to memory, 13-46
multichannel frame, 13-57
multichannel operation, 13-50 to 13-66
PAB error, 13-31
packing data, multichannel DMA, 13-65
pens, 13-1, 13-4
point-to-point connections, 21-11
port connection, 13-7
receive and transmit functions, 13-1
receive clock signal, 13-38
receive FIFO, 13-25
receive word length, 13-26
register writes, 13-11
RX Hold register, 13-26
sampling, 13-41
selecting bit order, 13-36
shortened active pulses, 13-10, 13-38
single clock for both receive and
transmit, 13-38
single word transfers, 13-46
stereo serial connection, 13-8
stereo serial frame sync modes, 13-54
support for standard protocols, 13-66
termination, 13-67
timing, 13-67
transmit clock signal, 13-38
transmitter FIFO, 13-23
transmit word length, 13-23
TX Hold register, 13-24
TX interrupt, 13-24
SPORT (continued)
unpack data, multichannel DMA, 13-65
window offset, 13-59
word length, 13-36
SPORT Error interrupt, 13-30
SPORT Multichannel Transmit Select registers (SPORTx_MTCSn), 13-61
SPORT RX interrupt, 13-26, 13-30
SPORTs (serial ports), 1-18
SPORT TX interrupt, 13-30
SPORTx_CHNL (SPORTx Current Channel register), 13-59
SPORTx Current Channel register (SPORTx_CHNL), 13-59
SPORTx_MCMCn (SPORTx Multichannel Configuration registers), 13-53
SPORTx_MRCSn (SPORTx Multichannel Receive Select registers), 13-62
SPORTx_MTCSn (SPORTx Multichannel Transmit Select registers), 13-64
SPORTx Multichannel Configuration registers (SPORTx_MCMCn), 13-53
SPORTx Multichannel Receive Select registers (SPORTx_MRCSn), 13-61, 13-62
SPORTx Multichannel Transmit Select registers (SPORTx_MTCSn), 13-61, 13-64
SPORTx_RCLKDIV (SPORTx Receive Serial Clock Divider registers), 13-31
SPORTx_RCR1 (Receive Configuration registers), 13-18
SPORTx_RCR2 (Receive Configuration registers), 13-18
SPORTx_RCR2 (SPORTx Receive Configuration register), 13-20
SPORTx Receive Configuration 2 registers (SPORTx_RCR2), 13-20
SPORTx Receive Data registers (SPORTx_RX), 13-25
SPORTx Receive Frame Sync Divider registers (SPORTx_RFSDIV), 13-32
SPORTx Receive registers (SPORTx_RX), 13-56
SPORTx Receive Serial Clock Divider registers (SPORTx_RCLKDIV), 13-31
SPORTx_RFSDIV (SPORTx Receive Frame Sync Divider registers), 13-32
SPORTx_RX (SPORTx Receive Data registers), 13-25
SPORTx_STAT (SPORTx Status registers), 13-28
SPORTx Status registers (SPORTx_STAT), 13-28
SPORTx_TCLKDIV (SPORTx Transmit Serial Clock Divider registers), 13-31
SPORTx_TCR1 (Transmit Configuration registers), 13-11
SPORTx_TCR2 (Transmit Configuration registers), 13-11
SPORTx_TFSDIV (SPORTx Transmit Frame Sync Divider registers), 13-32
SPORTx Transmit Data registers (SPORTx_TX), 13-23
SPORTx Transmit Frame Sync Divider registers (SPORTx_TFSDIV), 13-32
SPORTx Transmit registers (SPORTx_TX), 13-45, 13-56
SPORTx Transmit Serial Clock Divider registers (SPORTx_TCLKDIV), 13-31
SPORTx_TX (SPORTx Transmit Data registers), 13-23
Index

SRAM, 18-1
   interface, 21-7
   L1 data, 6-27
   L1 instruction access, 6-8
   scratchpad, 6-5
SRFS bit, 18-36, 18-38
SSEL bit, 7-1
stack, 4-4
Stack Pointer (SP), 4-4
Stack Pointer (SP) registers, 5-5
Stages of Instruction Pipeline (table), 4-7
stalling instructions, 4-8
stalls
   pipeline, 6-64
Start Address registers
   (DMAx_START_ADDR), 9-10
Start Address registers
   (MDMA_yy_START_ADDR), 9-10
Start and Stop conditions, 20-32
state transitions, RTC, 17-20
STATUS field, 12-10
status signals, 2-36
STDVAL, 20-8
stereo serial data, 13-2
stereo serial device, SPORT connection, 13-8
stereo serial frame sync modes, 13-54
STI, 6-70, 6-71, 8-21
sticky overflow status, 2-36
STOP, 20-13
STOPCK field, 8-8
stopwatch function, RTC, 17-2
store operation, 6-63
store ordering, 6-64
strong ordering requirement, 6-70
subroutines, 4-1
Supervisor mode, 3-7
Supervisor Stack Pointer register, 5-5
supply addressing, 5-1
supply addressing with offset, 5-1
SWRST (Software Reset register), 3-16
SYNC, 6-66
synchronization of DMA, 9-39
synchronous serial data transfer, 13-1
SYSCFG (System Configuration register), 4-6
SYSCR (System Reset Configuration register), 3-14
System and Core Event Mapping (table), 4-19
system clock (SCLK), 8-1
System Configuration register (SYSCFG), 4-6
system design, 21-1 to 21-15
   high frequency considerations, 21-11
   point-to-point connections, 21-11
   recommendations and suggestions, 21-12
   recommended reading, 21-14
system interfaces, 7-4
system internal interfaces, 7-1
System Interrupt Assignment register 0
   (SIC_IAR0), 4-34
System Interrupt Assignment registers
   (SIC_IARx), 4-34
System Interrupt Controller (SIC), 4-18
System Interrupt Mask register
   (SIC_IMASK), 4-32
system interrupt processing, 4-21
system interrupts, 4-18
System Interrupt Wakeup Enable register
   (SIC_IWR), 4-27
System Reset Configuration register
   (SYSCR), 3-14
system software reset, 3-13, 3-15
system stack, recommendation for allocating, 4-66
SZ bit, 10-27
tag (definition), 6-73
TAP registers
  Boundary-Scan, C-7
  Bypass, C-6
  Instruction, C-2, C-4
TAP (Test Access port), C-1, C-2
  controller, C-2
T AutorLD bit, 16-45
TBUFCTL (Trace Buffer Control register), 22-16
TBUFSTAT (Trace Buffer Status register), 22-17
TBUF (Trace Buffer register), 22-17
TCKFE bit, 13-17
TCKRE bit, 13-41
TCNTL (Core Timer Control register), 16-45
TCOUNT (Core Timer Count register), 16-46
TDM interfaces, 13-3
TDM Multichannel mode, 13-2
TDTYPE bits, 13-15
  technical support, xlix
Temporary Mailbox Disable Feature
  (CANMBTD), 19-59
TEMT bit, 12-6
  terminations, SPORT pin/line, 13-67
Test Access port (TAP), C-1, C-2
  controller, C-2
Test Clock (TCK), C-6
  test features, C-1 to C-7
testing circuit boards, C-1, C-6
Test-Logic-Reset state, C-3
TESTSET instruction, 6-69, 7-9, 21-5
TFS pins, 13-45
TFSR bit, 13-16, 13-39
TFS signal, 13-56
THRE bit, 12-6
THRE flag, 12-6, 12-15

throughput
  achieved by interlocked pipeline, 6-64
  achieved by SRAM, 6-2
  DAB, 7-10
  programmable flags, 14-22, 15-19
  SPORT, 13-5
TIMDISx bits, 16-5
Time-Division-Multiplexed (TDM) mode, 13-50
  See also SPORT, multichannel operation
TIMENx bits, 16-4
timer
  core, 16-44 to 16-48
  EXT_CLK mode, 16-35 to 16-37
  PWM_OUT mode, 16-15 to 16-25
  watchdog, 16-48 to 16-53
  WDTH_CAP mode, 16-25 to 16-35
Timer Configuration register
  (TIMERx_CONFIG), 16-8
Timer Counter register
  (TIMERx_COUNTER), 16-9
Timer Disable register
  (TIMER_DISABLE), 16-5
TIMER_DISABLE (Timer Disable register), 16-5
Timer Enable register
  (TIMER_ENABLE), 16-4
TIMER_ENABLE (Timer Enable register), 16-4
Timer Period register
  (TIMERx_PERIOD), 16-11
Timer Pulse Width register
  (TIMERx_WIDTH), 16-11
timers, 1-21, 16-1 to 16-53
  general-purpose, 16-1 to 16-42
  UART, 12-2
  watchdog, 1-24
Timer Status register (TIMER_STATUS), 16-6
Index

TIMER_STATUS (Timer Status register), 16-6
TIMERx_CONFIG (Timer Configuration register), 16-8
TIMERx_COUNTER (Timer Counter register), 16-9
TIMERx_PERIOD (Timer Period register), 16-11
TIMERx_WIDTH (Timer Pulse Width register), 16-11
Time Stamp Counter Mode, 19-77
TIMILx bits, 16-7
timing
  Auto-Refresh, 18-47
  external buffer, 18-61
  peripherals, 7-2
  SDRAM specifications, 18-60
  SPI, 10-39
timing examples, for SPORTs, 13-67
TIMOF field, 10-7, 10-9, 10-25
TIN_SEL bit, 16-34
TINT bit, 16-45
TLSBIT bit, 13-15
TMODE field, 16-15
TMPWR bit, 16-46
TMREN bit, 16-45
TMR_EN field, 16-52
TMRx pin, 16-1
tools, development, 1-31
TOVF bit, 13-24, 13-29
TOVF_ERRx bits, 16-8
TPERIOD (Core Timer Period register), 16-47
TPOLC bit, 12-14
trace buffer
  reading, 22-15
Trace Buffer Control register (TBUFCTL), 22-16
Trace Buffer exception, 4-50
Trace Buffer register (TBUF), 22-17
Trace Buffer Status register (TBUFSTAT), 22-17
Trace Unit, 22-14 to 22-18
Transfer Count register (PPI_COUNT), 11-11
Transfer Initiate command, 10-25
transfer initiation from SPI master, 10-25
  operating mode, 8-15, 8-18
  transmission error, SPI, 10-30
  transmission format
    SPORT, 13-2
Transmission Request Set Register (CANTRS), 19-51
Transmit Buffer Flush bit, 20-20
Transmit Buffer Interrupt Length bit, 20-19
Transmit Clock, serial (TSCLKx) pins, 13-38
transmit collision error, SPI, 10-30
Transmit Configuration registers (SPORTx_TCR1 and SPORTx_TCR2), 13-11
Transmit Control Registers, 19-51
Transmit FIFO Service bit, 20-23, 20-25
Transmit FIFO Status bit, 20-21
Transmit Frame Sync (TFS) pins, 13-39
Transmit Logic, 19-49
Transmit Priority defined by Mailbox Number, 19-51
transmit shift register, 20-3
Transmit Status (XMTSTAT) field, 20-28
  tRAS, 18-28
  TRAS field, 18-28, 18-29, 18-30, 18-35, 18-41
  tRC, 18-29
  tRCD, 18-29
  TRCD field, 18-29, 18-35
  tRFC, 18-29
  TRFST bit, 13-17
Index

Trm Transmit Mode, 19-12
\(t_{\text{RP}}\), 18-29
\(\text{TRP} \) field, 18-29, 18-30, 18-35, 18-43
\(t_{\text{RRD}}\), 18-30
t truncation, 2-22
TRUNx bits, 16-6, 16-7
TSCALE (Core Timer Scale register), 16-48
TSFSE bit, 13-17
TSPEN bit, 13-9, 13-11, 13-14, 19-7, 19-8, 19-9
TUVF bit, 13-24, 13-29, 13-46
TWI Bus Arbitration, 20-32
TWI_CLKDIV, 20-6
TWI Clock Generation and Synchronization, 20-31
TWI controller’s serial clock (SCL) output, 20-31
TWI Fast mode, 20-34
TWI FIFO Control register, 20-18
TWI_FIFO_CTL, 20-18
TWI FIFO Receive Data Double Byte register, 20-29
TWI FIFO Receive Data Single Byte register, 20-28
TWI_FIFO_STAT, 20-20, 20-28
TWI FIFO Status register, 20-20
TWI FIFO Transmit Data Double Byte register, 20-27
TWI FIFO Transmit Data Single Byte register, 20-27
TWI General Call Support, 20-33
TWI Interrupt Mask register, 20-22
TWI Interrupt Status register, 20-24
TWI Interrupt Status (TWI_INT_STAT) register, 20-22
TWI_INT_MASK, 20-22
TWI_INT_STAT, 20-24
TWI_MASTER_ADDR, 20-14
TWI_MASTER_CTL, 20-11
TWI Master Mode Address register, 20-14
TWI Master Mode Control register, 20-11
TWI Master Mode Status register, 20-14
TWI_MASTER_STAT, 20-14, 20-25
TWI_RCV_DATA16, 20-29
TWI_RCV_DATA8, 20-28
TWI_SLAVE_ADDR, 20-9
TWI_SLAVE_CTL, 20-7
TWI Slave Mode Address register, 20-9
TWI Slave Mode Control register, 20-7
TWI Slave Mode Status register, 20-9
TWI_SLAVE_STAT, 20-9
TWI transfer protocol, 20-30
TWI_XMT_DATA16, 20-27
TWI_XMT_DATA8, 20-27
two’s complement format, D-1
Two-Wire Interface (TWI) controller, 20-1
\(t_{\text{WR}}\), 18-30
TWR field, 18-30, 18-35, 18-44
TXCOL flag, 10-30
TXE bit, 10-30
TXF bit, 13-24, 13-28
TX Hold register, 13-24
TXS bit, 10-31
TXSE bit, 13-17
\(t_{\text{XSR}}\), 18-30

U

UART, 1-21, 12-1 to 12-20
and system DMA, 12-9
assigning interrupt priority, 12-11
autobaud detection, 16-34
baud rate, 12-6, 12-7
baud rate examples, 12-13
clearing interrupt latches, 12-11
clock rate, 7-2
data word, 12-6
divisor, 12-12
divisor reset, 12-13
UART (continued)
DMA channel latency requirement,
  12-17
DMA channels, 12-16
DMA mode, 12-16
glitch filtering, 12-20
interrupt channels, 12-8
interrupt conditions, 12-10
IrDA receiver, 12-19
IrDA support, 12-18
IrDA transmit pulse, 12-19
IrDA transmitter, 12-18
mixing modes, 12-17
non-DMA mode, 12-15
receive sampling window, 12-20
sampling clock period, 12-7
sampling point, 12-7
standard, 12-1
switching from DMA to non-DMA,
  12-17, 12-18
timers, 12-2
Universal Asynchronous Receiver
  Transmitter ports, 1-21
UART Divisor Latch register
  (UART_DLH), 12-12
  (UART_DLL), 12-12
UART_DLH (UART Divisor Latch
  register), 12-12
UART_DLL (UART Divisor Latch
  register), 12-12
UART_GCTL (UART Global Control
  register), 12-14
UART Global Control register
  (UART_GCTL), 12-14
UART_IER (UART Interrupt Enable
  register), 12-8
UART_IIR (UART Interrupt
  Identification register), 12-10
UART Interrupt Enable register
  (UART_IER), 12-8
UART Interrupt Identification register
  (UART_IIR), 12-10
UART_LCR (UART Line Control
  register), 12-3
UART Line Control register
  (UART_LCR), 12-3
UART Line Status register (UART_LSR),
  12-5
UART_LSR (UART Line Status register),
  12-5
UART_RBR (UART Receive Buffer
  register), 12-7
UART Receive Buffer register
  (UART_RBR), 12-7
UART Scratch register (UART_SCR),
  12-14
UART_SCR (UART Scratch register),
  12-14
UART_THR (UART Transmit Holding
  register), 12-6
UART Transmit Holding register
  (UART_THR), 12-6
UART Transmit Shift register
  (UART_TSR), 12-6
UART_TSR (UART Transmit Shift
  register), 12-6
UCEN bit, 12-13, 12-14
unbiased rounding, 2-18
unconditional branches
  branch latency, 4-15
  branch target address, 4-15
undefined instruction, 4-50
underflow, data, 13-39
UNDR bit, 11-9
unframed/framed, serial data, 13-39
Universal Asynchronous Receiver
  Transmitter (UART) ports, 1-21
Universal Counter Module, 19-77
unpopulated memory, 18-9
Unrecoverable Event, 4-50
unsigned integer, D-1
unsigned numbers, 2-4
data formats, 2-12
use of RTI instruction, 4-65
User mode
accessible registers, 3-3
entering, 3-5
leaving, 3-6
protected instructions, 3-4
User Stack Pointer (USP), 3-7, 5-5
User Stack Pointer (USP) register, 5-5

V
Valid bit
clearing, 6-38
figure, 6-23
function, 6-12
in cache line replacement, 6-15
in instruction cache invalidation, 6-19
valid (definition), 6-73
VCO frequency, changing, 21-8
vertical blanking interval only mode, PPI, 11-19
victim (definition), 6-73
video ALU
instructions, 5-14
operations, 2-37
video data transfers
using PPI, 11-32
VisualDSP++
development environment, 1-31
VLEV field, 8-27
voltage, 8-23
changing, 8-28
control, 8-11
dynamic control, 8-23
Voltage Controlled Oscillator (VCO), 8-3
Voltage Regulator Control register (VR_CTL), 8-25

W
wait states, additional, 18-21
WAKE bit, 8-26
WAKEUP signal, 3-10, 8-20
Watchdog Control register (WDOG_CTL), 16-52
Watchdog Count register (WDOG_CNT), 16-49
Watchdog Status register (WDOG_STAT), 16-50
watchdog timer, 16-48 to 16-53
functionality, 1-24
operation, 16-49
watchdog timer reset, 3-13, 3-15
Watchpoint Match, 4-50
Watchpoint Status register (WPSTAT), 22-12
Watchpoint Unit, 22-1 to 22-12
combination of instruction and data watchpoints, 22-3
data address watchpoints, 22-10
instruction watchpoints, 22-4
memory-mapped registers, 22-2
WPIACTL watchpoint ranges, 22-4
waveform generation, pulse width modulation, 16-17
Way
1-Way associative (direct-mapped), 6-71
definition, 6-73
priority in cache line replacement, 6-16
WB (Write Back), 4-7
WDOG_CNT (Watchdog Count register), 16-49
WDOG_CTL (Watchdog Control register), 16-52
WDOG_STAT (Watchdog Status register), 16-50
Index

WDSIZE[1:0] field, 9-14
WDTH_CAP mode, 16-25 to 16-35
WNR bit, 9-14
WOFF field, 13-59
WOM bit, 10-23
word (definition), 2-6
word length
SPI, 10-9
SPORT, 13-36
SPORT receive data, 13-26
SPORT transmission, 13-2
SPORT transmit data, 13-23
WPDACNTn (Data Watchpoint Address Count Value registers), 22-11
WPDACTL (Data Watchpoint Address Control register), 22-12
WPDA (Data Watchpoint Address registers), 22-11
WPIACNTn (Instruction Watchpoint Address Count registers), 22-6
WPIACTL (Instruction Watchpoint Address Control register), 22-7
WPIA (Instruction Watchpoint Address registers), 22-5
WPPWR bit, 22-7
WPSTAT (Watchpoint Status register), 22-12
wraparound buffer, 5-9
write, asynchronous, 18-19
write access for EBIU asynchronous memory controller, 18-11
write back (definition), 6-73
Write Back (WB), 4-7
write buffer depth, 6-36
write through (definition), 6-73
write to precharge delay, selecting, 18-43
WSIZE field, 13-58

X

XFR_TYPE field, 11-6
XMTFLUSH, 20-20
XMTINTLEN, 20-19, 20-25
XMTSTAT, 20-21
XMTSTAT field, 20-19
XOR, logical, 2-25

Y

YCbCr format, 11-3

Z

zero extending data, 2-11
zero-overhead loop registers, 4-5
zero status, 2-36
µ-law companding, 13-2, 13-62