

ADSP-BF54x Blackfin® Processor Hardware Reference (Volume 1 of 2) Preliminary

Revision 0.4, August 2008

Part Number
82-000000-02

Analog Devices, Inc.
One Technology Way
Norwood, Mass. 02062-9106



Copyright Information

© 2008 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.

Lockbox Secure Technology Disclaimer

Analog Devices products containing LockboxTM Secure Technology are warranted by Analog Devices as detailed in the Analog Devices Standard Terms and Conditions of Sale. To our knowledge, the Lockbox Secure Technology, when used in accordance with the data sheet and hardware reference manual specifications, provides a secure method of implementing code and data safeguards. However, Analog Devices does not guarantee that this technology provides absolute security. ACCORDINGLY, ANALOG DEVICES HEREBY DISCLAIMS ANY AND ALL EXPRESS AND IMPLIED WARRANTIES THAT THE LOCKBOX SECURE TECHNOLOGY CANNOT BE BREACHED, COMPROMISED OR OTHERWISE CIRCUMVENTED AND IN NO EVENT SHALL ANALOG DEVICES BE LIABLE FOR ANY LOSS, DAMAGE, DESTRUCTION OR RELEASE OF DATA, INFORMATION, PHYSICAL PROPERTY OR INTELLECTUAL PROPERTY.

Trademark and Service Mark Notice

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

Lockbox is a trademark of Analog Devices, Inc.

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

CONTENTS

PREFACE

Contents of Two Volumes	xliii
Purpose of This Manual	xliv
Intended Audience	xliv
Manual Contents	xl v
What's New in This Manual	xl viii
Technical or Customer Support	xl viii
Supported Processors	xl ix
Conventions	1
Register Diagram Conventions	li

INTRODUCTION

Peripherals	1-2
Memory Architecture	1-5
Internal Memory	1-6
External Memory	1-7
NAND Flash Controller (NFC)	1-8

I/O Memory Space	1-9
One-Time-Programmable (OTP) Memory	1-9
DMA Support	1-10
Host DMA Interface	1-12
External Bus Interface Unit	1-13
DDR SDRAM Controller	1-13
Asynchronous Controller	1-14
Ports	1-14
General-Purpose I/O (GPIO)	1-14
Two-Wire Interface	1-15
Controller Area Network	1-16
Enhanced Parallel Peripheral Interface (EPPI)	1-17
SPORT Controllers	1-19
Serial Peripheral Interface (SPI) Port	1-21
Timers	1-21
UART Ports	1-23
USB On-The-Go, Dual-Role Device Controller	1-24
ATA/ATAPI-6 Interface	1-24
Keypad Interface	1-25
Secure Digital (SD)/SDIO Controller	1-26
Rotary Counter Interface	1-26
Security	1-27
Media Transceiver Mac Layer (MXVR)	1-28
Real-Time Clock	1-29
Watchdog Timer	1-30

Contents

Clock Signals	1-31
Dynamic Power Management	1-32
Full On Mode (Maximum Performance)	1-32
Active Mode (Moderate Dynamic Power Savings)	1-32
Sleep Mode (High Dynamic Power Savings)	1-32
Deep Sleep Mode (Maximum Dynamic Power Savings)	1-33
Hibernate State (Maximum Power Savings)	1-33
Voltage Regulation	1-34
Boot Modes	1-34
Instruction Set Description	1-35
Development Tools	1-36

CHIP BUS HIERARCHY

Overview	2-1
Internal Interfaces	2-1
Internal Clocks	2-5
Core Bus Overview	2-6
System Overview	2-8
P Port Interface	2-9
D Port Interface	2-10
On-Chip L2 Interface	2-11
Peripheral Access Bus (PAB)	2-15
PAB Performance	2-15
PAB Agents (Masters, Slaves)	2-16
DMA-Related Buses	2-17

Peripheral DMA	2-18
DAB Bus Agents (Masters)	2-18
DAB Arbitration	2-19
DCB Arbitration	2-21
DEB Arbitration	2-23
DAB, DCB, and DEB Performance	2-23
External Access Bus (EAB)	2-25
EAB/DEB Arbitration	2-26
EAB/DEB Performance	2-26

MEMORY

Memory Architecture	3-2
Internal Memory	3-5
Overview of L1 Instruction SRAM	3-6
Overview of L1 Instruction ROM	3-6
Overview of L1 Data SRAM	3-7
Overview of Scratchpad Data SRAM	3-7
Overview of On-Chip L2	3-8
L1 Instruction Memory	3-8
Instruction Memory Control Register (IMEM_CONTROL)	3-9
L1 Instruction SRAM	3-12
L1 Instruction Cache	3-15
Cache Lines	3-15
Cache Hits and Misses	3-18

Contents

Cache-Line Fills	3-18
Line-Fill Buffer	3-19
Cache-Line Replacement	3-19
Instruction Cache Management	3-21
Instruction Cache Locking by Line	3-21
Instruction Cache Locking by Way	3-22
Instruction Cache Invalidation	3-23
Instruction Test Registers	3-24
ITEST_COMMAND Register	3-25
ITEST_DATA1 Register	3-26
ITEST_DATA0 Register	3-27
L1 Data Memory	3-28
Data Memory Control Register (DMEM_CONTROL)	3-28
L1 Data SRAM	3-31
L1 Data Cache	3-32
Example of Mapping Cacheable Address Space into Data Banks	3-35
Data Cache Access	3-39
Cache Write Method	3-41
Write Buffers	3-41
Interrupt Priority Register (IPRIO), Write Buffer Depth ...	3-42
Data Cache Control Instructions	3-43
Data Cache Invalidation	3-44
Data Test Registers	3-44
Data Test Command Register (DTEST_COMMAND)	3-45

Data Test Data 1 Register (DTEST_DATA1)	3-47
Data Test Data 0 Register (DTEST_DATA0)	3-47
On-Chip Level 2 (L2) Memory	3-49
On-Chip L2 Bank Access	3-49
Latency	3-50
One Time Programmable Memory	3-53
External Memory	3-53
Memory Protection and Properties	3-54
Memory Management Unit	3-54
Memory Pages	3-55
Memory Page Attributes	3-56
Page Descriptor Table	3-57
CPLB Management	3-58
MMU Application	3-60
Examples of Protected Memory Regions	3-62
ICPLB Data Registers (ICPLB_DATAx)	3-63
DCPLB Data Registers (DCPLB_DATAx)	3-65
DCPLB Address Registers (DCPLB_ADDRx)	3-67
ICPLB Address Registers (ICPLB_ADDRx)	3-69
CPLB Status Registers	3-70
DCPLB Status Register (DCPLB_STATUS)	3-71
ICPLB Status Register (ICPLB_STATUS)	3-72
CPLB Fault Address Registers	3-72
DCPLB Fault Address Register (DCPLB_FAULT_ADDR)	3-73

Contents

ICPLB Fault Address Register (ICPLB_FAULT_ADDR) ...	3-74
Memory Transaction Model	3-74
Load/Store Operation	3-75
Interlocked Pipeline	3-76
Ordering of Loads and Stores	3-77
Synchronizing Instructions	3-78
Speculative Load Execution	3-79
Conditional Load Behavior	3-80
Working With Memory	3-81
Alignment	3-81
Cache Coherency	3-81
Atomic Operations	3-82
Memory-Mapped Registers	3-83
Core MMR Programming Code Example	3-83
Terminology	3-84

SYSTEM INTERRUPTS

Overview	4-1
Features	4-2
Interfaces	4-2
Description of Operation	4-6
Events and Sequencing	4-6
System Peripheral Interrupts	4-10
Programming Model	4-22
System Interrupt Initialization	4-22

System Interrupt Processing Summary	4-22
System Interrupt Controller Registers	4-26
System Interrupt Assignment (SIC_IARx) Registers	4-27
System Interrupt Mask (SIC_IMASKx) Registers	4-34
System Interrupt Status (SIC_ISRx) Registers	4-38
System Interrupt Wakeup (SIC_IWRx) Registers	4-40
Programming Examples	4-44
Clearing Interrupt Requests	4-44

DIRECT MEMORY ACCESS

Overview and Features	5-2
DMA Controller Overview	5-6
External Interfaces	5-8
Internal Interfaces	5-8
Peripheral DMA	5-10
Memory DMA	5-13
Handshaked Memory DMA Mode	5-16
Modes of Operation	5-17
Register-Based DMA Operation	5-17
Stop Mode	5-18
Autobuffer Mode	5-19
Two-Dimensional DMA Operation	5-19
Examples of Two-Dimensional DMA	5-20
Descriptor-Based DMA Operation	5-21
Descriptor List Mode	5-22

Contents

Descriptor Array Mode	5-23
Variable Descriptor Size	5-23
Mixing Flow Modes	5-25
Functional Description	5-25
DMA Operation Flow	5-25
DMA Startup	5-25
DMA Refresh	5-31
Work Unit Transitions	5-33
DMA Transmit and MDMA Source	5-34
DMA Receive	5-36
Stopping DMA Transfers	5-37
DMA Errors (Aborts)	5-38
DMA Control Commands	5-40
Restrictions	5-44
Transmit Restart or Finish	5-45
Receive Restart or Finish	5-45
Handshaked Memory DMA Operation	5-46
Pipelining DMA Requests	5-48
HMDMA Interrupts	5-50
DMA Performance	5-51
DMA Throughput	5-52
Memory DMA Timing Details	5-55
Static Channel Prioritization	5-55
Temporary DMA Urgency	5-55

Memory DMA Priority and Scheduling	5-57
Traffic Control	5-59
Programming Model	5-61
Synchronization of Software and DMA	5-62
Single-Buffer DMA Transfers	5-64
Continuous Transfers Using Autobuffering	5-65
Descriptor Structures	5-67
Descriptor Queue Management	5-68
Descriptor Queue Using Interrupts on Every Descriptor	5-69
Descriptor Queue Using Minimal Interrupts	5-70
Software-Triggered Descriptor Fetches	5-72
DMA Registers	5-74
DMA Channel Registers	5-74
Peripheral Map (DMAx_PERIPHERAL_MAP and MDMA_yy_PERIPHERAL_MAP) Registers	5-79
DMA Configuration Registers	5-82
Interrupt Status Registers	5-87
Start Address Registers	5-91
Current Address Registers	5-93
Inner Loop Count Registers	5-96
Current Inner Loop Count Registers	5-98
Inner Loop Address Increment Registers	5-101
Outer Loop Count Registers	5-103
Current Outer Loop Count egisters	5-105
Outer Loop Address Increment Registers	5-108

Contents

Next Descriptor Pointer Registers	5-110
Current Descriptor Pointer Registers	5-113
Handshake MDMA (HMDMA) Registers	5-116
Handshake MDMA Control Registers	5-117
Handshake MDMA Initial Block Count Registers	5-120
Handshake MDMA Current Block Count Registers	5-120
Handshake MDMA Current Edge Count Registers	5-121
Handshake MDMA Initial Edge Count Registers	5-123
Handshake MDMA Edge Count Urgent Registers	5-124
Handshake MDMA Edge Count Overflow Registers	5-125
DMA Traffic Control Registers	5-125
DMA Traffic Control Counter Period Registers	5-126
DMA Traffic Control Counter Registers	5-127
DMA Controller 1 Peripheral Multiplexer Register	5-129
Programming Examples	5-129
Register-Based 2D Memory DMA	5-130
Initializing Descriptors in Memory	5-133
Software-Triggered Descriptor Fetch Example	5-136
Handshake Memory DMA Example	5-139

EXTERNAL BUS INTERFACE UNIT

General Overview	6-2
Block Diagram	6-5
On-Chip System Interfaces	6-8
Error Detection	6-8

System Arbitration	6-9
Address Resolution	6-10
Reorder Unit	6-10
DDR Queue Manager	6-12
DDR Arbitration	6-12
DDR SDRAM Controller	6-16
Features	6-16
DDR SDRAM Memory Interface	6-17
DDR SDRAM Programming Model	6-18
DDR Registers	6-20
Memory Control Register 0 (EBIU_DDRCTL0)	6-21
Memory Control Register 1 (EBIU_DDRCTL1)	6-22
Memory Control Register 2 (EBIU_DDRCTL2)	6-23
Memory Control Register 3, Regular DDR Devices	6-24
Memory Control Register 3, Mobile DDR Devices	6-25
Queue Configuration Register (EBIU_DDRQUE)	6-26
Error Address Register (EBIU_ERRADD)	6-27
Error Master Register (EBIU_ERRMST)	6-28
Reset Control Register (EBIU_RSTCTL)	6-29
Mode of Operation - DDR	6-29
Data Flow for 16-bit DDR SDRAMs	6-30
Definition of Standard DDR Terms	6-31
DDR SDRAM System Organization	6-37
DDR SDRAM Configurations Supported	6-39

Contents

DDR Timing Parameter Definitions	6-41
DDR Metrics Control Registers	6-42
DDR Metrics Counter Enable Register	6-42
DDR Metrics Counter Clear Register	6-45
DDR READ Access Count Registers	6-48
DDR WRITE Access Count Registers	6-49
DDR Page ACTIVATE Count Register	6-49
DDR TURN AROUND Count Register	6-50
DDR AUTO-REFRESH Count Register	6-50
DDR Grant Count (EBIU_DDRGCx) Registers	6-50
Asynchronous Memory Interface	6-53
Asynchronous Memory Address Decode	6-53
Asynchronous Memory Arbitration	6-54
ASYNC Interface Control Registers	6-56
Asynchronous Memory Global Control Register	6-57
Asynchronous Memory Bank Control Registers	6-58
Avoiding Bus Contention	6-62
ARDY Input Control	6-62
Memory Bank Select Control Register (EBIU_MBSCTL) ..	6-63
Flash Memory Bank Control Registers	6-64
Bootting From Page Mode or Synchronous Flash	6-64
Access Mode Selection	6-64
Memory Mode Control (EBIU_MODE) Register	6-66
Asynchronous Flash Mode	6-66

Flash Memory Bank Control (EBIU_FCTL) Register	6-67
Asynchronous Page Mode	6-67
Synchronous Burst Mode	6-67
EBIU Arbitration Status Register (EBIU_ARBSTAT)	6-69
Programmable Timing Characteristics	6-70
Asynchronous Accesses by Core Instructions	6-71
Asynchronous Reads	6-71
Asynchronous Writes	6-73
Asynchronous Writes Followed by Reads	6-76
Adding Additional Wait States	6-79
Asynchronous Flash Mode Writes and Reads	6-81
Asynchronous Page Mode Reads	6-82
Synchronous Burst Mode Read	6-83
Bus Request and Grant	6-84

PIXEL COMPOSITOR

Overview	7-2
Features	7-2
Interface Overview	7-3
Description of Operation	7-5
General Description	7-5
Data Buffer Formats	7-7
Operation in YUV 4:2:2 Format	7-7
Operation in RGB888 Format	7-8
DMA Channels	7-9

Contents

Functional Description	7-10
Data Overlay	7-10
Transparency Control	7-16
Transparent Color	7-18
Color Space Conversion	7-19
Case 1 - Image and Overlay in the Same Format	7-20
Case 2 - Image and Overlay in Different Formats	7-21
Case 3 - Color Space Conversion Only	7-22
Color Space Conversion Matrix Equations	7-23
Color Space Converter Output Thresholds	7-25
YUV Conversion Modes	7-25
Upsampling	7-25
Downsampling	7-26
PIXC Actions	7-27
Recommendations	7-28
Special Usage Cases	7-28
Example 1 - Currently Defined Mode	7-29
Example 1 - Special Usage of This Mode	7-29
Example 2 - Currently Defined Mode	7-30
Example 2 - Special Usage of This Mode	7-31
Example 3 - Currently Defined Mode	7-32
Example 3 - Special Usage of This Mode	7-32
Example 4 - Currently Defined Mode	7-33
Example 4 - Special Usage of This Mode	7-33

Programming Model	7-35
PIXC Registers	7-35
PIXC Control (PIXC_CTL) Register	7-38
PIXC Pixels Per Line (PIXC_PPL) Register	7-39
PIXC Lines Per Frame (PIXC_LPF) Register	7-39
PIXC Horizontal Start (PIXC_xHSTART) Registers	7-40
PIXC Horizontal End (PIXC_xHEND) Registers	7-40
PIXC Vertical Start (PIXC_xVSTART) Registers	7-41
PIXC Vertical End (PIXC_xVEND) Registers	7-41
PIXC Transparency Value (PIXC_xTRANSP) Registers	7-42
PIXC Interrupt Status (PIXC_INTRSTAT) Register	7-42
PIXC R/Y Conversion Coefficient (PIXC_RYCON) Register ..	7-43
PIXC G/U Conversion Coefficient (PIXC_GUCON) Register	7-44
PIXC B/V Conversion Coefficient (PIXC_BVCON) Register .	7-45
PIXC Color Conversion Bias (PIXC_CCBIAS) Register	7-46
PIXC Transparency Color Value (PIXC_TC) Register	7-47
Programming Examples	7-47

HOST DMA PORT

Overview	8-1
Features	8-2
Interface Overview	8-3
Description of Operation	8-4
Architecture	8-4
Functional Description	8-5

Contents

HOSTDP Configuration	8-5
HOSTDP Transactions	8-8
Host Read Status	8-8
Host Read Data and Host Write Data Operations	8-9
HOSTDP Modes of Operation	8-10
Acknowledge Mode	8-10
Interrupt Mode	8-14
DMA STOP Mode and AUTOBUFFER Mode	8-16
Bus Widths and Endian Order	8-17
Access Control	8-18
Improving HOSTDP DMA Bus Bandwidth	8-19
Control Commands Between External Host and HOSTDP	8-20
Programming Model	8-22
BF54x Slave	8-22
Host Processor	8-23
Host DMA Port Registers	8-24
Host DMA Port Control (HOST_CONTROL) Register	8-25
Host DMA Port Status (HOST_STATUS) Register	8-27
HOSTDP Timeout (HOST_TIMEOUT) Register	8-29
Programming Examples	8-31

GENERAL-PURPOSE PORTS

Overview	9-1
Features	9-2
Module Overview	9-3

External Interfaces	9-4
Internal Interfaces	9-4
Pin Multiplexing Scheme	9-4
Port A	9-9
Port B	9-10
Port C	9-11
Port D	9-12
Port E	9-13
Port F	9-14
Port G	9-15
Port H	9-16
Port I	9-17
Port J	9-18
Port Multiplexing Control	9-19
GPIO Functionality	9-21
Input Mode	9-21
Output Mode	9-21
Open-Drain Mode	9-22
Pin Interrupts	9-23
Programming Model	9-26
Port Registers	9-30
Port Multiplexing Registers	9-35
Port x Function Enable (PORTx_FER) Registers	9-36
Port Multiplexer Control (PORTx_MUX) Registers	9-36

Contents

GPIO Registers	9-38
Port x GPIO Direction Set Register Pairs	9-39
Port x GPIO Input Enable (PORTx_INEN) Registers	9-40
Port x GPIO Data Register Groups	9-41
Pin Interrupt Registers	9-44
Pin Interrupt Mask Register Pairs	9-46
Interrupt Request and Latch Registers	9-48
Interrupt Edge Register Pairs	9-51
Pin Interrupt Pin State (PINTx_PINSTATE) Register	9-53
Pin Interrupt Invert Set Register Pairs	9-54
Pin Interrupt Assignment (PINTx_ASSIGN) Registers	9-56
Programming Examples	9-60
GENERAL-PURPOSE TIMERS	
Overview and Features	10-1
Features	10-2
Interface Overview	10-3
External Interface	10-5
Internal Interface	10-6
Description of Operation	10-7
Interrupt Processing	10-8
Illegal States	10-11
Modes of Operation	10-14
Pulse Width Modulation (PWM_OUT) Mode	10-14
Output Pad Disable	10-16

Single Pulse Generation	10-16
Pulse-Width Modulation Waveform Generation	10-17
PULSE_HI Toggle Mode	10-19
Externally-Clocked PWM_OUT	10-23
Stopping the Timer in PWM_OUT Mode	10-24
Pulse-Width Count and Capture (WDTH_CAP) Mode	10-27
Autobaud Mode	10-35
Capturing Timings from the GP Counter Module	10-36
External Event (EXT_CLK) Mode	10-36
Programming Model	10-37
Timer Registers	10-39
Timer Enable (TIMER_ENABLEx) Registers	10-40
Timer Disable (TIMER_DISABLEx) Registers	10-41
Timer Status (TIMER_STATUSx) Registers	10-43
Timer Configuration (TIMERx_CONFIG) Registers	10-47
Timer Counter (TIMERx_COUNTER) Registers	10-49
TIMERx_PERIOD and TIMERx_WIDTH Registers	10-52
Summary	10-56
Programming Examples	10-58
 CORE TIMER	
Overview and Features	11-1
Timer Overview	11-2
External Interfaces	11-2
Internal Interfaces	11-2

Contents

Description of Operation	11-3
Interrupt Processing	11-3
Core Timer Registers	11-4
Core Timer Control (TCNTL) Register	11-5
Core Timer Count (TCOUNT) Register	11-6
Core Timer Period (TPERIOD) Register	11-7
Core Timer Scale (TSCALE) Register	11-7
Programming Examples	11-8

WATCHDOG TIMER

Overview and Features	12-1
Interface Overview	12-3
External Interface	12-3
Internal Interface	12-3
Description of Operation	12-4
Watchdog Timer Registers	12-6
Watchdog Count (WDOG_CNT) Register	12-6
Watchdog Status (WDOG_STAT) Register	12-7
Watchdog Control (WDOG_CTL) Register	12-8
Programming Examples	12-10

ROTARY COUNTER

Overview	13-1
Features	13-2
Interface Overview	13-3

Description of Operation	13-4
Quadrature Encoder Mode	13-4
Binary Encoder Mode	13-5
Rotary Counter Mode	13-6
Direction Counter Mode	13-7
Timed Direction Mode	13-7
Functional Description	13-8
Input Noise Filtering (Debouncing)	13-8
Zero Marker (Pushbutton) Operation	13-11
Boundary Comparison Modes	13-14
Rotary Encoder Events: Control and Signaling	13-16
Illegal Gray/Binary Code Events (Two-Step Detection)	13-16
Up/Down Count Events	13-17
Zero Count Events	13-17
Overflow Events	13-17
Boundary Match Events	13-18
Zero Marker Events	13-18
Capturing Timing (Using General-Purpose Timer)	13-18
Capturing Time Interval, Successive Counter Events	13-19
Capturing Counter Interval and Read Timing	13-21
Counter Commands	13-23
Programming Mode	13-24
Rotary Counter Registers	13-24
.....	13-26

Contents

Boundary Register Mode	13-26
Interrupt Mask (CNT_IMASK) Register	13-28
Status (CNT_STATUS) Register	13-29
Command (CNT_COMMAND) Register	13-29
Debounce Prescale (CNT_DEBOUNCE) Register	13-31
Counter (CNT_COUNTER) Register	13-32
Boundary (CNT_MIN and CNT_MAX) Registers	13-32
Programming Examples	13-34

REAL-TIME CLOCK

Overview	14-1
Interface Overview	14-3
Description of Operation	14-5
RTC Clock Requirements	14-5
Prescaler Enable	14-5
RTC Programming Model	14-7
Register Writes	14-8
Write Latency	14-9
Register Reads	14-10
Deep Sleep	14-10
Event Flags	14-11
Setting Time of Day	14-13
Using the Stopwatch	14-14
Interrupts	14-15
State Transitions Summary	14-17

RTC Registers	14-20
RTC Status (RTC_STAT) Register	14-21
RTC Interrupt Control (RTC_ICTL) Register	14-21
RTC Interrupt Status (RTC_ISTAT) Register	14-22
RTC Stopwatch Count (RTC_SWCNT) Register	14-22
RTC Alarm (RTC_ALARM) Register	14-23
RTC Prescaler Enable (RTC_PREN) Register	14-23
Programming Examples	14-24
Enable RTC Prescaler	14-24
RTC Stopwatch For Exiting Deep Sleep Mode	14-25
RTC Alarm to Come Out of Hibernate State	14-27

SECURITY

Overview	15-2
.....	15-5
Description of Operation	15-6
Secure State Machine	15-7
Open Mode	15-8
Secure Entry Mode	15-9
Secure Mode	15-10
SecureMode Control	15-10
Functional Description	15-13
Digital Signature Authentication	15-13
Digital Signature Authentication Performance	15-16
Protection Features	15-17

Contents

Operating in Secure Mode	15-21
Entering Secure Mode	15-21
Exiting Secure Mode	15-21
Reset Handling in Secure Mode	15-21
Hardware Reset	15-21
Clearing Private Data	15-22
Public Key Requirements	15-24
Storing public cipher key in public OTP	15-26
Cryptographic Ciphers	15-26
Keys	15-27
Programming Model	15-27
Secure Entry Service Routine (SESR) API	15-27
Starting Authentication	15-28
Memory Configuration	15-29
Message Placement	15-30
Digital Signature	15-30
Message Size Constraints	15-30
Memory Usage	15-31
Memory Protection	15-31
Secure Function, Secure Entry Service Routine Arguments ...	15-32
Secure Function Arguments	15-32
Secure Entry Service Routine Arguments	15-33
usFlags	15-33
usRQMask	15-35

ulMessageSize	15-35
ulSFEntryPoint	15-35
ulMessagePtr	15-36
Secure Message Execution	15-36
Return Codes	15-36
Advanced Encryption Standard (AES) API	15-39
ADI_AES_DATA Data Type	15-39
ADI_AES_KEYEXPANSION Data Type	15-41
ADI_AES_CIPHER Data Type	15-41
bfrom_AesInit() ROM Routine	15-43
bfrom_AesKeyexp() ROM Routine	15-44
bfrom_AesInvKeyexp() ROM Routine	15-45
bfrom_AesCipher() ROM Routine	15-45
bfrom_AesInvCipher() ROM Routine	15-46
SECURE HASH ALGORITHM (SHA-1) API	15-47
ADI_SHA1 Data Type	15-47
bfrom_Sha1Init ROM Routine	15-48
bfrom_Sha1Hash ROM Routine	15-48
ARC4 API	15-49
ADI_ARC4_KEY Data Type	15-49
ADI_ARC4_DATA Data Type	15-49
bfrom_Arc4Init ROM Routine	15-50
bfrom_Arc4Cipher ROM Routine	15-50
Security Registers	15-51

Contents

Secured System Switches	15-52
SECURE_SYSSWT (0xFFC04320)	15-52
SECURE_SYSSWT (0xFFC04320)	15-53
SECURE_CONTROL (0xFFC04324)	15-62
SECURE_STATUS (0xFFC04328)	15-65

ONE-TIME PROGRAMMABLE MEMORY

OTP Memory Overview	16-1
OTP Memory Map	16-2
Error Correction	16-5
Error Correction Policy	16-6
OTP Access	16-9
OTP Timing Parameters	16-11
OTP_TIMING Register	16-14
Callable ROM Functions for OTP ACCESS	16-14
Initializing OTP	16-14
bfrom_OtpCommand	16-15
Programming and Reading OTP	16-17
bfrom_OtpRead	16-17
bfrom_OtpWrite	16-19
Error Codes	16-23
Write-protecting OTP Memory	16-25
Accessing Private OTP Memory	16-27
OTP Programming Examples	16-28

SYSTEM RESET AND BOOTING

Overview	17-1
Reset and Power-up	17-4
Hardware Reset	17-5
Software Resets	17-6
Reset Vector	17-7
Servicing Reset Interrupts	17-9
Preboot	17-10
Factory Page Settings (FPS)	17-11
Preboot Page Settings (PBS)	17-12
Alternative PBS Pages	17-13
Programming PBS Pages	17-14
Recovering From Misprogrammed PBS Pages	17-14
Customizing Power Management	17-15
Customizing Booting Options	17-18
Customizing the Asynchronous Port	17-19
Customizing the Synchronous Port	17-21
Basic Booting Process	17-22
Block Headers	17-25
Block Code	17-27
Target Address	17-31
Byte Count	17-32
Argument	17-33
Boot Host Wait (HWAIT) Feedback Strobe	17-33

Contents

Using HWAIT as RESETOUT Indicator	17-35
Boot Termination	17-35
Single Block Boot Streams	17-36
Direct Code Execution	17-37
Advanced Boot Techniques	17-39
Initialization Code	17-39
Quick Boot	17-43
Indirect Booting	17-45
Callback Routines	17-46
Error Handler	17-48
CRC Checksum Calculation	17-49
Load Functions	17-49
Calling the Boot Kernel at Run Time	17-50
Debugging the Boot Process	17-51
Boot Management	17-54
Booting a Different Application	17-54
Multi-DXE Boot Streams	17-55
Determining Boot Stream Start Addresses	17-61
Initialization Hook Routine	17-61
Specific Boot Modes	17-62
No Boot Mode	17-63
Flash Boot Modes	17-64
SDRAM Boot Mode	17-69
FIFO Boot Mode	17-69

SPI Master Boot Mode	17-70
SPI Device Detection Routine	17-72
SPI Slave Boot Mode	17-76
TWI Master Boot Mode	17-79
TWI Slave Boot Mode	17-83
UART Slave Mode Boot	17-84
OTP Boot Mode	17-89
Host DMA Boot Modes	17-90
NAND Flash Boot Mode	17-93
Supported Devices	17-94
Auto Detection	17-98
Boot Stream Processing	17-99
Software Configurable NAND Boot Modes	17-101
Sequential Block Mode	17-101
Block Skip Mode	17-102
Multiple Image Mode	17-103
NAND Flash Page Structure	17-105
Reset and Booting Registers	17-107
Software Reset (SWRST) Register	17-107
System Reset Configuration (SYSCR) Register	17-109
Boot Code Revision Control (BK_REVISION)	17-111
Boot Code Date Code (BK_DATECODE)	17-112
Zero Word (BK_ZEROS)	17-113
Ones Word (BK_ONES)	17-114

Contents

OTP Memory Pages for Booting	17-115
Lower PBS00 Half Page	17-115
Upper PBS00 Half Page	17-119
Upper PBS01 Half Page	17-120
Lower PBS02 Half Page	17-122
Upper PBS02 Half Page	17-124
Reserved Half Pages	17-126
Data Structures	17-126
ADI_BOOT_HEADER	17-126
ADI_BOOT_BUFFER	17-127
ADI_BOOT_DATA	17-127
dFlags Word	17-131
ADI_BOOT_NAND	17-133
ADI_BOOT_NAND_DEVICE	17-134
ADI_BOOT_NAND_BUFFER	17-136
ADI_BOOT_NAND_ACCESS	17-138
ADI_BOOT_NAND_ADDRESS	17-139
ADI_BOOT_NAND_ECC	17-141
Callable ROM Functions for Booting	17-144
BFROM_FINALINIT	17-144
BFROM_PDMA	17-144
BFROM_MDMA	17-144
BFROM_MEMBOOT	17-145
BFROM_TWIBOOT	17-146

BFROM_SPIBOOT	17-147
BFROM_OTPBOOT	17-149
BFROM_NANDBOOT	17-150
BFROM_BOOTKERNEL	17-151
BFROM_CRC32	17-151
BFROM_CRC32POLY	17-152
BFROM_CRC32CALLBACK	17-152
BFROM_CRC32INITCODE	17-153
Programming Examples	17-154
System Reset	17-154
Exiting Reset to User Mode	17-155
Exiting Reset to Supervisor Mode	17-155
Initcode (SDRAM Controller Setup)	17-157
Initcode (Power Management Control)	17-158
Initcode (NAND Boot Mode Configuration)	17-160
Quickboot With Restore From SDRAM	17-162
XOR Checksum	17-163
Direct Code Execution	17-164
Managing PBS Pages in OTP Memory	17-166
 DYNAMIC POWER MANAGEMENT	
Phase-Locked Loop and Clock Control	18-1
PLL Overview	18-2
PLL Clock Multiplier Ratios	18-3
Core Clock/System Clock Ratio Control	18-5

Contents

Dynamic Power Management Controller	18-7
Operating Modes	18-8
Dynamic Power Management Controller States	18-8
Full On Mode	18-8
Active Mode	18-9
Sleep Mode	18-9
Deep Sleep Mode	18-10
Hibernate State	18-10
Operating Mode Transitions	18-11
Programming Operating Mode Transitions	18-14
Dynamic Supply Voltage Control	18-16
Power Supply Management	18-16
Controlling the Voltage Regulator	18-18
Changing Voltage	18-20
Powering Down the Core (Hibernate State)	18-22
Recovery From Hibernate State	18-25
PLL and VR Registers	18-26
PLL Divide (PLL_DIV) Register	18-27
PLL Control (PLL_CTL) Register	18-28
PLL Status (PLL_STAT) Register	18-29
PLL Lock Count (PLL_LOCKCNT) Register	18-29
Voltage Regulator Control (VR_CTL) Register	18-30
System Control ROM Function	18-31
Programming Model	18-33

Access System Control ROM Function in C/C++	18-33
Access System Control ROM Function in Assembly	18-34
Programming Examples	18-37
Full On Mode to Active Mode and Back	18-38
Transition to Sleep Mode or Deep Sleep Mode	18-40
Setting Wakeups and Entering Hibernate State	18-42
Perform a System Reset or Soft-Reset	18-44
Change VCO, Core Clock, and System Clock Frequency	18-45
Changing Voltage Levels	18-47

SYSTEM DESIGN

Pin Descriptions	19-1
Managing Clocks	19-2
Managing Core and System Clocks	19-2
Configuring and Servicing Interrupts	19-2
Semaphores	19-3
Example Code for Query Semaphore	19-4
Data Delays, Latencies, and Throughput	19-4
Bus Priorities	19-5
System-Level Hardware Design	19-5
External Memory Design Issues	19-5
DDR Memory	19-5
Memory Bus Pin Muxing and Flow Control	19-6
Example Asynchronous Memory Interfaces	19-7
Avoiding Bus Contention	19-9

Contents

BURST FLASH	19-10
NAND FLASH	19-10
USB Controller	19-12
ATAPI Bus	19-13
Voltage Regulator	19-13
Signal Integrity	19-14
Decoupling Capacitors and Ground Planes	19-15
5 Volt Tolerance	19-17
Resetting the Processor	19-18
Recommendations for Unused Pins	19-18
Programmable Outputs and Pin Multiplexing	19-18
Test Point Access	19-19
Oscilloscope Probes	19-19
Recommended Reading	19-19

GLOSSARY

SYSTEM MMR ASSIGNMENTS

Dynamic Power Management Registers	A-3
System Reset and Interrupt Control Registers	A-4
Watchdog Timer Registers	A-6
Real-Time Clock Registers	A-6
UART0 Controller Registers	A-7
UART1 Controller Registers	A-7
UART2 Controller Registers	A-7

UART3 Controller Registers	A-8
SPI0 Controller Registers	A-8
SPI1 Controller Registers	A-8
TWI Controller Registers	A-8
SPORT0 Controller Registers	A-8
SPORT1 Controller Registers	A-9
SPORT2 Controller Registers	A-9
SPORT3 Controller Registers	A-9
MXVR Registers	A-9
Keypad Registers	A-9
SDH Registers	A-10
ATAPI Registers	A-10
USB_OTG Registers	A-10
External Bus Interface Unit Registers	A-10
DMA/Memory DMA Control Registers	A-12
EPPI0 Registers	A-14
EPPI1 Registers	A-14
Host DMA Registers	A-15
PIXC Registers	A-15
Ports Registers	A-17
Timer Registers	A-26
CAN Registers	A-28
Handshake MDMA Control Registers	A-29
NAND Flash Controller Registers	A-30

Core Timer Registers A-31
Rotary Counter Registers A-31
Security Registers A-32
Processor-Specific Memory Registers A-33

INDEX

INDEX

PREFACE

Thank you for purchasing and developing systems using an enhanced Blackfin[®] processor from Analog Devices.

Contents of Two Volumes

Contents of Volume 1 and Volume 2 are listed below.

Volume 1	Volume 2
Introduction	Introduction
Chip Bus Hierarchy	Media Transceiver Module (MXVR)
Memory	Keypad Interface
System Interrupts	Secure Digital Host
Direct Memory Access	ATAPI Interface
External Bus Interface Unit	NAND Flash Controller
Pixel Compositor	Enhanced Parallel Peripheral Interface
Host DMA Port	CAN Module
General-Purpose Ports	SPI-Compatible Port Controllers
General-Purpose Timers	Two-Wire Interface Controller
Core Timer	SPORT Controllers
Watchdog Timer	UART Port Controllers
Rotary Counter	USB OTG Controller
Real-Time Clock	System MMR Assignments
Security	Test Features
OTP Memory	
System Reset and Booting	
Dynamic Power Management	
System Design	
System MMR Assignments	

Purpose of This Manual

The *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 1 of 2)* provides architectural information about the ADSP-BF542, ADSP-BF544, ADSP-BF547, ADSP-BF548, and ADSP-BF549 processors. The companion volume, *ADSP-BF54x Blackfin Peripheral Processor Hardware Reference (Volume 2 of 2)* provides architectural information about additional peripheral features of these processors. The architectural descriptions cover functional blocks, buses, and ports, including all features and processes that they support. For programming information, see the appropriate *Blackfin Processor Programming Reference*. For timing, electrical, and package specifications, see the *ADSP-BF54x Embedded Processor Data Sheet*.

Intended Audience

The primary audience for this manual is a programmer who is familiar with Analog Devices processors. This manual assumes that 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 the appropriate instruction set reference manuals and data sheets) that describe your target architecture.

Manual Contents

This manual consists of:

- Chapter 1, “[Introduction](#)”
Provides a high level overview of the processor, including peripherals, power management, and development tools.
- Chapter 2, “[Chip Bus Hierarchy](#)”
Describes on-chip buses, including how data moves through the system.
- Chapter 3, “[Memory](#)”
Describes processor-specific memory topics, including L1 memories and processor-specific memory MMRs.
- Chapter 4, “[System Interrupts](#)”
Describes the system peripheral interrupts, including setup and clearing of interrupt requests.
- Chapter 5, “[Direct Memory Access](#)”
Describes the peripheral DMA and memory DMA controllers. Includes performance and software management of DMA, and DMA errors.
- Chapter 6, “[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 7, “[Pixel Compositor](#)”
Describes the overlay manager of the processor. The overlay manager provides data overlay, transparent color, color space conversion support for active (TFT) flat-panel digital color and monochrome LCD displays or analog NTSC and PAL video output.

Manual Contents

- Chapter 8, “[Host DMA Port](#)”
Describes the Host DMA port of the processor. The Host DMA port (HOSTDP) facilitates a host device external to the chip to be a DMA master and transfer data back and forth. The host device always masters the transactions and the DSP is always a DMA slave device.
- Chapter 9, “[General-Purpose Ports](#)”
Describes the general-purpose I/O ports, including the structure of each port, multiplexing, configuring the pins, and generating interrupts.
- Chapter 10, “[General-Purpose Timers](#)”
Describes the general-purpose timer modules that contain identical 32-bit timers.
- Chapter 11, “[Core Timer](#)”
Describes the programmable 32-bit interval timer that can generate periodic interrupts.
- Chapter 12, “[Watchdog Timer](#)”
Describes 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 watchdog expires before being updated by software.
- Chapter 13, “[Rotary Counter](#)”
Describes the rotary (up/down) counter. This counter provides support for manually controlled rotary controllers, such as the volume wheel on a radio device. This unit also supports industrial or motor-control type of wheels.
- Chapter 14, “[Real-Time Clock](#)”
Describes a set of digital watch features of the processor, including time of day, alarm, and stopwatch countdown.

- Chapter 15, “[Security](#)”
Describes the secure digital interface (SDH) of the processor. The secure digital interface provides an interface for multi-media cards (MMC), secure digital memory cards (SD Card), and secure digital input/output Cards (SDIO).
- Chapter 16, “[One-Time Programmable Memory](#)”
Describes an on-chip, one-time-programmable (OTP) memory array which provides 64k-bits of non-volatile memory. This includes the array and logic to support read access and programming.
- Chapter 17, “[System Reset and Booting](#)”
Describes the booting methods, booting process and specific boot modes for the processor.
- Chapter 18, “[Dynamic Power Management](#)”
Describes the clocking (including the PLL) and the dynamic power management controller.
- Chapter 19, “[System Design](#)”
Describes how to use the processor as part of an overall system. The chapter includes information about bus timing and latency numbers, semaphores, and a discussion of the treatment of unused pins.
- Appendix A, “[System MMR Assignments](#)”
Lists the memory-mapped registers included in this manual, their addresses, and cross-references to text.
- “[Glossary](#)”
Contains definitions of terms used in this manual, including acronyms.

What's New in This Manual

This is Revision 0.4 of the *ADSP-BF54x Blackfin Processor Hardware Reference* (Volume 1 of 2). With each revision of this document, modifications and corrections shall be based on errata reports against the manual.

Technical or Customer Support

You can reach Analog Devices, Inc. Customer Support in the following ways:

- Visit the Embedded Processing and DSP products Web site at <http://www.analog.com/processors/technicalSupport>
- E-mail tools questions to processor.tools.support@analog.com
- E-mail processor questions to processor.support@analog.com (World wide support)
processor.europe@analog.com (Europe support)
processor.china@analog.com (China support)
- Phone questions to **1-800-ANALOGD**
- Contact your Analog Devices, Inc. local sales office or authorized distributor
- Send questions by mail to:
Analog Devices, Inc.
One Technology Way
P.O. Box 9106
Norwood, MA 02062-9106
USA

Supported Processors

The following is the list of Analog Devices, Inc. processors supported in VisualDSP++®.

Blackfin (ADSP-BFxxx) Processors

The name *Blackfin* refers to a family of 16-bit, embedded processors. VisualDSP++ currently supports the following Blackfin families: ADSP-BF52x, ADSP-BF53x, ADSP-BF54x, and ADSP-BF56x.

TigerSHARC® (ADSP-TSxxx) Processors




The name *TigerSHARC* refers to a family of floating-point and fixed-point (8-bit, 16-bit, and 32-bit) processors. VisualDSP++ currently supports the following TigerSHARC families: ADSP-TS101 and ADSP-TS20x.


SHARC® (ADSP-21xxx) Processors

The name *SHARC* refers to a family of high-performance, 32-bit, floating-point processors that can be used in speech, sound, graphics, and imaging applications. VisualDSP++ currently supports the following SHARC families: ADSP-2106x, ADSP-2116x, ADSP-2126x, ADSP-2136x, and ADSP-2137x.

Conventions


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

Example	Description
SWRST software reset register	Register names appear in UPPERCASE and a special typeface. The descriptive names of registers are in mixed case and regular typeface.
TMR0E, $\overline{\text{RESET}}$	Pin names appear in UPPERCASE and a special typeface. Active low signals appear with an $\overline{\text{OVERBAR}}$.
DRx, I[3:0] $\overline{\text{SMS}}[3:0]$	Register, bit, and pin names in the text may refer to groups of registers or pins: A lowercase x in a register name (DRx) indicates a set of registers (for example, DR2, DR1, and DR0). A colon between numbers within brackets indicates a range of registers or pins (for example, I[3:0] indicates I3, I2, I1, and I0; $\overline{\text{SMS}}[3:0]$ indicates $\overline{\text{SMS}}3$, $\overline{\text{SMS}}2$, $\overline{\text{SMS}}1$, and $\overline{\text{SMS}}0$).
0xabcd, b#1111	A 0x prefix indicates hexadecimal; a b# prefix indicates binary.
	Note: For correct operation, ... 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 ... Caution: Device damage 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 device users. In the online version of this book, the word Warning appears instead of this symbol.

 Additional conventions, which apply only to specific chapters, may appear throughout this document.

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 (see [Table P-1](#)).
 - 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.

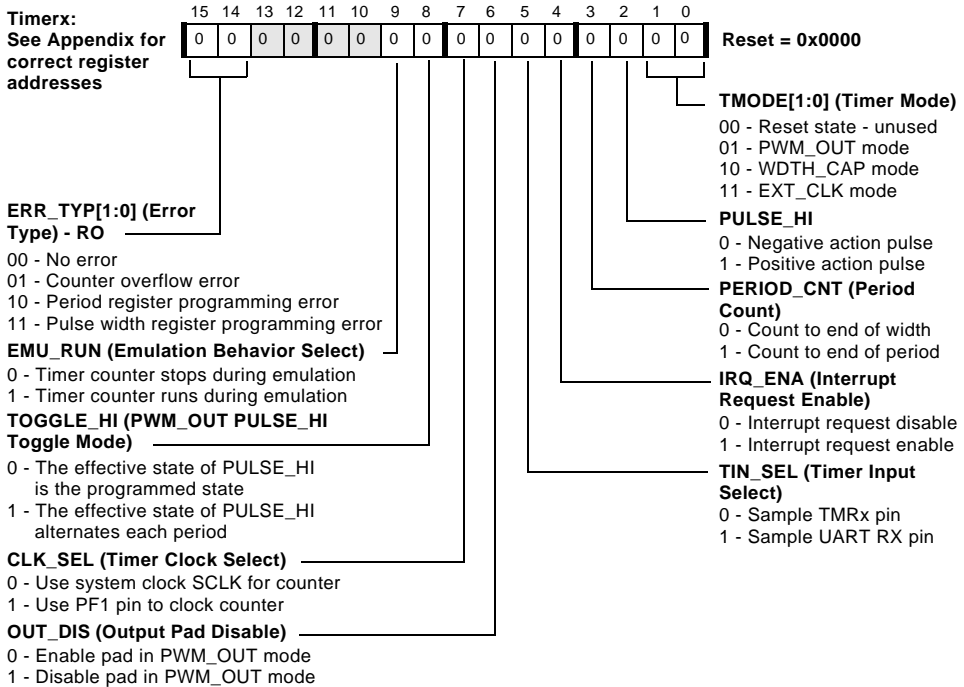
Examples of these conventions are shown in [Figure P-1](#).

Conventions

Table P-1. Short Form of Register Names

Pattern	Description	Examples
TIMER x _CONFIG	The x refers to multiple instances of the peripheral.	TIMER0_CONFIG TIMER1_CONFIG TIMER2_CONFIG
SIC_IAR n	The n refers to multiple registers within the same peripheral or within the same core component.	SIC_IAR2 ICPLB_DATA15
SPORT x _TCR n	The combination of x and n indicates multiple instances of the peripheral <i>and</i> multiple registers within the same peripheral.	SPORT0_TCR0 SPORT1_TCR1
MDMA_ yy _CONFIG	The yy represents MemDMA stream 0 or 1, either destination or source.	MDMA_D0_CONFIG MDMA_S0_CONFIG MDMA_D1_CONFIG MDMA_S1_CONFIG

Timer Configuration Registers (TIMERx_CONFIG)



Core Timer Count Register (TCOUNT)

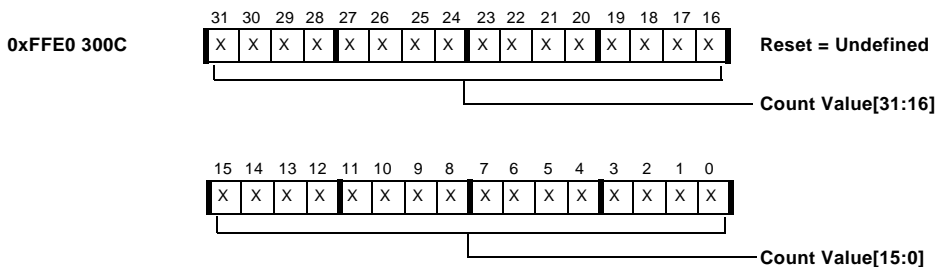


Figure P-1. Register Diagram Examples

Conventions

1 INTRODUCTION

The ADSP-BF54x processors are new members of the Blackfin processor family that offer significant high performance and low power while retaining their ease-of-use benefits. The ADSP-BF54x processors are completely pin compatible, differing only in their performance and on-chip memory, mitigating many risks associated with new product development but allowing the possibility to scale up or down based on specific application demands.

The chapter includes the following sections:

- “Peripherals” on page 1-2
- “Memory Architecture” on page 1-5
- “DMA Support” on page 1-10
- “External Bus Interface Unit” on page 1-13
- “Ports” on page 1-14
- “Two-Wire Interface” on page 1-15
- “Controller Area Network” on page 1-16
- “Enhanced Parallel Peripheral Interface (EPPI)” on page 1-17
- “SPORT Controllers” on page 1-19
- “Serial Peripheral Interface (SPI) Port” on page 1-21
- “Timers” on page 1-21

Peripherals

- “UART Ports” on page 1-23
- “USB On-The-Go, Dual-Role Device Controller” on page 1-24
- “ATA/ATAPI-6 Interface” on page 1-24
- “Keypad Interface” on page 1-25
- “Secure Digital (SD)/SDIO Controller” on page 1-26
- “Rotary Counter Interface” on page 1-26
- “Security” on page 1-27
- “Media Transceiver Mac Layer (MXVR)” on page 1-28
- “Real-Time Clock” on page 1-29
- “Watchdog Timer” on page 1-30
- “Clock Signals” on page 1-31

Peripherals

The processor system peripherals include combinations of:

- High-speed USB on-the-go (OTG) with integrated PHY
- SD/SDIO controller
- ATA/ATAPI-6 controller
- Up to four synchronous serial ports (SPORTs)
- Up to three serial peripheral interfaces (SPI-compatible)
- Up to four UARTs, two with automatic hardware flow control
- Up to two CAN (controller area network) 2.0B interfaces

- Up to two TWI (2-Wire interface) controllers
- 8- or 16-bit asynchronous Host DMA interface
- Multiple enhanced parallel peripheral interfaces (EPPI), supporting ITU-R BT.656 video formats and 18/24-bit LCD connections
- Video data compositor/blender
- Up to eleven 32-bit timers/counters with PWM support
- Real-time clock (RTC) and watchdog timer
- Rotary counter with support for rotary encoder
- Up to 152 general-purpose I/O (GPIOs)
- On-chip PLL capable of 1x to 63x frequency multiplication
- Debug/JTAG interface

These peripherals are connected to the core through several high bandwidth buses, as shown in [Figure 1-1](#).

All of the peripherals, except for general-purpose I/O, CAN, TWI, RTC, and timers, are supported by a flexible DMA structure. There are also two separate memory DMA channels dedicated to data transfers between the processor's memory spaces, which include external DDR1 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.

Peripherals

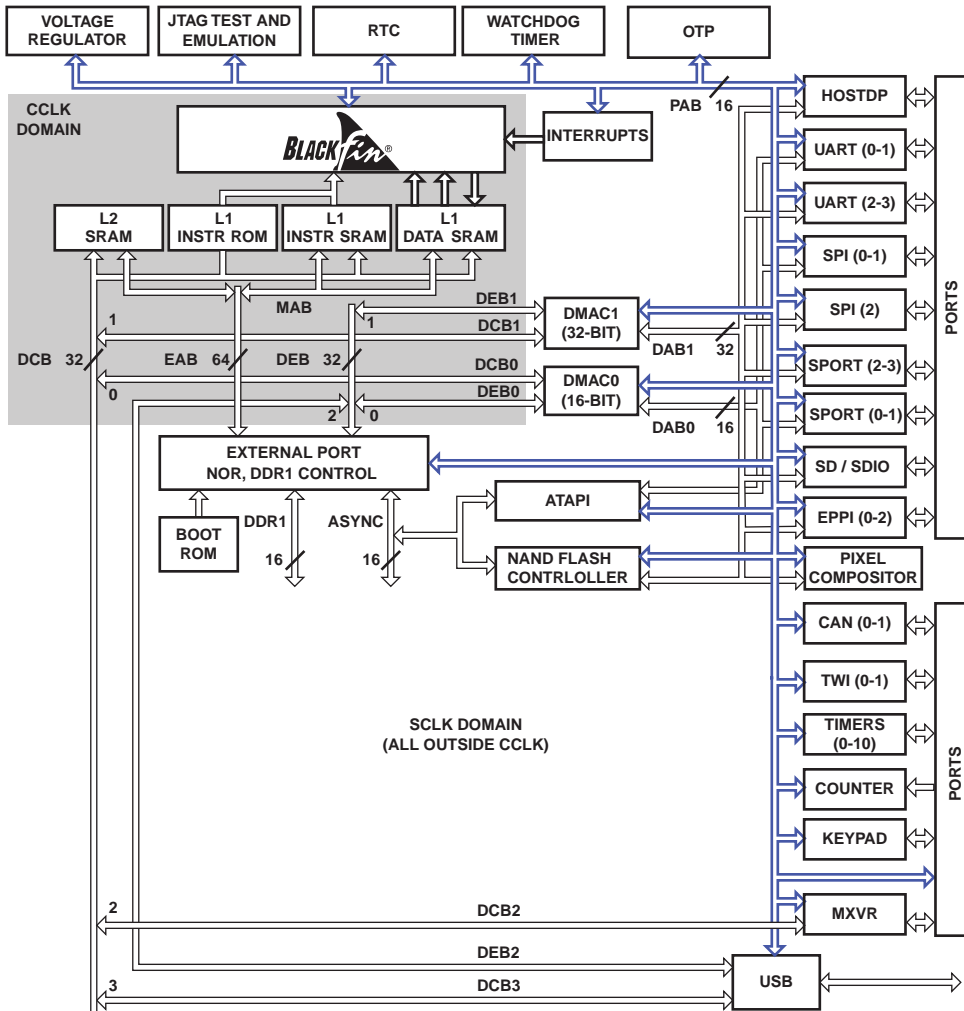


Figure 1-1. ADSP-BF54x Processor Block Diagram

Memory Architecture

The Blackfin processor 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 1-1](#) shows the memory comparison for the ADSP-BF54x processors.

Table 1-1. Memory Configurations

Memory Configurations (K Bytes)	ADSP- BF549	ADSP- BF548	ADSP- BF547	ADSP- BF544	ADSP- BF542
L1 Instruction SRAM/Cache	16	16	16	16	16
L1 Instruction SRAM	48	48	48	48	48
L1 Data SRAM/Cache	32	32	32	32	32
L1 Data SRAM	32	32	32	32	32
L1 Scratchpad SRAM	4	4	4	4	4
L1 ROM ¹	64	64	64	64	64
L2	128	128	128	64	–
L3 Boot ROM ¹	4	4	4	4	4
OTP Memory	8	8	8	8	8

¹ This ROM is not customer configurable.

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 double-data SDRAM (DDR1), flash memory, and SRAM, optionally accessing up to 516M bytes of physical memory.

Memory Architecture

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.

Certain models of the ADSP-BF54x processor also include an L2 SRAM memory array which provides up to 128K bytes of high speed SRAM operating at one half the frequency of the core, and slightly longer latency than the L1 memory banks. The memory other than L1 is a unified instruction and data memory and can hold any mixture of code and data required by the system design.

Internal Memory

The processor has several 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. 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.
- L1 instruction ROM, operating at full processor speed. This ROM is not customer configurable.
- L2 SRAM, providing up to 128K bytes of unified instruction and data memory, operating at one half the frequency of the core.
- 4K boot ROM that can be seen as L3 memory. It operates at full SCLK rate.

External Memory

Through the external bus interface unit (EBIU) the ADSP-BF54x processors provide glueless connectivity to external 16-bit wide memories, such as DDR SDRAM, mobile DDR, SRAM, NOR flash, NAND flash, and FIFO devices. To provide the best performance, the bus system of the DDR interface is completely separate from the other parallel interfaces.

The DDR memory controller can gluelessly manage up to two banks of double-rate synchronous dynamic memory (DDR1 SDRAM). The 16-bit wide interface operates at `SCLK` frequency, enabling maximum throughput of 532 Mbyte/s. The DDR or mobile DDR controller is augmented with a queuing mechanism that performs efficient bursts onto the DDR. The controller is an industry standard DDR SDRAM controller with each bank supporting from 64 Mbit to 512 Mbit device sizes and 4-, 8-, or 16-bit widths. The controller supports up to 512 Mbytes in one bank, but the total in two banks is limited to 512 Mbytes. Each bank is independently programmable and is contiguous with adjacent banks regardless of the sizes of the different banks or their placement.

Traditional 16-bit asynchronous memories, such as SRAM, EPROM, and flash devices, can be connected to one of the four 64 Mbyte asynchronous memory banks, represented by four memory select strobes. Alternatively, these strobes can function as bank-specific read or write strobes preventing further glue logic when connecting to asynchronous FIFO devices.

In addition, the external bus can connect to advanced flash device technologies, such as:

- Page-mode NOR flash devices
- Synchronous burst-mode NOR flash devices
- NAND flash devices

NAND Flash Controller (NFC)

The ADSP-BF54x provides a NAND flash controller (NFC) as part of the external bus interface. NAND flash devices provide high-density, low-cost memory. However, NAND flash devices also have long random access times, invalid blocks, and lower reliability over device lifetimes. Because of this, NAND flash is often used for read-only code storage. In this case, all DSP code can be stored in NAND flash and then transferred to a faster memory (such as DDR or SRAM) before execution. Another common use of NAND flash is for storage of multimedia files or other large data segments. In this case, a software file system may be used to manage reading and writing of the NAND flash device. The file system selects memory segments for storage with the goal of avoiding bad blocks and equally distributing memory accesses across all address locations. Hardware features of the NFC include:

- Support for page program, page read, and block erase of NAND flash devices, with accesses aligned to page boundaries
- Error checking and correction (ECC) hardware that facilitates error detection and correction
- A single 8-bit or 16-bit external bus interface for commands, addresses and data
- Support for SLC (single-level cell) NAND flash devices unlimited in size, with page sizes of 256 and 512 bytes. Larger page sizes can be supported in software
- Capability of releasing external bus interface pins during long accesses
- Support for internal bus requests of 16- or 32-bits
- DMA engine to transfer data between internal memory and NAND flash device

I/O Memory Space

Blackfin processors do not define a separate I/O space. All resources are mapped through the flat 32-bit address space. Control registers for on-chip I/O devices are 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.

One-Time-Programmable (OTP) Memory

The ADSP-BF54x processor also includes an on-chip OTP memory array which provides 64K bits of non-volatile memory that can be programmed by the developer only one time. It includes the array and logic to support read access and programming. A mechanism for error correction is provided. Additionally, its pages can be write protected.

The OTP is not part of the Blackfin linear memory map. OTP memory is not accessed directly using the Blackfin memory map, rather, it is accessed through four 32-bit wide registers (OTP_DATA3-0) which act as the OTP memory read/write buffer.

This memory is organized into 512 pages each comprised of 128 bits and equally separated into two distinct areas with privileged access dependant upon modes of operation when security features are utilized. Approximately 400 pages are available for developer use. The remaining 100 pages are utilized for page protection bits, error correction, and ADI factory reserved areas. One area is read/write accessible at all time (public OTP memory). The second area maintains privileged access and can only be accessed (read/write) upon entry to secure mode when security features are utilized (private OTP memory).

DMA Support

All together, OTP memory provides a means to store public keys in public OTP memory or secrets such as private keys or symmetric keys in private OTP memory. One page of the public OTP memory is initialized in the Analog Devices factory with a unique chip ID.

This OTP memory provides a means to store public and private cipher keys as well as chip, customer, and factory identification data.

DMA Support

ADSP-BF54x processors have multiple, independent DMA channels that support automated data transfers with minimal overhead for the processor core. DMA transfers can occur between the ADSP-BF54x processor's 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 DDR and asynchronous memory controllers.

While the USB controller and MXVR have their own dedicated DMA controllers, the other on-chip peripherals are managed by two centralized DMA controllers, called DMAC1 (32-bit) and DMAC0 (16-bit). Both operate in the `SCLK` domain. Each DMA controller manages twelve independent DMA channels. The DMAC1 controller masters high bandwidth peripherals over a dedicated 32-bit DMA access bus (DAB32). Similarly, the DMAC0 controller masters most of serial interfaces over the 16-bit DAB16 bus. Individual DMA channels have fixed access priority on the DAB buses. DMA priority of peripherals is managed by flexible peripheral-to-DMA channel assignment.

All four DMA controllers use the same 32-bit DCB bus to exchange data with L1 memory. This includes L1 ROM, but excludes scratchpad memory. Fine granulation of L1 memory and special DMA buffers minimize potential memory conflicts, if the L1 memory is accessed by the core contemporaneously. Similarly, there are dedicated DMA buses between the

DMAC1, DMAC0, and USB DMA controllers and the external bus interface unit (EBIU) that arbitrates DMA accesses to external memories and boot ROM.

The ADSP-BF54x processor DMA controllers support both one-dimensional (1D) and two-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 $\pm 32K$ 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 by the ADSP-BF54x processor DMA controller include:

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

In addition to the dedicated peripheral DMA channels, both the DMAC1 and the DMAC0 controllers feature two memory DMA channel pairs for transfers between the various memories of the ADSP-BF54x processor system. This enables transfers of blocks of data between any of the memories—including external DDR, ROM, SRAM, and flash memory—with minimal processor intervention. Like peripheral DMAs, memory DMA transfers can be controlled by a very flexible descriptor-based methodology or by a standard register-based autobuffer mechanism.

DMA Support

The memory DMA channels of the DMAC1 controller (MDMA2 and MDMA3) can be optionally controlled by the external DMA request input pins. When used in conjunction with the external bus interface unit (EBIU) this so-called handshaked memory DMA (HMDMA) scheme can be used to efficiently exchange data with block-buffered or FIFO-style devices connected externally. Users can select whether the DMA request pins control the source or the destination side of the memory DMA. It allows control of the number of data transfers for memory DMA. The number of transfers per edge is programmable. This feature can be programmed to allow memory DMA to have an increased priority on the external bus relative to the core.

Host DMA Interface

The Host DMA port (HOSTDP) facilitates a host device external to the ADSP-BF54x to be a DMA master and transfer data back and forth. The host device always masters the transactions and the processor is always a DMA slave device.

The HOSTDP port is enabled through the peripheral access bus. Once enabled, the DMA is controlled by the external host. The external host can then program the DMA to send/receive data to any valid internal or external memory location. The HOSTDP port controller includes the following features:

- Allows an external master to configure DMA read/write data transfers and read port status
- Uses an asynchronous memory protocol for its external interface
- Allows 8- or 16-bit external data interface to the host device
- Supports half-duplex operation
- Supports little/big endian data transfers

- Acknowledge mode allows flow control on host transactions
- Interrupt mode guarantees a burst of FIFO depth host transactions

External Bus Interface Unit

Through the external bus interface unit (EBIU) the ADSP-BF54x processors provide glueless connectivity to external 16-bit wide memories, such as DDR SDRAM, SRAM, NOR flash, NAND flash, and FIFO devices. To provide the best performance, the bus system of the DDR interface is completely separate from the other parallel interfaces.

DDR SDRAM Controller

The DDR memory controller can gluelessly manage up to two banks of double-rate synchronous dynamic memory (DDR1 SDRAM). The 16-bit wide interface operates at `SCLK` frequency enabling maximum throughput of 532M byte/s. The DDR controller is augmented with a queuing mechanism that performs efficient bursts onto the DDR. The controller is an industry-standard DDR SDRAM controller.

The maximum size of supported DDR SDRAM is 512M bit (64M byte). Most of these memory devices can be configured as x4, x8 and x16. With x16, one memory chip is configured per “external” bank; with x8 configure two chips; and four chips with x4 configuration. Thus with x4 configuration, $64\text{M byte} \times 4 = 256\text{M byte}$ per external bank can be supported. ADSP-BF54x two external banks provide support for a maximum of $2 \times 256\text{M byte} = 512\text{M byte}$.

Each bank is independently programmable and is contiguous with adjacent banks regardless of the sizes of the different banks or their placement.

Ports

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 64M 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.

Ports

Because of their rich set of peripherals, the ADSP-BF54x processors group the many peripheral signals to ten ports—referred to as Port A to Port J. Most ports contain 16 pins, a few have less. Many of the associated pins are shared by multiple signals. The ports function as multiplexer controls. Every port has its own set of memory-mapped registers to control port multiplexing and GPIO functionality.

General-Purpose I/O (GPIO)

Every pin in Port A to Port J can function as a GPIO pin resulting in a GPIO pin count of 154. While it is unlikely that all GPIOs will be used in an application as all pins have multiple functions, the richness of GPIO functionality guarantees nonrestrictive pin usage. Every pin that is not used by any function can be configured in GPIO mode on an individual basis.

After reset, all pins are in GPIO mode by default. Neither GPIO output nor input drivers are active by default. Unused pins can be left unconnected. GPIO data and direction control registers provide flexible

write-1-to-set and write-1-to-clear mechanisms so that independent software threads do not need to protect against each other because of expensive read-modify-write operations when accessing the same port.

Two-Wire Interface

The ADSP-BF54x processor offers up to two two-wire interface (TWI) interfaces and is fully compatible with the widely used I²C bus standard. It is designed with a high level of functionality and is compatible with multimaster, multislave bus configurations. To preserve processor bandwidth, the TWI controller can be set up and a transfer initiated with interrupts only to service FIFO buffer data reads and writes. Protocol-related interrupts are optional.

The 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 controller includes these features:

- Simultaneous master and slave operation on multiple device systems
- Support for multimaster data arbitration
- 7-bit addressing
- 100K bits/second and 400K bits/second data rates
- General call address support
- 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 lockup

Controller Area Network

- Input filter for spike suppression
- Serial camera control bus support as specified in the *OmniVision Serial Camera Control Bus (SCCB) Functional Specification version 2.1*

Controller Area Network

The ADSP-BF54x processor offers up to two CAN controllers that are communication controllers that implement the controller area network (CAN) 2.0B (active) protocol. This protocol is an asynchronous communications protocol used in both industrial and automotive control systems. The CAN protocol is well suited for control applications due to its capability to communicate reliably over a network since the protocol incorporates CRC checking message error tracking, and fault node confinement.

The ADSP-BF54x CAN controllers offer:

- 32 mailboxes (8 receive only, 8 transmit only, 16 configurable for receive or transmit)
- Dedicated acceptance masks for each mailbox
- Additional data filtering on first two bytes
- Support for both the standard (11-bit) and extended (29-bit) identifier (ID) message formats
- Support for remote frames
- Active or passive network support
- CAN wakeup from hibernation mode (lowest static power consumption mode)
- Interrupts, including: TX complete, RX complete, error, global

The electrical characteristics of each network connection are very demanding so the CAN interface is typically divided into two parts: a controller and a transceiver. This allows a single controller to support different drivers and CAN networks. The ADSP-BF54x CAN module represents only the controller part of the interface. The controller interface supports connection to 3.3V high speed, fault-tolerant, single-wire transceivers.

Enhanced Parallel Peripheral Interface (EPPI)

The ADSP-BF54x processor provides multiple enhanced parallel peripheral interfaces (EPPIs), one 16 bits wide and one 18 bits wide. The EPPI supports the direct connection to active TFT LCD, parallel A/D and D/A converters, video encoders and decoders, image sensor module and other general-purpose peripherals.

The following features are supported in the EPPI module.

- Programmable data length: 8, 10, 12, 14, 16, 18, and 24 bits per clock
- Bidirectional and half-duplex port
- PPI_CLK can be provided externally or can be generated internally
- Various framed and nonframed operating modes. Frame syncs can be generated internally or can be supplied by an external device
- Various general-purpose modes with one frame syncs, two frame syncs, three frame syncs and zero frame sync modes for both receive and transmit
- ITU-656 status word error detection and correction for ITU-656 receive modes
- ITU-656 preamble and status word decode

Enhanced Parallel Peripheral Interface (EPPI)

- Three different modes for ITU-656 receive modes: active video only, vertical blanking only, and entire field mode
- Horizontal and vertical windowing for GP 2 and 3 FS modes
- Optional packing and unpacking of data to/from 32 bits from/to 8, 16 and 24 bits. If packing/unpacking is enabled, endianness can be altered to change the order of packing/unpacking of bytes/words
- Optional sign extension or zero fill for receive modes
- During receive modes, alternate even or odd data sample can be filtered out
- Programmable clipping of data values for 8-bit and 16-bit transmit modes
- RGB888 can be converted to RGB666 or RGB565 for transmit modes
- Various de-interleaving/interleaving modes for receiving/transmitting 4:2:2 YCrCb data
- FIFO watermarks and urgent DMA features
- Clock gating by an external device asserting the clock gating control

SPORT Controllers

The ADSP-BF54x processor incorporates up to four dual-channel synchronous serial ports (SPORT0, SPORT1, SPORT2, SPORT3) for serial and multiprocessor communications. The SPORTs support these features:

- I²S capable operation

Bidirectional operation. Each SPORT has two sets of independent transmit and receive pins, which enable eight channels of I²S stereo audio.

- Buffered (eight-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.

SPORT Controllers

- 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.

Serial Peripheral Interface (SPI) Port

The ADSP-BF54x processor has up to three SPI-compatible ports that enable the processor to communicate with multiple SPI-compatible devices.

Each SPI port 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, and seven SPI chip select output pins let the processor select other SPI devices. The SPI select pins are reconfigured, general-purpose I/O 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's baud rate and clock phase/polarities are programmable. It has an integrated DMA controller, configurable to support either transmit or receive data streams. The SPI's DMA controller can only service unidirectional accesses at any given time.

During transfers, the SPI port simultaneously transmits and receives by serially shifting data in and out of its two serial data lines. The serial clock line synchronizes the shifting and sampling of data on the two serial data lines.

Timers

There are up to two timer units in the ADSP-BF54x processors. Depending on the processor, one unit provides eight general-purpose programmable timers, and the other unit provides three of them. Each timer has 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 and periods of external events.

Timers

These timers can be synchronized to an external clock input (to the several other associated GPIO pins) to an external clock input to the `PPI_CLK` input pin, or to the internal `SCLK`.

The timer units can be used in conjunction with the two UARTs and the CAN controllers to measure the width of the pulses in the data stream to provide a software auto-baud detect function for the respective serial channels.

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

In addition to the general-purpose programmable timers, another timer is also provided by the processor core. 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 ADSP-BF54x processor provides four full-duplex universal asynchronous receiver/transmitter (UART) ports. Each UART port provides a simplified UART interface to other peripherals or hosts, providing half-duplex, DMA-supported, asynchronous transfers of serial data. The UART ports include support for five to eight data bits; one or two stop bits; and none, even, or odd parity. The UART ports support two modes of operation:

- Programmed I/O

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 of the two UARTs have 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 baud rate, serial data format, error code generation and status, and interrupts of the UARTs can be programmed to support:

- 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

USB On-The-Go, Dual-Role Device Controller

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

UART1 and UART3 feature a pair of $\overline{\text{UARTxRTS}}$ (request to send) and $\overline{\text{UARTxCTS}}$ (clear to send) signals for hardware flow purposes. The transmitter hardware is automatically prevented from sending further data when the $\overline{\text{UARTxCTS}}$ input is deasserted. The receiver can automatically deassert its $\overline{\text{UARTxTS}}$ output when the enhanced receive FIFO exceeds a certain high water level.

The capabilities of the UART ports are further extended with support for the Infrared Data Association (IrDA[®]) Serial Infrared Physical Layer Link Specification (SIR) protocol.

USB On-The-Go, Dual-Role Device Controller

The USB OTG controller provides a low-cost connectivity solution for consumer mobile devices such as cell phones, digital still cameras and MP3 players, allowing these devices to transfer data using a point-to-point USB connection without the need for a PC host. The USBDR module can operate in a traditional USB peripheral-only mode as well as the host mode presented in the on-the-go (OTG) supplement to the USB 2.0 specification. In host mode, the USB module supports transfers at high-speed (480 Mbps), full speed (12 Mbps), and low speed (1.5 Mbps) rates. Peripheral-only mode supports the high and full speed transfer rates.

ATA/ATAPI-6 Interface

The ATA/ATAPI interface connects to CD/DVD and HDD drives and is ATAPI-6 compliant. The controller implements the peripheral I/O mode, the multi-DMA mode, and the Ultra DMA mode. The DMA modes

enable faster data transfer and reduced host management. The ATAPI Controller supports PIO, multi-DMA, and Ultra DMA ATAPI accesses. Key features include:

- Supports PIO modes 0, 1, 2, 3, 4
- Supports multiword DMA modes 0, 1, 2
- Supports Ultra DMA modes 0, 1, 2, 3, 4, 5 (up to UDMA 100)
- Programmable timing for ATA interface unit
- Supports CompactFlash card using True IDE mode

Keypad Interface

The keypad interface is a 16-pin interface module that is used to detect the key pressed in a 8-by-8 (maximum) keypad matrix. The size of the input keypad matrix is programmable. The interface is capable of filtering the bounce on the input pins, which is common in keypad applications. The width of the filtered bounce is programmable. The interface module is capable of generating an interrupt request to the core once it identifies that any key is pressed.

The interface supports a press-release-press mode and infrastructure for a press-hold mode. The former mode identifies a press, a release and another press of a key as two consecutive presses of the same key. The later mode checks the input key's state in periodic intervals to determine the number of times the same key is meant to be pressed. Key features include:

- Supports a maximum of 8-by-8 keypad matrix
- Programmable input keypad matrix size
- Debounce filter on input signals
- Programmable debounce filter width

Secure Digital (SD)/SDIO Controller

- Press-release/press mode supported
- Infrastructure for press-hold mode present
- Interrupt on any key pressed capability
- Multiple key pressed detection and limited multiple key resolution capability

Secure Digital (SD)/SDIO Controller

The SD/SDIO controller is a serial interface that stores data at a rate of up to 10M bytes per second using a 4-bit data line. The interface runs at 25 MHz.

The SD/SDIO controller supports the SD memory mode only. The interface supports all the power modes and performs error checking by CRC.

Rotary Counter Interface

A 32-bit rotary counter is provided that can sense 2-bit quadrature or binary codes as typically emitted by industrial drives or manual thumb-wheels. The counter can also operate in general-purpose up/down count modes. Then, count direction is either controlled by a level-sensitive input pin or by two edge detectors.

A third input can provide flexible zero marker support and can alternatively be used to input the push-button signal of thumb wheels. All three pins have a programmable debouncing circuit.

An internal signal forwarded to the timer unit enables one timer to measure the intervals between count events. Boundary registers enable auto-zero operation or simple system warning by interrupts when programmable count values are exceeded.

Security

The ADSP-BF54x Blackfin processor provides security features (Blackfin Lockbox™ Secure Technology) that enable customer applications to use secure protocols consisting of code authentication and execution of code within a secure environment. Implementing secure protocols on Blackfin processors involve a combination of hardware and software components. Together these components protect secure memory spaces and restrict control of security features to authenticated developer code.

- Blackfin Lockbox Secure Technology incorporates a secure hardware platform for confidentiality and integrity protection of secure code and data with authenticity maintained by secure software.
- This secure platform provides:
 - A secure execution mode
 - Secure storage for on-chip keys
 - On-chip secure ROM
 - Secure RAM
- Access to code and data in the secure domain is monitored by the hardware and any unauthorized access to the secure domain is prevented.
- The secure ROM code establishes the root of trust for the secure software in the system.
- The secure RAM provides integrity protection and confidentiality for authenticated code and data.

Media Transceiver Mac Layer (MXVR)

- User-defined cipher key(s) and ID(s) and can be securely stored in the on-chip OTP memory.
- Every processor ships from the ADI factory with a unique chip ID value stored in publicly accessible OTP memory area.

Media Transceiver Mac Layer (MXVR)

The ADSP-BF54x processor provides a media transceiver (MXVR) MAC layer, allowing the processor to be connected directly to a MOST®¹ network through just an FOT or electrical PHY.

The MXVR is fully compatible with the industry-standard standalone MOST controller devices, supporting 22.579 Mbps or 24.576 Mbps data transfer. It offers faster lock times, greater jitter immunity, a sophisticated DMA scheme for data transfers. The high-speed internal interface to the core and L1 memory allows the full bandwidth of the network to be utilized. The MXVR can operate as either the network master or as a network slave.

The MXVR supports synchronous data, asynchronous packets, and control messages using dedicated DMA channels which operate autonomously from the processor core moving data to and from L1 memory. Synchronous data is transferred to or from the synchronous data physical channels on the MOST bus through eight programmable DMA channels. The synchronous data DMA channels can operate in various modes including modes which trigger DMA operation when data patterns are detected in the receive data stream. Furthermore two DMA channels support asynchronous traffic and another two support control message traffic.

¹ MOST is a registered trademark of Standard Microsystems, Corp.

Interrupts are generated when a user-defined amount of synchronous data is sent or received by the processor or when asynchronous packets or control messages have been sent or received.

The MXVR peripheral can wake up the ADSP-BF54x processor from sleep mode when a wakeup preamble is received over the network or based on any other MXVR interrupt event. Additionally, detection of network activity by the MXVR can be used to wake up the ADSP-BF54x processor from sleep mode or the hibernate state, and wake up the on-chip internal voltage regulator from a powered-down state. These features allow the ADSP-BF54x to operate in a low-power state when there is no network activity or when data is not currently being received or transmitted by the MXVR.

The MXVR clock is provided through a dedicated external crystal or crystal oscillator. The frequency of the external crystal or the crystal oscillator can be 256Fs, 384Fs, 512Fs, or 1024Fs for Fs = 38 kHz, 44.1 kHz, or 48 kHz. If using a crystal to provide the MXVR clock, use a parallel-resonant, fundamental mode, microprocessor-grade crystal.

Real-Time Clock

The processor's 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.

Watchdog Timer

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 second resolution. When the stopwatch is enabled and the counter underflows, an interrupt is generated.

Like the other peripherals, the RTC can wake up the processor from sleep mode or deep sleep mode upon generation of any RTC wakeup event. An RTC wakeup event can also wake up the on-chip internal voltage regulator from a powered-down state.

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, nonmaskable 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 core and the ADSP-BF54x processor 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} .

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 processor's CLKIN pin. The 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 phase-locked loop (PLL) is capable of multiplying the CLKIN signal by a user-programmable (0.5x to 64x) multiplication factor (bounded by specified minimum and maximum VCO frequencies). The default multiplier is 8x, 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.

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 Mode (Maximum Performance)

In the full on mode, the 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 Mode (Moderate Dynamic Power Savings)

In the 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 and L2 memories.

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 Mode (High Dynamic Power Savings)

The sleep mode reduces dynamic 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 enabled in the SIC_IWRx registers 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 and memory other than L1 is not supported.

Deep Sleep Mode (Maximum Dynamic Power Savings)

The deep sleep mode maximizes dynamic power savings by disabling the processor core and synchronous system clocks (CCLK and SCLK). Asynchronous systems, such as the RTC, may still be running, but cannot 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, an RTC asynchronous interrupt causes the processor to transition to the active mode. Assertion of $\overline{\text{RESET}}$ while in deep sleep mode causes the processor to transition to the full on mode.

Hibernate State (Maximum Power Savings)

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.

Voltage Regulation

The processor provides an on-chip voltage regulator that can generate internal voltage levels. The voltage regulation circuit figure in the data sheet shows the typical external components required to complete the power management system. The regulator controls the internal logic voltage levels and is programmable with the voltage regulator control register (VR_CTL) in increments of 50 mV. To reduce standby power consumption, the internal voltage regulator can be programmed to remove power to the processor core while keeping I/O power supplied. While in this state, V_{DDEXT} can still be applied, eliminating the need for external buffers. The regulator can also be disabled and bypassed at the user's discretion. For more information, see the Voltage Regulator Circuit diagram in the ADSP-BF54x data sheet.

Boot Modes

The ADSP-BF54x processor has many mechanisms (listed in [Table 1-2](#)) for automatically loading internal and external memory after a reset. The boot mode is defined by four $BMODE$ input pins dedicated to this purpose. There are two categories of boot modes, master and slave. In master boot mode, the processor actively loads data from parallel or serial memories. In slave boot mode, the processor receives data from an external host device.

Table 1-2. Booting Modes

$BMODE$ [3: 0]	Description
b#0000	Idle—no boot
b#0001	Boot from 8- or 16-bit external flash memory
b#0010	Boot from 16-bit asynchronous FIFO
b#0011	Boot from serial SPI memory (EEPROM or flash)

Table 1-2. Booting Modes

BMODE [3: 0]	Description
b#0100	Boot from SPI host device
b#0101	Boot from serial TWI memory (EEPROM/flash)
b#0110	Boot from TWI host
b#0111	Boot from UART host
b#1000	Reserved
b#1001	Reserved
b#1010	Boot from (DDR) SDRAM
b#1011	Boot from OTP memory
b#1100	Reserved
b#1101	Boot from 8- or 16-bit NAND flash memory via NFC
b#1110	Boot from 16-Bit Host DMA
b#1111	Boot from 8-Bit Host DMA

Instruction Set Description

The ADSP-BF54x processor family assembly language instruction set employs an algebraic syntax designed for ease of coding and readability. Refer to the appropriate *Blackfin Processor Programming Reference* for detailed information. 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 micro controllers, this instruction set is very efficient when compiling C and C++ source code. In addition, the archi-

Development Tools

ecture 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 processor's unique architecture, offers these advantages:

- Embedded 16/32-bit 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
- Seamlessly integrated DSP/CPU features 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-mapped into a unified 4G byte memory space, providing a simplified programming model

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 with a complete set of CrossCore[®] software and hardware development tools, including Analog Devices emulators and the VisualDSP++ development environment. The same emulator hardware that supports other Analog Devices products also fully emulates the ADSP-BF54x processor family.

The VisualDSP++ project management environment lets programmers develop and debug an application. This environment includes an easy-to-use assembler that is based on an algebraic syntax, an archiver (librarian/library builder), a linker, a loader, a cycle-accurate, instruction-level simulator, a C/C++ compiler, and a C/C++ runtime library that includes DSP and mathematical functions. A key point for these tools is C/C++ code efficiency. The compiler is developed for efficient translation of C/C++ code to Blackfin processor assembly. The Blackfin processor has architectural features that improve the efficiency of compiled C/C++ code.

Debugging both C/C++ and assembly programs with the VisualDSP++ debugger, programmers can:

- View mixed C/C++ and assembly code (interleaved source and object information)
- Insert breakpoints
- Set conditional breakpoints on registers, memory, and stacks
- Trace instruction execution
- Perform linear or statistical profiling of program execution
- Fill, dump, and graphically plot the contents of memory
- Perform source-level debugging
- Create custom debugger windows

Development Tools

The VisualDSP++ Integrated Development Environment (IDE) lets programmers define and manage software development. Its dialog boxes and property pages let programmers configure and manage all development tools, including color syntax highlighting in the VisualDSP++ editor. These capabilities permit programmers to:

- Control how the development tools process inputs and generate outputs
- Maintain a one-to-one correspondence with the tool's command-line switches

The VisualDSP++ Kernel (VDK) incorporates scheduling and resource management tailored specifically to address the memory and timing constraints of DSP programming. These capabilities enable engineers to develop code more effectively, eliminating the need to start from the very beginning, when developing new application code. The VDK features include threads, critical and unscheduled regions, semaphores, events, and device flags. The VDK also supports priority-based, preemptive, cooperative, and time-sliced scheduling approaches. In addition, the VDK was designed to be scalable. If the application does not use a specific feature, the support code for that feature is excluded from the target system.

Because the VDK is a library, a developer can decide whether to use it or not. The VDK is integrated into the VisualDSP++ development environment but can also be used with standard command-line tools. The VDK development environment assists in managing system resources, automating the generation of various VDK-based objects, and visualizing the system state during application debug.

Analog Devices emulators use the IEEE 1149.1 JTAG test access port of the processor 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's JTAG interface—the emulator does not affect target system loading or timing.

In addition to the software and hardware development tools available from Analog Devices, third parties provide a wide range of tools supporting the Blackfin processor family. Hardware tools include the ADSP-BF54x EZ-KIT Lite standalone evaluation/development cards. Third party software tools include DSP libraries, real-time operating systems, and block diagram design tools.

2 CHIP BUS HIERARCHY

This chapter discusses on-chip buses, 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, including the interfaces between core buses and system buses.

The chapter includes the following sections:

- [“Overview” on page 2-1](#)
- [“System Overview” on page 2-8](#)
- [“Peripheral Access Bus \(PAB\)” on page 2-15](#)
- [“DMA-Related Buses” on page 2-17](#)
- [“External Access Bus \(EAB\)” on page 2-25](#)

Overview

This section provides an overview of the on-chip buses.

Internal Interfaces

[Figure 2-1](#) shows the processor core, on-chip peripherals, and the bus interfaces between them.

The processor core has several blocks of on-chip memory. The *L1 instruction memory* is 48K bytes SRAM plus 16K bytes that can be configured as a four-way set-associative cache or SRAM. The *L1 data memory* is

Overview

32K bytes SRAM plus 32K bytes that can be configured as a two-way set associative cache or SRAM. The *scratchpad SRAM memory* (not shown in [Figure 2-1](#)) consists of 4K bytes, which is only accessible as data SRAM (cannot be configured as cache memory). The *L1 instruction ROM memory* is factory programmed; this ROM is not customer-configurable. The *L2 SRAM memory* provides 128K bytes of unified instruction and data memory. Unlike L1 memory - which operates at the full core clock (CCLK) rate - the memory other than L1 operates at one half the frequency of the core. The *4K boot ROM* is seen as part of L3 memory. Because the boot ROM is outside the CCLK domain, this ROM operates at the system clock (SCLK) rate.

External memories, such as DDR and flash, can be accessed through the external bus interface unit (EBIU).

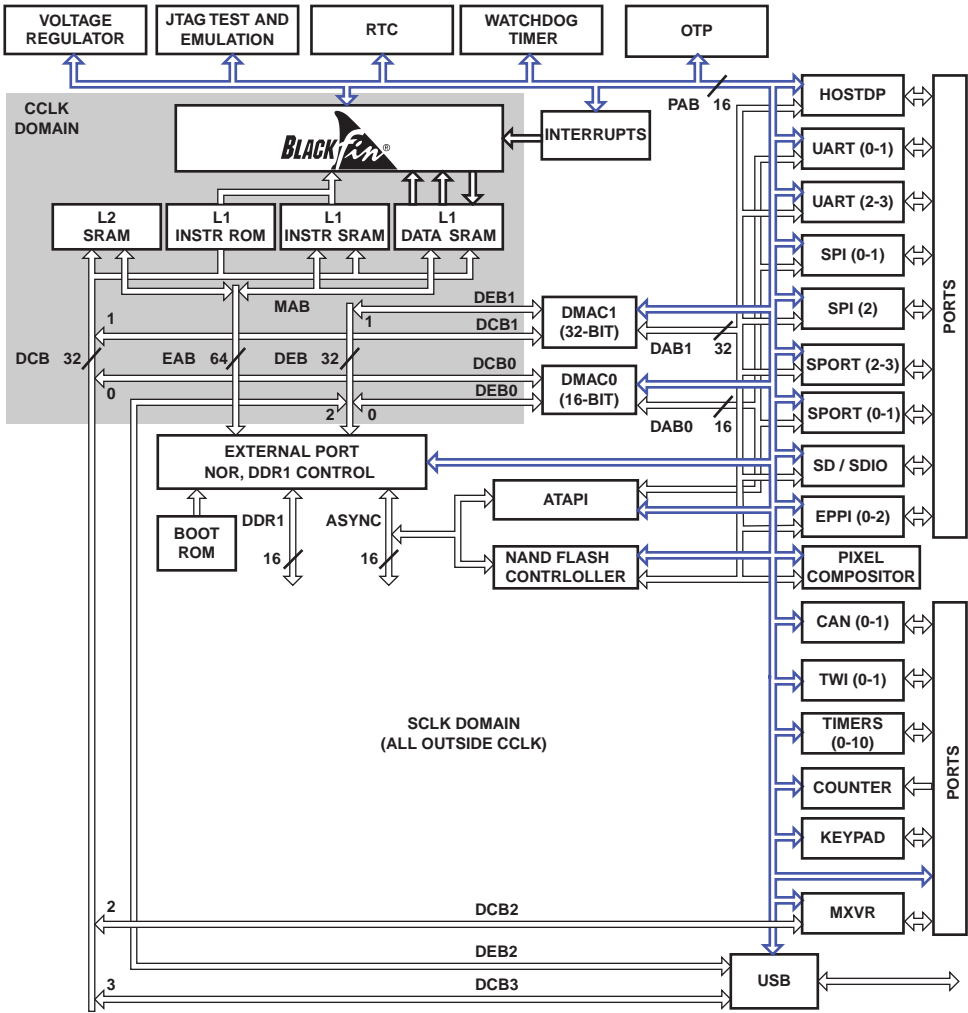


Figure 2-1. ADSP-BF54x Processor Bus Hierarchy

Overview

The ADSP-BF54x processor has many on-chip peripherals. The peripherals access memory in the processor core using a set of buses and DMA controllers. Two buses (DAB32 and DAB16) connect the peripherals and DMA controllers to support the data transfers between peripherals and memories.

The processor core has three ports connected to system and memory other than L1:

- 16-bit core port

This is the system memory-mapped register (MMR) access port. Through this port, the 16-bit peripheral access bus (PAB) connects all off-core system MMR registers. For more information, see [“Peripheral Access Bus \(PAB\)” on page 2-15](#).

- 64-bit core P port

This is the processor core L1 memory access port. The P port provides the interface to the external bus interface unit (EBIU) and to the memory other than L1 through the 32-bit external access bus (EAB) and the 64-bit processor core L2 bus separately. The memory access request commands from the core are pipelined; no arbitration logic is needed in this interface. For more information, see [“P Port Interface” on page 2-9](#).

- 32-bit core D port

DMA controllers (DMAC0, DMAC1, USB, and MXVR) transfer data to or from core L1 memory through this port. Because there are multiple DMA controllers that can simultaneously request access to L1 memory through the 32-bit D port, interface arbitration logic is provided and described in [“D Port Interface” on page 2-10](#).

Memory other than L1 also has two ports connected to the following two buses, which run at core clock frequency (CCLK domain):

- 64-bit core L2 bus

This bus supports memory other than L1 data/instruction accesses requested by the processor core.

- 32-bit system L2 bus (system L2 bus)

This bus supports DMAC0 and DMAC1 data transfers to or from L2; could be to or from L1, L2, or external memory.

Overall system functions of the ADSP-BF54x processor are supported by the following system buses, which run at the system clock frequency (SCLK domain):

- 16-bit peripheral access bus (PAB)
- 32-bit external access bus (EAB)
- 32-bit DMA core bus (DCB0, DCB1, DCB2, and DCB3)
- 32- and 16-bit DMA access buses (DAB0 and DAB1)
- 32-bit DMA external buses (DEB0, DEB1, and DEB2)

The DDR and ASYNC buses connect between the external bus interface unit (EBIU) and external memory. These buses run at the system clock frequency (SCLK domain).

Internal Clocks

The core processor clock (CCLK) rate is highly programmable with respect to the CLKIN input pin. The CCLK rate is divided down from the PLL output rate (VCO). This divider ratio is set using the CSEL parameter of the PLL_DIV register. For more information, see [“Phase-Locked Loop and Clock Control” on page 18-1](#).

Overview

The peripheral access bus (PAB), the DMA access buses (DAB32 and DAB16), the external access bus (EAB), the DMA core buses (DCB0, DCB1, DCB2, and DCB3), the DMA external buses (DEB0, DEB1, and DEB2), 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 CSEL parameter of the PLL_DIV register. Note that this divider must be set such that these buses run as specified in the processor data sheet, running at a speed 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. A subset of the peripherals derive their timing from the SCLK. For example, the UART baud rate is determined by further dividing this clock frequency.

Core Bus Overview

Figure 2-2 shows a processor core block diagram that includes a processor core and L1 memory connected by internal core buses. The core bus structure between the processor core and L1 memory runs at the full core frequency (CCLK domain). Data loads are performed using the LD0 and LD1 buses. The SD bus is used to perform writes. There are two address buses (DA0 and DA1) used for data fetches. The instruction address and data buses (IAB and IDB) are used to fetch instructions.

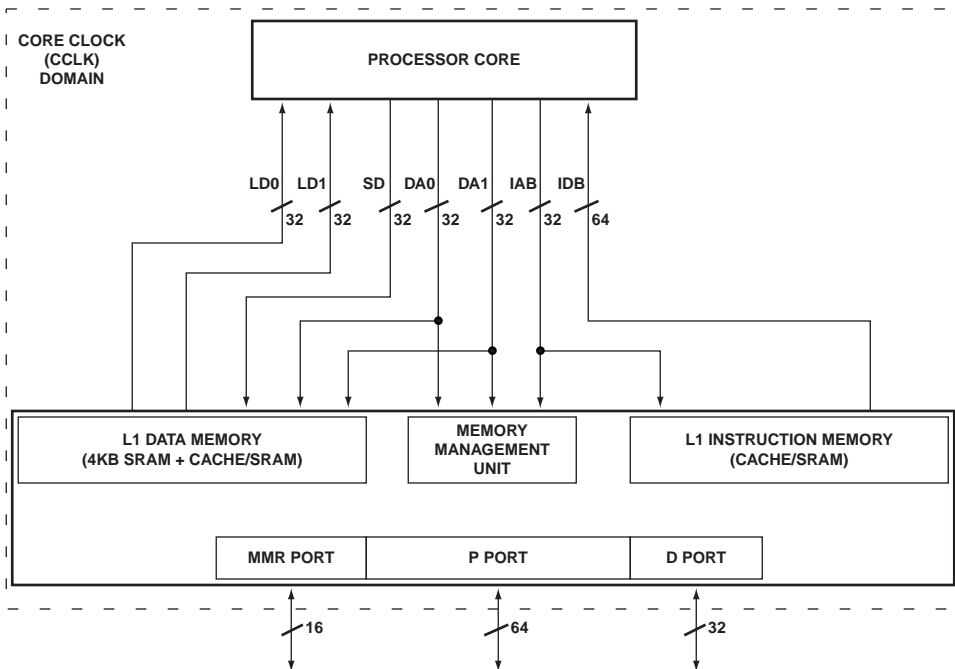


Figure 2-2. Processor Core and L1 Memory Block Diagram

These buses allow the processor core to perform the following L1 memory accesses per core clock cycle (CCLK):

- One 64-bit instruction fetch through the IDB bus
- One 32-bit data reference through the DA0 bus
- One 32-bit data reference through the DA1 bus
- Two 32-bit data loads through the LD0 and LD1 buses
- One 32-bit data store through the SD bus

System Overview

The processor core has three ports and can generate up to the following simultaneous off-core accesses per core clock cycle (CCLK):

- One DMA data transfer through the D port
- One L2 or external memory access through the P port
- One MMR register access through the MMR port

The L2 or external memory access through the P port includes normal data or instruction access and cache read or write operation.

System Overview

The ADSP-BF54x processor system includes a Blackfin processor core, a 128K byte level 2 (L2) memory, the peripheral set (see [Figure 2-1 on page 2-3](#)), the external memory controller (EBIU, AMC and DDR), the DMA controllers, and bus interfaces.

The external bus interface unit (EBIU) is the primary interface to the chip pins. Detailed information about the EBIU is discussed in [“External Bus Interface Unit” on page 6-1](#).

P Port Interface

Figure 2-3 shows the interface between the processor core P port and memory other than L1 through the 64-bit core L2 bus and shows the interface between processor core P port and the EBIU through the 32-bit EAB bus.

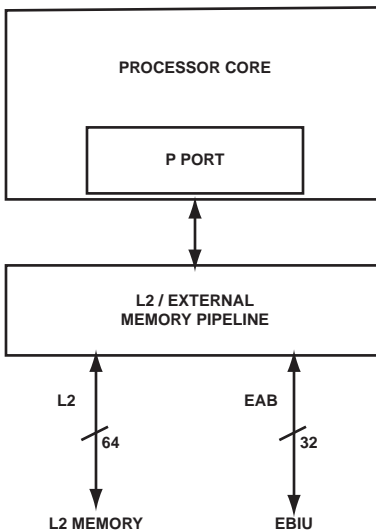


Figure 2-3. Core Interface to Memory other than L1 and the EBIU

At each `CCLK` cycle, the processor core can:

- Transfer one 64-bit instruction word from memory other than L1
- *Or* transfer one 32/16/8-bit data word to or from the same (or different) memory other than L1 data bank
- *Or* transfer one 32/16/8-bit data word to or from external memory

Data transfers requested from the processor core to L2 or the EBIU are fully pipelined.

D Port Interface

Figure 2-4 shows the interface between the DMA controllers core access buses (32-bit DCB buses) and the processor core's D port. This 32-bit D port provides DMA access to L1 memory.

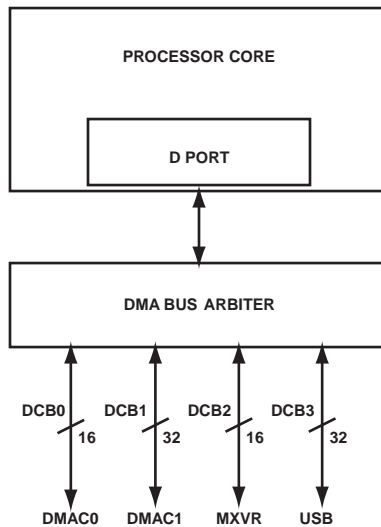


Figure 2-4. Core (A or B) Interface to DMA Controllers

The DCB buses provide the following DMA data transfers:

- The DCB0 bus supports up to 16-bit DMA data transfers between the processor core internal memory and peripheral on the DAB0 bus; or transfers between external memory and internal memory.

Where internal memory is L1, a DCB bus can also support internal memory to internal memory transfers. The DCB0 bus is in the SCLK domain.

- The DCB1 bus supports up to 32-bit DMA data transfer between the processor core internal memory and peripherals on the DAB1 bus; or between external memory and internal memory. The DCB1 bus is in the SCLK domain.
- The DCB2 bus supports up to 32-bit DMA data transfer between the processor core internal memory and MXVR. The DCB2 bus is in the SCLK domain.
- The DCB3 bus supports up to 32-bit DMA data transfer between the processor core internal memory and USB. The DCB3 bus is in the SCLK domain.

Because D port access requests can come from multiple independent DMA controllers, DMA bus arbitration is necessary to resolve possible D port access conflicts. The D port interface performs DMA bus (DCB0, DCB1, DCB2, and DCB3) arbitration, converts transactions on these buses to the core DMA bus protocol, and conducts transactions over the core DMA buses to L1 memory or over a separate bus to the memory other than L1. For more information on DMA priority arbitration, see [“DCB Arbitration” on page 2-21](#).

On-Chip L2 Interface

The L2 SRAM memory block is organized into eight banks that can be accessed by either two independent buses: the 64-bit processor core L2 bus or the 32-bit sys L2 bus. [Figure 2-5](#) shows this interface diagram. L2 is organized as a multi-bank architecture of single-ported SRAMs, such that multiple accesses can occur in parallel, as long as they are to different banks. L2 has two ports: the processor core L2 port is connected to the 64-bit processor core L2 bus and dedicated to processor core access requests.

System Overview

The sys L2 port is connected to the 32-bit sys L2 bus and dedicated to system DMA access requests. Two different banks can be accessed simultaneously by the 64-bit processor core L2 bus and the 32-bit system L2 bus. When both buses attempt to access the same bank at the same time, the L2 arbitration logic resolves the conflict.

An L2 access requires two `CCLK` cycles for the access itself, plus any latency involved in the operation (see [Table 2-2 on page 2-14](#)). L2 interface control logic is clocked at the core frequency (`CCLK` clock domain). The system DMA access request comes from the DCB0, DCB1, DCB2, and the DCB3 busses, which run at system clock frequency (`SCLK` domain). The interface circuit synchronizes the DCB buses to the core clock domain and converts them to system L2 bus protocol.

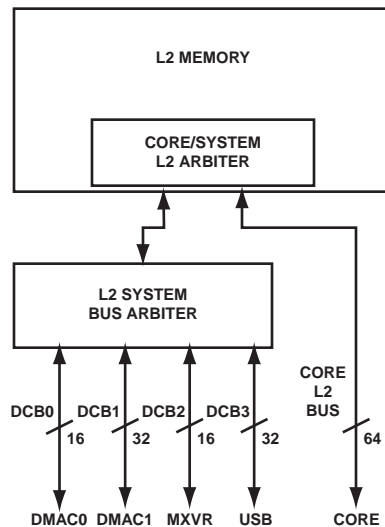


Figure 2-5. L2 Bus Interfaces

As shown in [Figure 2-5 on page 2-12](#), there are several arbitration stages in the interface:

- Arbitration for core L2 port access requests is based on a fixed priority scheme. There is no arbitration while the current Core L2 bus requestor is performing locked or cache line fill transactions.

After the processor core is granted the core L2 bus, no other user can access this bus until the data transaction is accepted by L2.

- Arbitration for system L2 port access request is based on a programmable priority scheme. After reset, the following fixed priority is maintained: DCB2 (MXVR) > DCB0 (DMAC0) > DCB1 (DMAC1) > DCB3 (USB) for L2 accesses through the system L2 bus. The priorities between the DCB0 bus and DCB1 bus is programmable. This can be using the L2DMAPRO bit in the SYSCR register.
- When both the processor core L2 bus and the system L2 bus attempt to access the same bank at the same time, bank arbitration is required. [Table 2-1](#) shows the L2 access bus arbitration.

Table 2-1. L2 Interface Bus Arbitration

Requestor	Priority (L2DMAPRIO = 0) (Default)	Priority (L2DMAPRIO = 1)
Currently locked core access	1	1
Complete current core cache access	2	2
DCB2 (MXVR)	3	3
DCB0 (DMAC0)	4	5
DCB1 (DMAC1)	5	4
DCB3 (USB)	6	6
Core L2	7	7

System Overview

[Table 2-2 on page 2-14](#) shows the target latency and throughput for various types of accesses. Since the DMA bus has a dedicated port to the L1 and L2 memories, as long as the processor core access and DMA access are not to the same memory bank, no stalls occur. DMA access to L1 or memory other than L1 can only be stalled by:

- An access already in progress from another DMA controller/channel
- A core access already in progress, which locks the bank to be addressed

For more details about DMA data transfer latency and DMA traffic control/optimization, please see [“DMA Performance” on page 5-51](#).

Table 2-2. L2 Interface Data and Instruction Fetch Transaction Latency

Transaction Type	Number of Cycles to Complete
Core L2 Read	9 CCLKs for each read
Dual DAG Read (same instruction)	9 CCLKs (first 32-bit fetch) 2 CCLKs (second 32-bit fetch)
Cache Line Fill (data and instruction)	9 CCLKs (first 64-bit fetch) 2-2-2 CCLKs (for next three 64-bit fetches)
Dual DAG Cache Line Miss (same instruction)	9-2-2-2 CCLKs (first miss, four 64-bit fetches) 2-2-2-2 CCLKs (second miss, four 64-bit fetches)
64-bit Instruction Fetch	9 CCLKs
Sys DMA Read	1 SCLK plus 2 CCLKs
Sys DMA Write	1 SCLK

When executing code from memory other than L1, a core can fetch a 64-bit word. In the best case, the 64-bit word contains four 16-bit instructions. For consecutive fetches of single-cycle, 16-bit instructions, the maximum execution rate is four instructions every nine CCLKs—due to pre-fetching by the core.

When the processor core writes to memory other than L1, a write buffer within the interface of each core improves performance. Up to five writes can be made to memory other than L1 without stalling a core. The sixth write, and subsequent writes when the buffer is full, take four CCLKs for each write. Specifically, a loop of eight writes to memory other than L1 would take five CCLKs for the first five writes plus four CCLKs for each of the three subsequent writes.

Peripheral Access Bus (PAB)

The ADSP-BF54x has a dedicated peripheral access bus (PAB) that connects all off-core peripherals to system MMR registers. The low-latency peripheral access bus keeps core stalls to a minimum and allows for manageable interrupt latencies to time-critical peripherals. All peripheral resources accessed through the PAB bus are mapped into the system MMR space of the ADSP-BF54x memory map.



The processor core is the only master on the PAB bus. No arbitration is necessary.

PAB Performance

For the PAB, the primary performance criteria is latency, not throughput. Transfer latencies for write transfers on the PAB are two SCLK cycles, and transfer latencies for read transfers on the PAB are three SCLK cycles.

For example, the core can transfer up to 32 bits per access to the PAB slaves. With the core clock running at two times the frequency of the system clock, the first and subsequent system MMR write accesses take four 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. System MMR addresses are listed in [“System MMR Assignments” on page A-1](#).

The slaves on the PAB bus are:

- Event Controller
- Clock and Power Management Controller
- Watchdog Timer
- Real Time Clock
- Timer 0–10
- SPORT 0–3
- SPI 0–2
- General-Purpose Input/Output (GPIOs)
- UART 0–3
- ATAPI
- EPPI0-2
- Pixel Compositor
- Secure Digital Host (SDH)
- USB
- MXVR
- TWI 0–1

- CAN 0–1
- Asynchronous Memory Controller (AMC)
- DDR SDRAM Controller (DDC)
- DMA Controller 0–1 (DMAC0 and DMAC1)

DMA-Related Buses

Figure 2-6 shows the DMA bus connections. These buses run at the system clock frequency (SCLK domain).

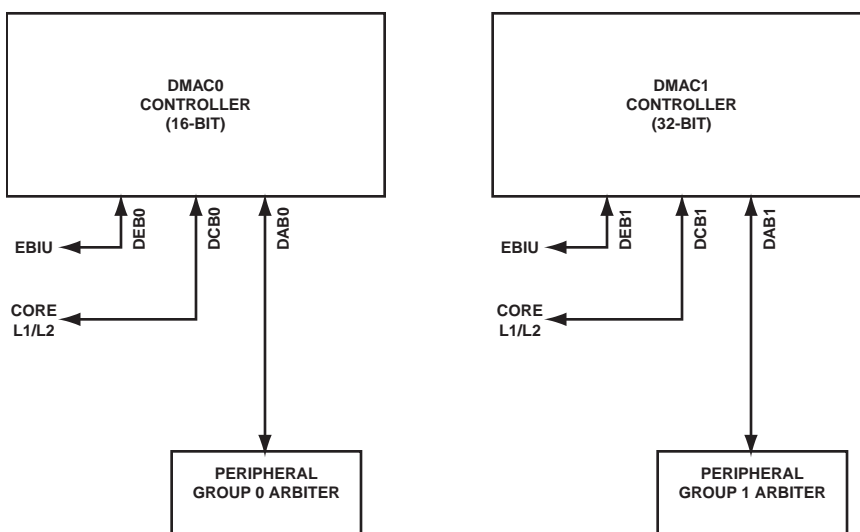


Figure 2-6. DMA Bus Connection and Arbitration Block Diagram

The 32/16-bit DAB bus provides DMA between the peripherals and L1/L2 internal memory through the DCB bus, or between peripherals and external memories through the DEB bus. A central DMA controller keeps track of DMA addresses and mediates the transfers. DMA is handled iden-

DMA-Related Buses

tically for 8-, 16- and 32-bit data sizes. The maximum bandwidth for any individual 16-bit peripheral is one 16-bit word transferred for every two SCLK cycles. Peripherals that are capable of 32-bit DMA (and also configured for 32-bit mode) can transfer up to one 32-bit word every two SCLK cycles.

Peripheral DMA

The DMA-capable peripherals in the ADSP-BF54x system are managed by DMA controllers. Each DMA controller also has memory DMA channels for DMA data transfer between external memory and L1 or memory other than L1. The peripheral DMA controllers can transfer data between peripherals and internal or external memory.

The DCB bus arbitration for L2 configured as SRAM is shown in [Table 2-1 on page 2-13](#). The ADSP-BF54x has programmable priority for peripherals on the DAB bus. For details about programmable DMA peripheral and DMA channel mapping, see [“Direct Memory Access” on page 5-1](#).

DAB Bus Agents (Masters)

All peripherals capable of sourcing a DMA access through one of the centralized DMA controllers are masters on these buses, as shown in [Table 2-3 on page 2-20](#) and [Table 2-4 on page 2-21](#). A single arbiter supports a programmable priority arbitration policy for access to each DAB.

When two or more DMA master channels are actively requesting a 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. The MXVR and USB peripherals have their own DMA channels and are not part of the DAB.

DAB Arbitration

There are two centralized DMA controllers in the system which together support 14 peripherals and four memory DMA channels. 32 DMA channels and bus masters support these devices, with eight channels being assigned to memory DMA, and the remaining 24 channels being assigned to peripheral DMA. The memory DMA channels can transfer data between L1, L2, and external memory. The peripheral DMA controllers can transfer data between peripherals and internal (L1/L2) 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. DAB0 offers 16 bits of data transfer per `SCLK` cycle and DAB1 offers 32 bits of data transfer per `SCLK` cycle. Arbitration of channels on the DAB bus is programmable within each centralized DMA controller. [Table 2-3](#) and [Table 2-4](#) show the default arbitration priority of each DMA controller.

DMA-Related Buses

Table 2-3. Controller 0 (DAB0) Arbitration Priority

DAB, DCB, DEB Master	Default Arbitration Priority
SPORT0 Rx DMA Controller	0 - highest
SPORT1 Rx DMA Controller	1
SPORT0 Tx DMA Controller	3
SPORT1 Tx DMA Controller	2
SPI0 DMA Controller	4
SPI1 DMA Controller	5
UART0 Rx DMA Controller	6
UART0 Tx DMA Controller	7
UART1 Rx DMA Controller	8
UART1 Tx DMA Controller	9
ATAPI Rx DMA Controller	10
ATAPI Tx DMA Controller	11
Memory DMA0 (dest) Controller	12
Memory DMA0 (source) Controller	13
Memory DMA1 (dest) Controller	14
Memory DMA1 (source) Controller	15 - lowest

Table 2-4. Controller 1 (DAB1) Arbitration Priority

DAB, DCB, DEB Master	Default Arbitration Priority
EPPI0 DMA Controller	0 - highest
EPPI1 DMA Controller	1
EPPI2 DMA Controller/Host DMA Port	2
Pixel Compositor DMA Controller 0 (input data)	3
Pixel Compositor DMA Controller 1 (overlay data)	4
Pixel Compositor DMA Controller 2 (output data)	5
SPORT2/UART2 Rx DMA Controller	6
SPORT2/UART2 Tx DMA Controller	7
SPORT3/UART3 Rx DMA Controller	8
SPORT3/UART3 Tx DMA Controller	9
SDH/NAND Flash DMA Controller	10
SPI2 DMA Controller	11
Memory DMA 2 (dest) Controller	12
Memory DMA 2 (source) Controller	13
Memory DMA 3 (dest) Controller	14
Memory DMA 3 (source) Controller	15 - lowest

DCB Arbitration

Each of the two centralized DMA controllers as well as the MXVR transceiver and the USB controller, access L1 memory through the DCB buses (DCB0/DCB1/DCB2/DCB3). In the event of simultaneous requests to L1 memory, access is granted based on a programmable arbitration scheme.

DMA-Related Buses

The DCB buses have priority over the core processor on arbitration into L1 configured as SRAM. These same buses are used to access memory other than L1, which has a similar arbitration scheme. L1 and L2 accesses from the DMA controllers may happen in parallel.

Into L1 and memory other than L1, an access by the system bus always wins over an access by the core. On the system bus, by default, the order of priority is:

1. MXVR
2. DMAC0
3. DMAC1
4. USB

The priority order for DMAC0 and DMAC1 may be swapped. [Table 2-5](#) describes the priority configuration for L1 accesses, which is defined by the `CDMAPRIO` bit of the `SYSCR` register. For L2 accesses, the `L2DMAPRIO` bit in `SYSCR` is used in the same way. For more information, see [“System Reset Configuration \(SYSCR\) Register”](#) on page 17-109.

Table 2-5. D Port DCB0 (DMAC0) and DCB1 (DMAC1) Arbitration

DMA Controllers	Priority (CDMAPRIO = 0, default)	Priority (CDMAPRIO = 1)
DMAC0	1	2
DMAC1	2	1

If any of the DMA channels is urgent, it is elevated above the others in terms of priority. For example, an urgent USB DMA channel is higher priority than a non-urgent DMAC0 channel.

DEB Arbitration

Each of the two DMA controllers, as well as the USB controller, access external memory through the DEB buses (DEB0/DEB1/DEB2).

 The MXVR does not have DMA access to external memory.

In the event of simultaneous requests to external memory, access is granted based on a programmable arbitration scheme. This priority can be changed by using the `DEB_ARB_PRIORITY` bits in the `EBIU_DDRQUE` register. For off-chip memory, the core has priority over the DEB buses by default. However, the priorities of the specific DMA bus with respect to the core can be changed for both synchronous and asynchronous accesses. The complete arbitration at the EBIU is described in [“External Bus Interface Unit” on page 6-1](#).

DAB, DCB, and DEB Performance

The ADSP-BF54x DAB buses support 8-bit, 16-bit, and 32-bit data transfers. DAB1 is a 32-bit data bus. DAB0 is a 16-bit bus. Both operate at the system clock rate, at a maximum frequency of 133 MHz, although a single peripheral DMA channel on a DAB bus operates at a maximum of $SCLK/2$. The DCB buses have a dedicated D port into L1 memory and another dedicated sys L2 port into memory other than L1. No stalls occur as long as the core access and the DMA access are not to the same memory bank. If there is a conflict, DMA is the highest priority requester, followed by the core. 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.

DMA-Related Buses

Memory DMA transfers can 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.

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 an additional cycle between each transfer.

Table 2-6 shows many types of 32-bit memory DMA transfers (on DMAC1). In the table, it is assumed that no other DMA activity is conflicting with ongoing operations. The numbers in the table are theoretical values. These values may be higher when they are measured on actual hardware due to a variety of reasons relating to the device that is connected to the EBIU.

For non-DMA accesses (for example, a core access through the EAB), a 32-bit access to DDR SDRAM (of the form $R0 = [P0]$; where P0 points to an address in DDR SDRAM) always more efficient than executing two 16-bit accesses (of the form $R0 = W[P0++]$; where P0 points to an address in DDR SDRAM). In this example, a 32-bit DDR SDRAM read takes ten SCLK cycles while two 16-bit reads take nine SCLK cycles each.

Table 2-6. Performance of DMA Access (on DMAC1) to External Memory

Source	Destination	Approximate SCLKs for n Words (Max word size 32-bits) (From Start of DMA to Interrupt at End)
16-bit DDR SDRAM	L1 Data memory	$n + 14$
L1 Data memory	16-bit DDR SDRAM	$n + 11$
16-bit Async memory	L1 Data memory	$xn + 12$, where x is the number of wait states + setup/hold SCLK cycles (minimum $x = 2$)

Table 2-6. Performance of DMA Access (on DMAC1) to External Memory (Cont'd)

Source	Destination	Approximate SCLKs for n Words (Max word size 32-bits) (From Start of DMA to Interrupt at End)
L1 Data memory	16-bit Async memory	$xn + 9$, where x is the number of wait states + setup/hold SCLK cycles (minimum $x = 2$)
16-bit DDR SDRAM	16-bit DDR SDRAM	$10 + (17n/7)$
16-bit Async memory	16-bit Async memory	$10 + 2xn$, where x is the number of wait states + setup/hold SCLK cycles (minimum $x = 2$)
L1 Data memory	L1 Data memory	$2n + 12$

The EAB and the DEB buses must arbitrate for access to external memory through the EBIU. [Figure 2-6 on page 2-17](#) shows the bus connection to the EBIU and the bus arbiters. Users must manage specific memory access traffic patterns to ensure that isochronous peripherals have enough allocated bandwidth and appropriate maximum data latency for both internal and external memory accesses.

External Access Bus (EAB)

The external access bus (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. The EAB supports single-word accesses of either 8-bit, 16-bit, or 32-bit data types. The EAB operates at the system clock rate.

External Access Bus (EAB)

EAB/DEB Arbitration

Arbitration for use of external memory interface resources (DDR or ASYNC) is required because of possible contention between the potential masters of these resources. A fixed-priority arbitration scheme is used to arbitrate between EAB accesses and DEB accesses, with core accesses winning by default. For more details on arbitration, see [“External Bus Interface Unit” on page 6-1](#). For information on external memory interface resources, see [“DDR SDRAM Memory Interface” on page 6-17](#) or [“Asynchronous Memory Interface” on page 6-53](#).

EAB/DEB Performance

The EAB supports single-word accesses of 8-bit, 16-bit, 32-bit, or 64-bit data types. The EAB operates at the same frequency as the PAB and the DAB, up to the maximum SCLK frequency specified in the ADSP-BF54x data sheet.

Table 2-7 shows many types of 16-bit and 32-bit memory DMA transfers. In the table, it is assumed that no other DMA activity is conflicting with ongoing operations.

Table 2-7. Performance of DMA Access (on DMAC0) to External Memory

Source	Destination	Approximate SCLKs For n 16-bit Words (From Start of DMA to Interrupt at Rnd)	Approximate SCLKs For n 32-bit Words (From Start of DMA to Interrupt at end) ¹
16-bit DDR SDRAM	L1 Data memory	$n + 14$	$2n + 14$
L1 Data memory	16-bit DDR SDRAM	$n + 14$	$2n + 14$
16-bit Async memory	L1 Data memory	$xn + 12$, where x is the number of wait states + setup/hold SCLK cycles (minimum $x = 2$)	$2xn + 12$, where x is the number of wait states + setup/hold SCLK cycles (minimum $x = 2$)
L1 Data memory	16-bit Async memory	$xn + 12$, where x is the number of wait states + setup/hold SCLK cycles (minimum $x = 2$)	$2xn + 12$, where x is the number of wait states + setup/hold SCLK cycles (minimum $x = 2$)
16-bit DDR SDRAM	16-bit DDR SDRAM	$10 + (17n/7)$	$10 + 2*((17n/7))$
16 bit Async memory	16-bit Async memory	$10 + 2xn$, where x is the number of wait states + setup/hold SCLK cycles (minimum $x = 2$)	$10 + 2*(2xn)$, where x is the number of wait states + setup/hold SCLK cycles (minimum $x = 2$)
L1 Data memory	L1 Data memory	$2n + 12$	$2*2n + 12$

¹ Note that DMAC0 is only a 16-bit controller although it can be programmed for 32-bit word accesses. For 32-bit accesses it will take twice as much SCLK cycles as compared to transactions on DMAC1.

External Access Bus (EAB)

The corresponding access time for EAB accesses (assuming rows are open and pre-charged) are:

- 16-bit processor core read from DDR – 8 SCLK cycles
- 32-bit processor core read from DDR – 8 SCLK cycles
- 32 byte cache line fill (8, 4 byte accesses) - $8 + (7 \times 1)$ SCLK cycles

3 MEMORY

This chapter includes the following sections:

- “Memory Architecture” on page 3-2
- “Instruction Test Registers” on page 3-24
- “L1 Data Memory” on page 3-28
- “Data Test Registers” on page 3-44
- “On-Chip Level 2 (L2) Memory” on page 3-49
- “One Time Programmable Memory” on page 3-53
- “External Memory” on page 3-53
- “Memory Protection and Properties” on page 3-54
- “Memory Transaction Model” on page 3-74
- “Load/Store Operation” on page 3-75
- “Working With Memory” on page 3-81
- “Terminology” on page 3-84

Memory Architecture

The ADSP-BF54x 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 provide faster access. Level 2 (L2) memories are on-chip memory systems (which are farther from the core) and typically have longer access latencies. The faster L1 memories, which include instruction SRAM and instruction ROM, data, and scratchpad memory as part of the Blackfin core are accessed in a single cycle. The L2 memories, which include an on-chip SRAM and off-chip synchronous and asynchronous devices, provide much higher memory space with higher latency.

The ADSP-BF54x 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, 272M byte of address space is dedicated to internal, on-chip resources. The ADSP-BF54x processor populates portions of this internal memory space with:

- L1 and L2 static random access memories (SRAM)
- L1 instruction ROM (IROM)
- 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 ADSP-BF54x processor also provides support for an external memory space that includes asynchronous memory space and DDR space. See [Chapter 6, “External Bus Interface Unit,”](#) for a detailed discussion of each of these memory regions and the controllers that support them.

The diagram in [Figure 3-1 on page 3-4](#) provides an overview of the ADSP-BF54x system memory map. Note that 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.

As shown in [Figure 3-1 on page 3-4](#), total on-chip memory for the DSP core occupies 100 Kbytes, as follows:

- 64K byte of instruction SRAM memory:
 - 48K byte of instruction SRAM
 - 16K byte of instruction cache/SRAM, lockable by way or line
- 64K byte of instruction ROM
- 64K byte of data memory:
 - 32K byte of data cache/SRAM
 - 32K byte of SRAM
- 4K byte of data scratch pad SRAM
- 4K byte of boot ROM

An on-chip SRAM provides 128K byte of L2 space. For systems using some or all ADSP-BF54x processor L1 memory as cache, the on-chip L2 SRAM memory can help provide deterministic, bounded-memory access times.

The upper portion of internal memory processor space is allocated to the core and system MMRs of the ADSP-BF54x processor. Accesses to this area are allowed only when the processor is in supervisor mode or emulation mode. (For information about these modes, see the appropriate *Blackfin Processor Programming Reference*.)

The lowest 4K byte of internal memory space is occupied by the boot ROM of the ADSP-BF54x processor. Depending on the booting option selected, the appropriate boot program is executed from this memory space when the ADSP-BF54x processor is reset. See [“System Reset and Booting” on page 17-1](#).

Memory Architecture

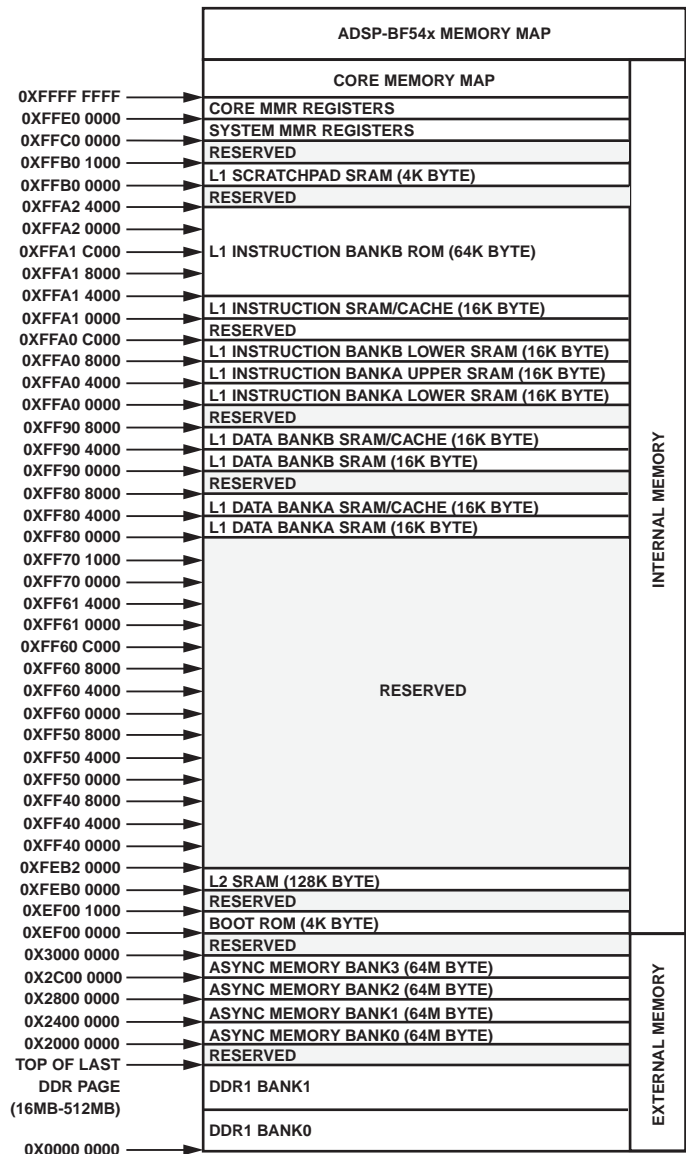


Figure 3-1. Memory Map

Within the external memory map, four banks of asynchronous memory space and two banks of DDR memory are available. Each of the asynchronous banks is 64M byte and each of the synchronous banks can be configured 8-256 M byte.

Internal Memory

The ADSP-BF54x 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 chip. The ADSP-BF54x processor supports this memory architecture for applications that require direct control over access time.

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 the L1 memories. Code can be ported to or developed for the ADSP-BF54x processor quickly without requiring performance optimization for the memory organization.

Each core's 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

Memory Architecture

- 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 L1 Instruction SRAM

The 64 Kbyte L1 instruction SRAM consists of a dedicated 48 Kbyte SRAM plus an additional 16 Kbyte bank which can be configured as either SRAM or cache. The upper 16 Kbyte, L1 instruction memory can be configured as a 4-way set-associative cache (see [Figure 3-4 on page 3-16](#)). Consequently, instructions can be brought into four different ways of cache, decreasing the frequency of cache-line replacements and increasing overall performance. When the upper 16 Kbyte of L1 memory is configured as a cache, individual ways or lines of L1 instruction cache can be locked down, allowing further control over the location of time-critical code. The cache-locking concept is explained further in [“Instruction Cache Locking by Way” on page 3-22](#). When configured as SRAM, each of the four 16K byte banks of memory is broken into 4K byte sub-banks which can be independently accessed by the processor and DMA. For more information about L1 instruction SRAM, see [“L1 Instruction SRAM” on page 3-12](#).

Overview of L1 Instruction ROM

The 64K byte L1 instruction ROM consists of a single 64K byte bank of read-only memory. The instruction ROM does not have 4K byte sub-banks which can be independently accessed by the processor and DMA. At every processor cycle either the processor or the DMA is able to access the instruction ROM. The instruction ROM is completely contained within instruction bank B without sub-bank divisions.

Write accesses to the instruction ROM region do not generate errors nor do they modify the data in the ROM. They take the same number of cycles to execute as if the write was actually occurring.

Multiple read accesses to the instruction ROM region behave as if they were reads to a single instruction bank B sub-bank.

Overview of L1 Data SRAM

Each core on the ADSP-BF54x processor provides two 32K byte, L1 data SRAM banks (data bank A and data bank B). Each data bank has a dedicated lower 16K byte SRAM bank plus an additional upper 16K byte bank which can be configured as SRAM or cache.

When configured as cache, the upper 16K byte bank in each L1 data bank is a 2-way, set-associative structure. This provides two separate locations that can hold cached data, decreasing the rate of cache-line replacements and increasing overall performance.


If configured as SRAM, each of the two upper 16K byte banks of memory is broken into four 4 Kbyte sub-banks which can be independently accessed by the processor and DMA. For more information about L1 data SRAM, see [“L1 Data SRAM” on page 3-31](#).

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.

Memory Architecture

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.

-  The L1 memories operate at the core clock frequency (CCLK). Scratchpad data SRAM cannot be accessed by the DMA controller.


Overview of On-Chip L2

The on-chip level 2 (L2) memory provides 128 Kbyte of low latency, high-bandwidth capacity. This memory system is referred to as on-chip L2 because it forms an on-chip memory hierarchy with L1 memory. On-chip L2 provides more capacity than L1 memory, but the latency is higher. The on-chip L2 is SRAM and cannot be configured as cache. It is capable of storing both instructions and data. The L1 caches can be configured to cache instructions and data located in the on-chip L2.

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 16 Kbyte 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 any of the following:

- A simple SRAM
- A 4-way, set-associative instruction cache
- A cache with as many as four locked ways

-  L1 instruction memory can be used only to store instructions.

Instruction Memory Control Register (IMEM_CONTROL)

The instruction memory control (IMEM_CONTROL) register contains control bits for the L1 instruction memory. By default after reset, cache and cacheability protection lookaside buffer (CPLB) address checking is disabled (see [“L1 Instruction Cache” on page 3-15](#)).

When the LRUPRIORST bit is set to 1, the cached states of all CPLB_LRUPRIO bits (see [“ICPLB Data Registers \(ICPLB_DATAx\)” on page 3-63](#)) 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 3-21](#) 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.

Memory Architecture

L1 Instruction Memory Control Register (IMEM_CONTROL)

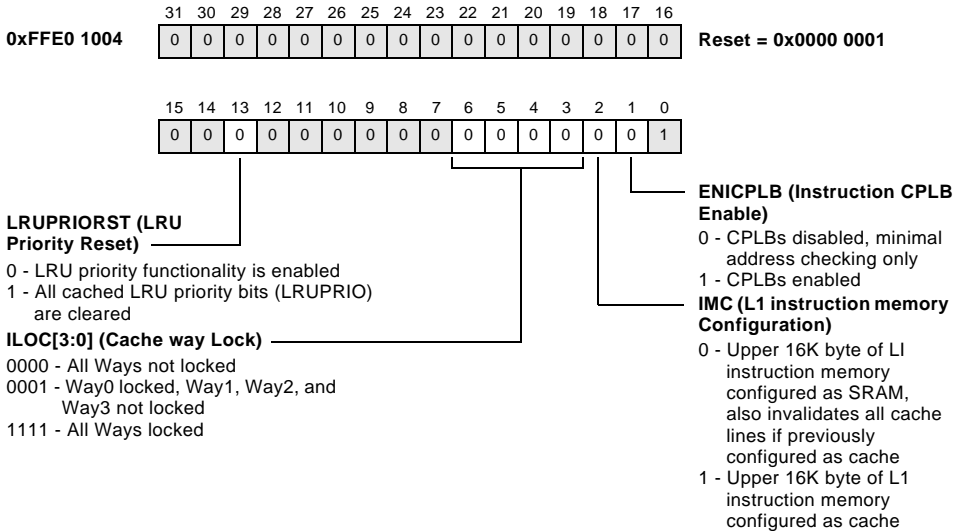



Figure 3-2. L1 Instruction Memory Control Register

The `ILOC[3:0]` bits provide a useful feature only after code is manually loaded into cache. See [“Instruction Cache Locking by Way” on page 3-22](#). These bits specify which ways to remove from the cache replacement policy. This has the effect of locking code present in non-participating ways. Code in non-participating ways can still be removed from the cache using an `IFLUSH` 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 does not alone enable memory other than L1 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.

 Reserving memory to serve as cache does not alone enable memory other than L1 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 (nonpopulated) 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 3-77), disabling of CPLBs should be preceded by a CSYNC.

 When enabling or disabling cache or CPLBs, immediately follow the write to IMEM_CONTROL with a SSYNC 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 ADSP-BF54x 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 cannot access L1 instruction memory directly. A DAG reference to instruction memory SRAM space generates an exception. (For information about DAG addressing, see the appropriate *Blackfin Processor Programming Reference*.)

Write access to the L1 instruction SRAM memory must be made through the 64-bit system DMA port. Because the SRAM is implemented as a collection of single-ported sub-banks, the instruction memory is effectively dual-ported. Provided that system and core accesses do not access the same 32-bit polarity (address bits 2 match) of the same sub-bank, effective dual-porting of the instruction memory is achieved. If both system and core attempt to access the same 32-bit polarity (address bits 2 match) of the same bank, the system DMA controller has priority over the core instruction fetch.

Table 3-1 lists the instruction memory sub-banks.

Table 3-1. L1 Instruction Memory Sub-banks

Memory Sub-bank	Memory Start Location
0	0xFFA0 0000
1	0xFFA0 1000
2	0xFFA0 2000
3	0xFFA0 3000
4	0xFFA0 4000
5	0xFFA0 5000
6	0xFFA0 6000
7	0xFFA0 7000
8	0xFFA0 8000
9	0xFFA0 9000
10	0xFFA0 A000
11	0xFFA0 B000
12	0xFFA1 0000
13	0xFFA1 1000
14	0xFFA1 2000
15	0xFFA1 3000


 Before changing the configuration state, be sure to flush the cache or move all modified data from the SRAM, if so configured.

Figure 3-3 on page 3-14 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.

Memory Architecture

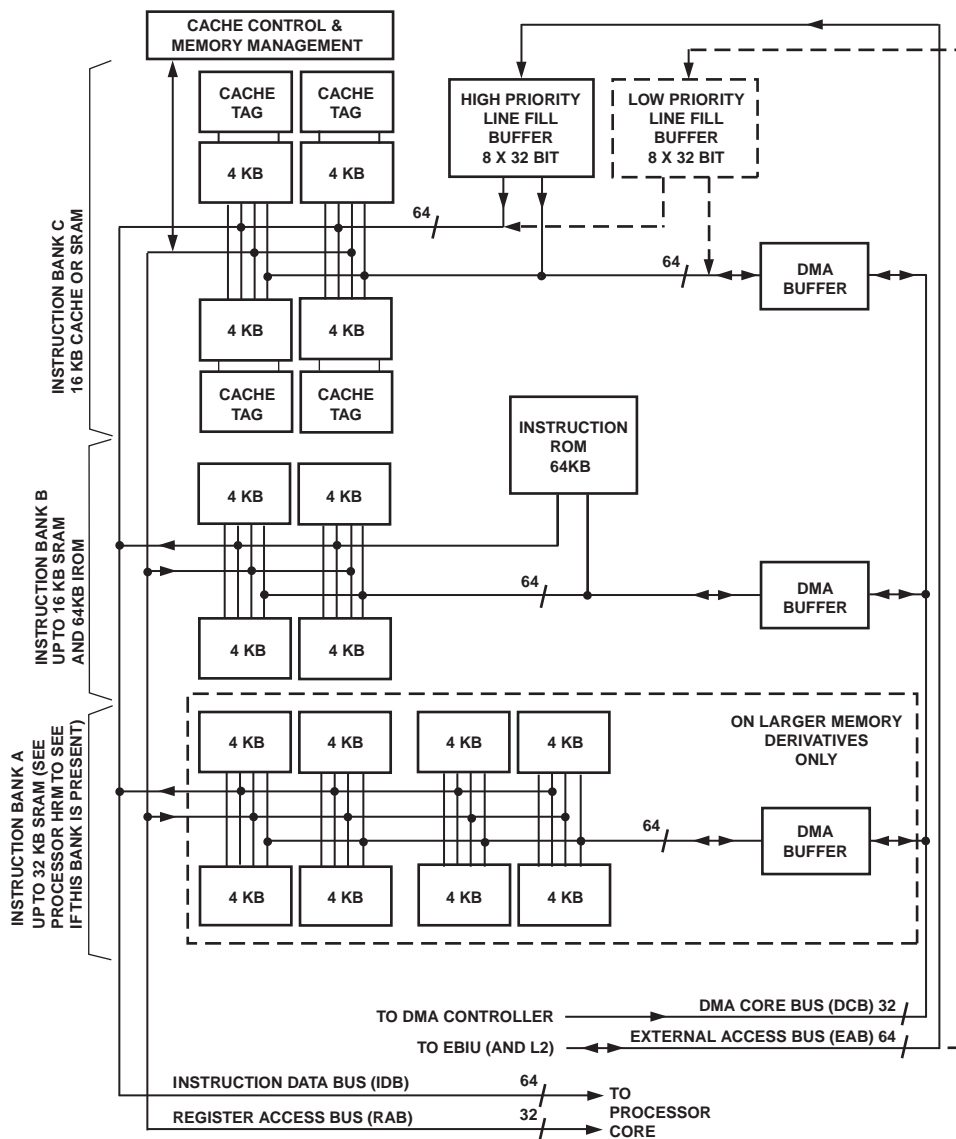


Figure 3-3. L1 Instruction Memory Bank Architecture

L1 Instruction Cache

The L1 instruction memory may also be configured as a flexible, 4-way set-associative instruction 16 Kbyte cache. To improve the average access latency for critical code sections, each way 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 specified as cacheable by cacheability protection lookaside buffers (CPLBs) are 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 3-54](#).

[Figure 3-4 on page 3-16](#) shows the overall Blackfin processor instruction cache organization.

Cache Lines

As shown in [Figure 3-4](#), 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.

Memory Architecture

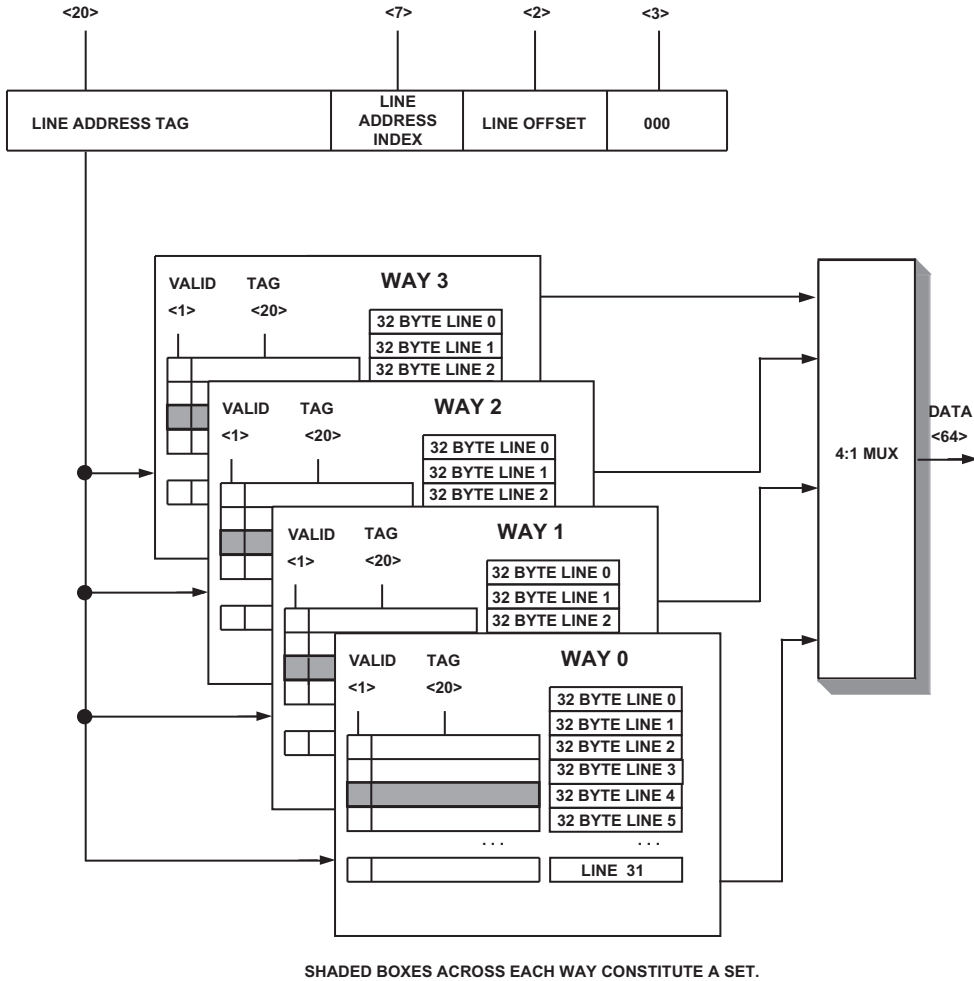


Figure 3-4. Blackfin Processor Instruction Cache Organization

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 4 Kbyte 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.

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 is 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.

The tag and data components of a cache line are illustrated in [Figure 3-5](#).

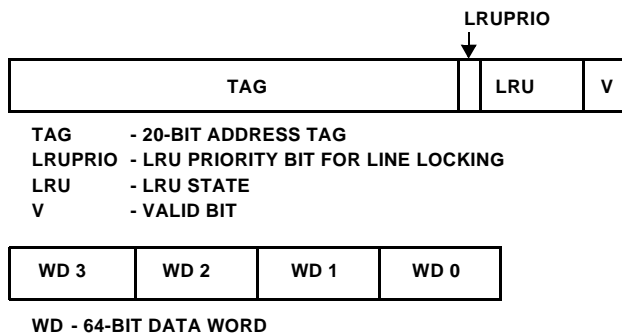


Figure 3-5. Cache Line – Tag and Data Portions

Memory Architecture

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 hit occurs, the target 64-bit instruction word is first sent to the instruction alignment unit (IAU) where it is stored in one of two 64-bit instruction buffers.

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 on-chip L2 or 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 on-chip L2 or 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 on-chip L2 or 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 on-chip L2 or 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 3-2](#).

Table 3-2. Cache-Line Word-Fetching Order

Target Word	Fetching Order for Next Three Words
WD0	WD0, WD1, WD2, WD3
WD1	WD1, WD2, WD3, WD0
WD2	WD2, WD3, WD0, WD1
WD3	WD3, WD0, WD1, WD2

Line-Fill Buffer

As the new cache line is retrieved from on-chip L2 or external memory, each 64-bit word is buffered into one of two four-entry line-fill buffer before it is written to a 4 Kbyte 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 on-chip L2 or external memory, rather than having to wait until the line is written into the cache.

Two separate line-fill buffers are provided to allow a load from slow external memory to continue without causing jumps to higher speed on-chip memory other than L1 to stall. The CPLB_MEMLEV bit in the memory pages CPLBs determines which line buffer is used. See [“Memory Protection and Properties” on page 3-54](#).

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 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 3-4, “Blackfin Processor Instruction Cache Organization,” on page 3-16](#)).

Memory Architecture

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.

For more information, see [“Instruction Cache Invalidation”](#) on page 3-23.

Instruction Cache Locking by Line

The `CPLB_LRUPRIO` bits in the `ICPLB_DATAx` registers (see [“Memory Protection and Properties”](#) on page 3-54) 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 `LRUPRIRST` bit in the `IMEM_CONTROL` register (see [page 3-9](#)).

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 $IFLUSH$ instruction, or “backdoor” MMR assisted manipulation of the tag array.

An example sequence is provided 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 Invalidation” on page 3-23](#)).
- 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 $ILOC[0]$), are 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 an address, cache line, or a 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 is 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 ADSP-BF54x processor memory map, it is impractical to use this instruction to invalidate an entire bank of cache. A second, faster technique can be used to invalidate an entire cache bank directly. This second technique directly invalidates 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 cache entries directly.

For invalidating the complete instruction cache, a third method is available. By clearing the `IMC` bit in the `IMEM_CONTROL` register (see [Figure 3-2 on page 3-10](#)), 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 then configures the instruction memory as cache again. An `SSYNC` should be run before invalidating the cache and a `CSYNC` 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_DATA_x 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 3-6 on page 3-25](#).

The ITEST registers are described in [Table 3-3](#).

Table 3-3. ITEST Registers

Name	Description/ Refer to
ITEST_COMMAND	Instruction test command register For more information, see “ITEST_COMMAND Register” on page 3-25.
ITEST_DATA1	Instruction test data 1 register For more information, see “ITEST_DATA1 Register” on page 3-26.
ITEST_DATA0	Instruction test data 0 register For more information, see “ITEST_DATA0 Register” on page 3-27.

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.

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)

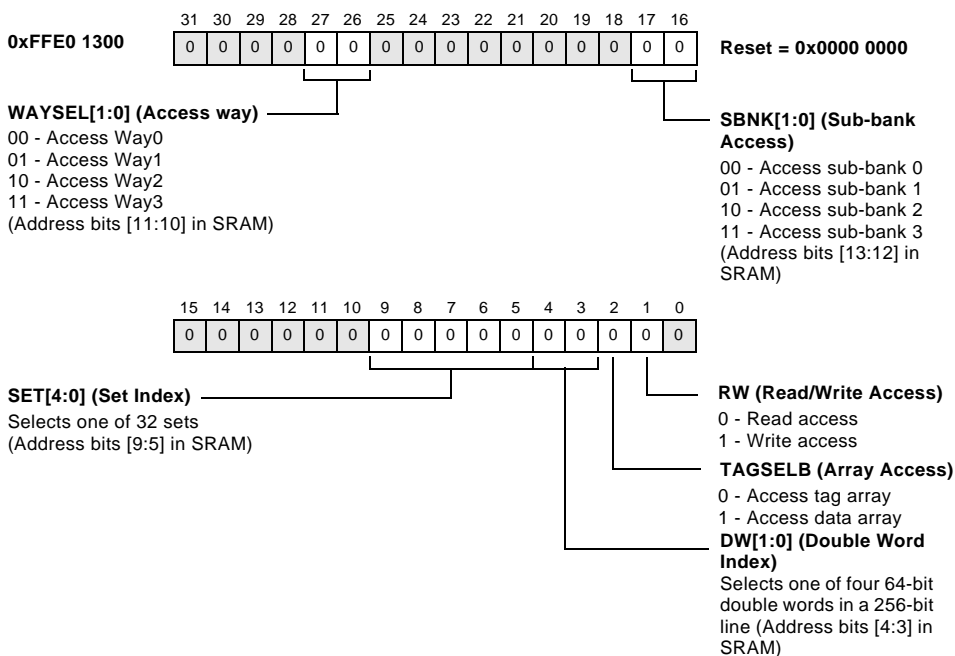


Figure 3-6. Instruction Test Command Register

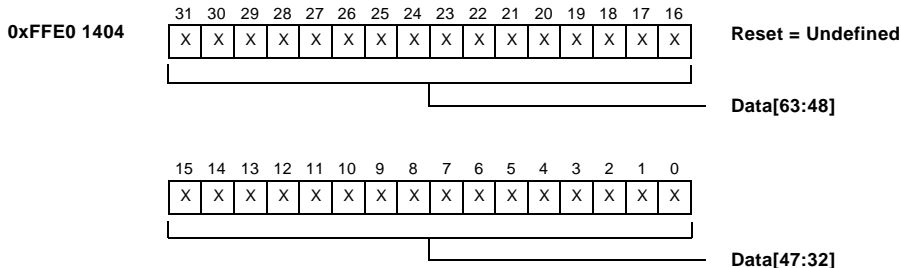
Instruction Test Registers

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 3-15](#).



When accessing tag arrays, all bits are reserved.

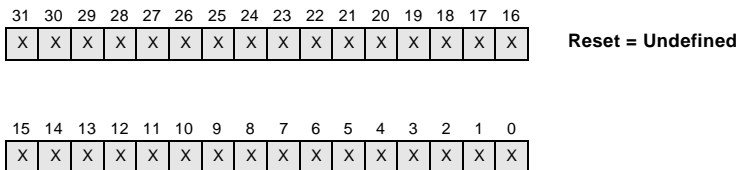


Figure 3-7. Instruction Test Data 1 Register

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 3-15](#).

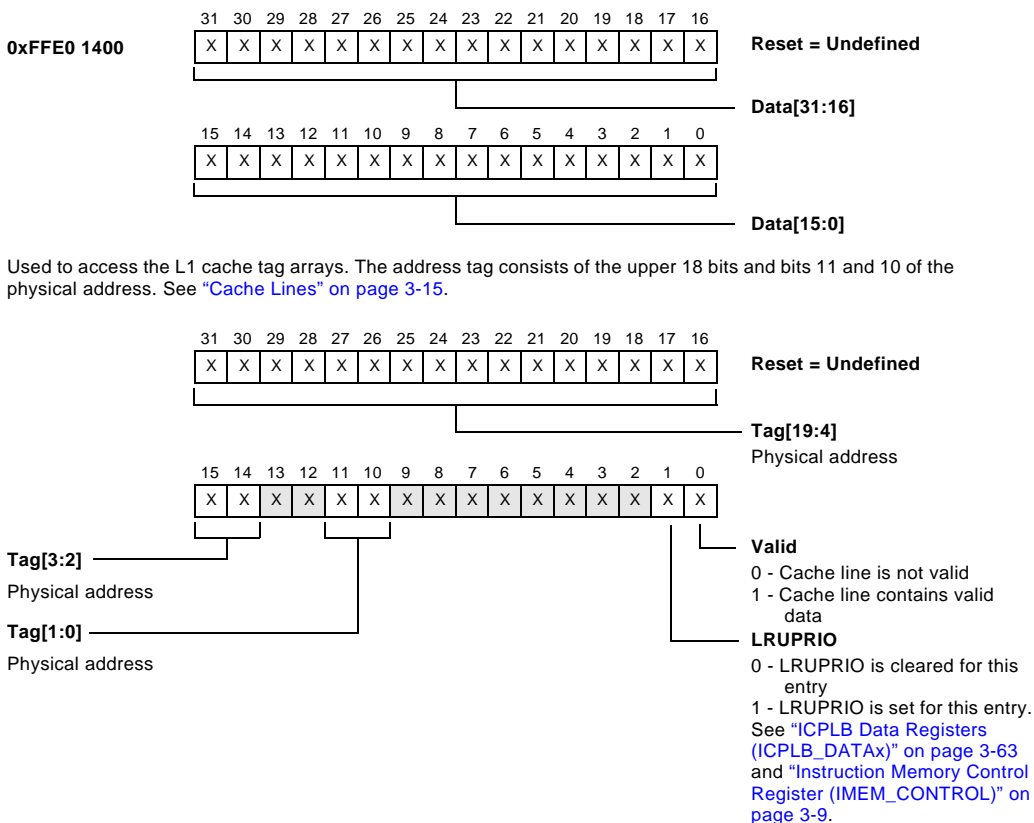


Figure 3-8. Instruction Test Data 0 Register

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 multiported 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 I/O
- One 64-bit cache fill/victim access



L1 data memory can be used only to store data.

Data Memory Control Register (DMEM_CONTROL)

The data memory control register (DMEM_CONTROL) contains control bits for the L1 data memory. See [Figure 3-9 on page 3-29](#).

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 memory other than L1 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 memory other than L1 full.

Data Memory Control Register (DMEM_CONTROL)

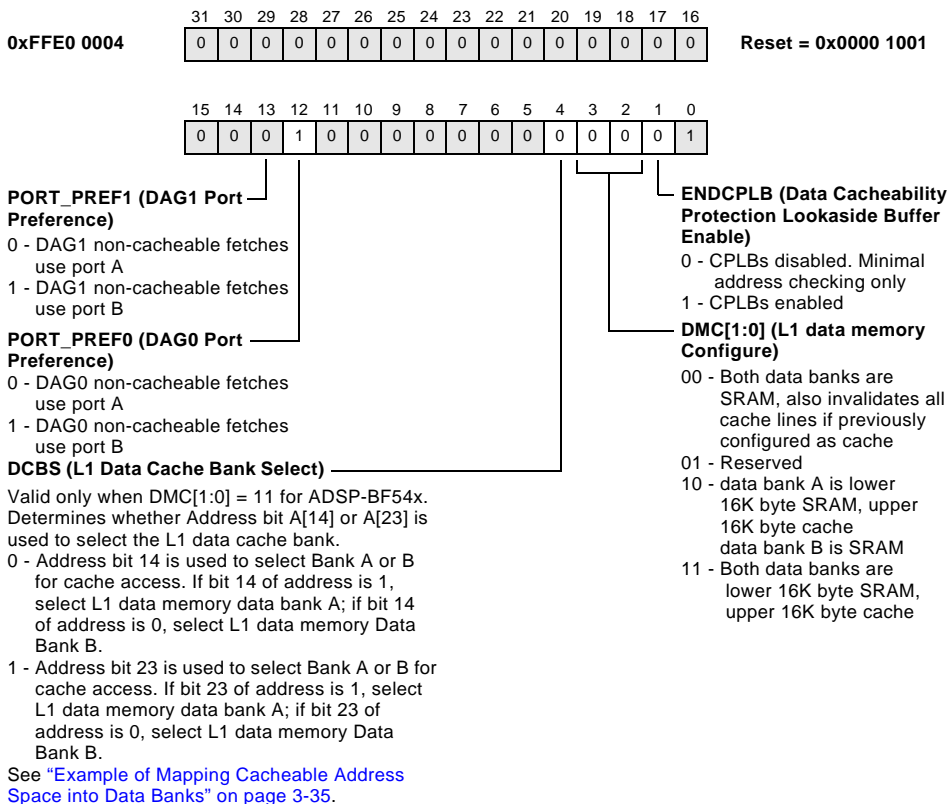


Figure 3-9. L1 Data Memory Control Register

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.


L1 Data Memory

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 3-32). 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
- Attempts to write the on-chip boot ROM


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 3-77), disabling CPLBs should be preceded by a `CSYNC` instruction, and enabling CPLBs should be followed by a `CSYNC` instruction in order to ensure predictable behavior.

 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 memory other than L1 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 they 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 is 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.

L1 Data Memory

[Table 3-4](#) shows how the subbank organization is mapped into memory.

Table 3-4. L1 Data Memory SRAM Sub-bank Start Addresses

Memory Bank and Sub-bank	Start Address
Data Bank A, Sub-bank 0	0xFF80 0000
Data Bank A, Sub-bank 1	0xFF80 1000
Data Bank A, Sub-bank 2	0xFF80 2000
Data Bank A, Sub-bank 3	0xFF80 3000
Data Bank A, Sub-bank 4	0xFF80 4000
Data Bank A, Sub-bank 5	0xFF80 5000
Data Bank A, Sub-bank 6	0xFF80 6000
Data Bank A, Sub-bank 7	0xFF80 7000
Data Bank B, Sub-bank 0	0xFF90 0000
Data Bank B, Sub-bank 1	0xFF90 1000
Data Bank B, Sub-bank 2	0xFF90 2000
Data Bank B, Sub-bank 3	0xFF90 3000
Data Bank B, Sub-bank 4	0xFF90 4000
Data Bank B, Sub-bank 5	0xFF90 5000
Data Bank B, Sub-bank 6	0xFF90 6000
Data Bank B, Sub-bank 7	0xFF90 7000

[Figure 3-10 on page 3-33](#) shows the L1 data memory architecture.

L1 Data Cache

For definitions of cache terminology, see [“Terminology” on page 3-84](#).

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.

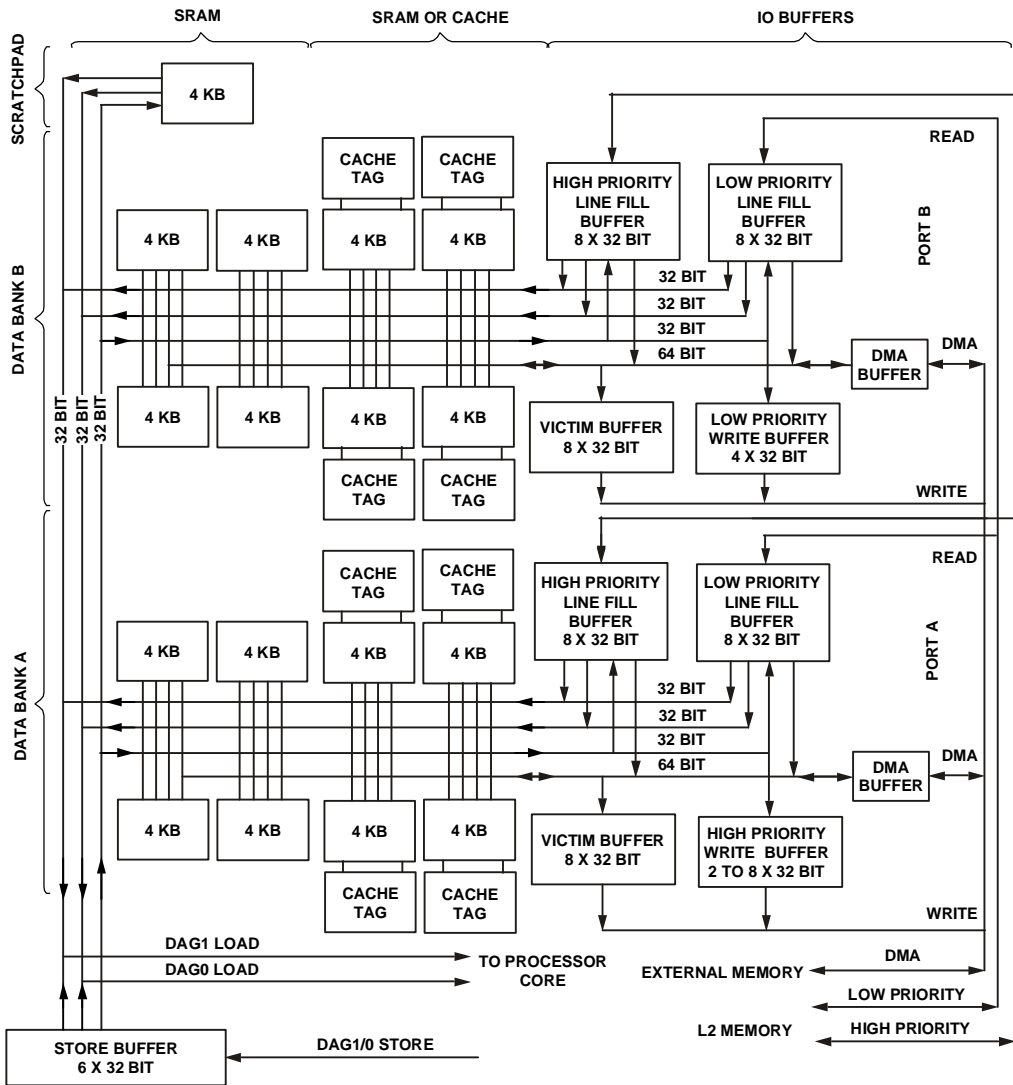


Figure 3-10. L1 Data Memory Architecture for ADSP-BF54x

L1 Data Memory

For the ADSP-BF54x processor, the upper 16K byte is 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 (address bits 13 and 12 match), the same half bank (address bits 16 match), and to the same bank (address bits 21 and 20 match). Cache has less apparent multiported behavior 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 3-54](#)). 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 are cached. The default behavior is no caching when data CPLBs are disabled.

- ⊘ 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 into Data Banks

An example of how the cacheable address space maps into two data banks follows.

When both banks are configured as cache on the ADSP-BF54x processor, they operate as two independent, 16 Kbyte, 2-way set associative caches that can be independently mapped into the Blackfin processor address space.

L1 Data Memory

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 A. All addresses in which `A[14] = 1` use data bank B.

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 `DCBS = 1`, then `A[23]` is part of the address index, and all addresses where `A[23] = 0` use data bank A. All addresses where `A[23] = 1` use data bank B.

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 $DCBS = 0$ or $DCBS = 1$ is:

- If $DCBS = 0$, $A[14]$ 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.

Consequently:

Any data in the first 16K byte of memory could be stored only in data bank A.

Any data in the next address range (16K byte through 32K byte) – 1 could be stored only in data bank B.

Any data in the next range (32K byte through 48K byte) – 1 would be stored in data bank A.

Alternate mapping would continue.

As a result, the cache operates as if it were 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).


- If $DCBS = 1$, $A[23]$ selects data bank A instead of data bank B.

With $DCBS = 1$, 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 A caches all data accesses for the first 8M byte of

L1 Data Memory

memory address range. That is, every 8M byte of range vies for the two line entries (rather than every 16K byte repeat). Likewise, data bank B caches data located above 8M byte and below 16M byte.

For example, if the application is working from a data set that is 1 Mbyte 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 A (a 2-way set associative 16K byte cache). In this instance, the application never derives any benefit from data bank B.


 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 8 Mbyte 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 3-11 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.

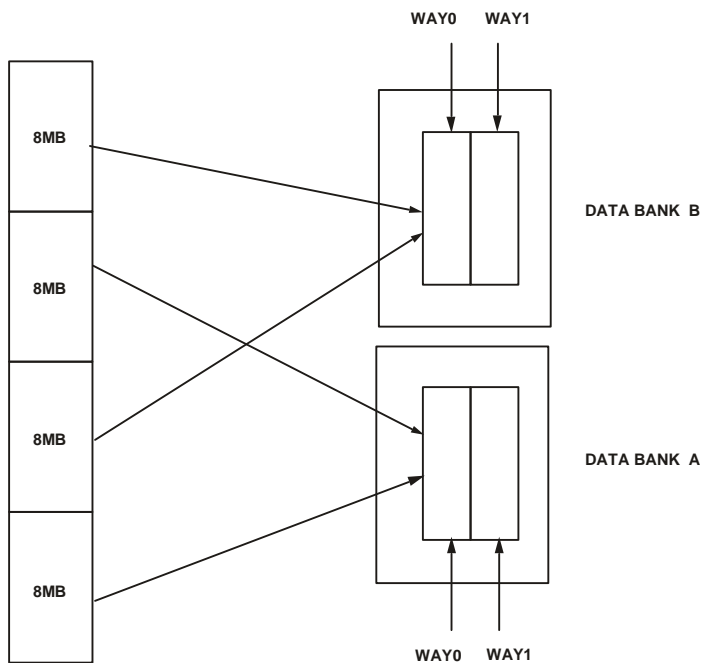


Figure 3-11. Data Cache Mapping When DCBS = 1

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 through 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 memory other than L1.

L1 Data 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:

- 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 memory other than L1 before 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 memory other than L1. 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 `PREFETCH` instruction that misses the cache (and is to a cacheable region), the instruction pipeline incurs a minimum of a four-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 [“System Interrupts” on page 4-1](#).

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 memory other than L1 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 memory other than L1.

- A write-back cache does not write to memory other than L1 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.

Write Buffers

Two separate write buffers are provided. These buffers allow stores to slow external memory to continue without causing stores to higher-speed on-chip memory other than L1 to stall. Which buffer is used is determined by the `CPLB_MEMLEV` bit in the data memory page's CPLBs. See [“Memory Protection and Properties” on page 3-54](#).

These two write buffers in the L1 data memory accept all stores with each cache inhibited or store-through protection.



An `SSYNC` instruction flushes the write buffers.

L1 Data Memory

Interrupt Priority Register (IPRIO) and Write Buffer Depth

The interrupt priority register (IPRIO) can be used to control the size of the high priority write buffer on port A (see “L1 Data Memory Architecture for ADSP-BF54x” on page 3-33).

Interrupt Priority Register (IPRIO)

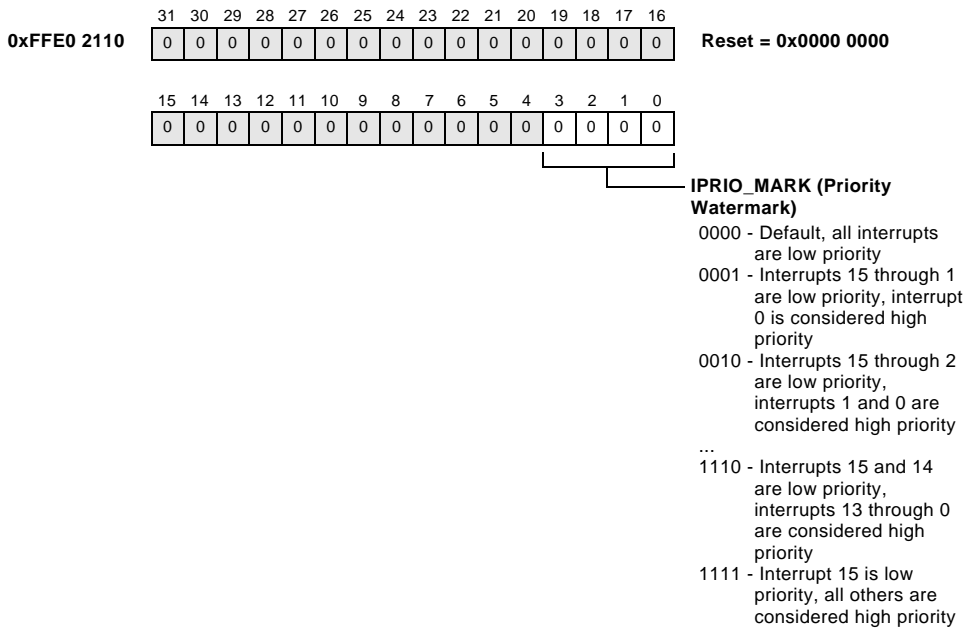


Figure 3-12. Interrupt Priority Register

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 low priority 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 low

priority 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 low priority 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`.
- `FLUSH` (data cache flush) causes the data cache to synchronize the specified cache line with memory other than L1. 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 memory other than L1. 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 Test Registers

Data Cache Invalidation

Besides the `FLUSHINV` instruction, 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.

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 3-9 on page 3-29](#)), all valid bits in the data cache are set to the invalid state. A second write to the `DMEM_CONTROL` register sets 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 accessed, the `DTEST_DATA[0]` register is used.



A `CSYNC` instruction is required after writing the `DTEST_COMMAND` MMR.

The DTEST registers are described in the following subsections.

Table 3-5. DTEST Registers

Name	Description/ Refer to
DTEST_COMMAND	Data test command register For more information, see “Data Test Command Register (DTEST_COMMAND)” on page 3-45.
DTEST_DATA1	Data test data 1 register For more information, see “Data Test Data 1 Register (DTEST_DATA1)” on page 3-47.
DTEST_DATA0	Data test data 0 register For more information, see “Data Test Data 0 Register (DTEST_DATA0)” on page 3-47.

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 Command Register (DTEST_COMMAND)

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 by way of the DTEST_COMMAND MMR to L1 instruction SRAM. Note that L1 instruction ROM is not directly accessible. Instruction ROM is accessible only through instruction fetches or DMA accesses.

Data Test Registers

Data Test Command Register (DTEST_COMMAND)

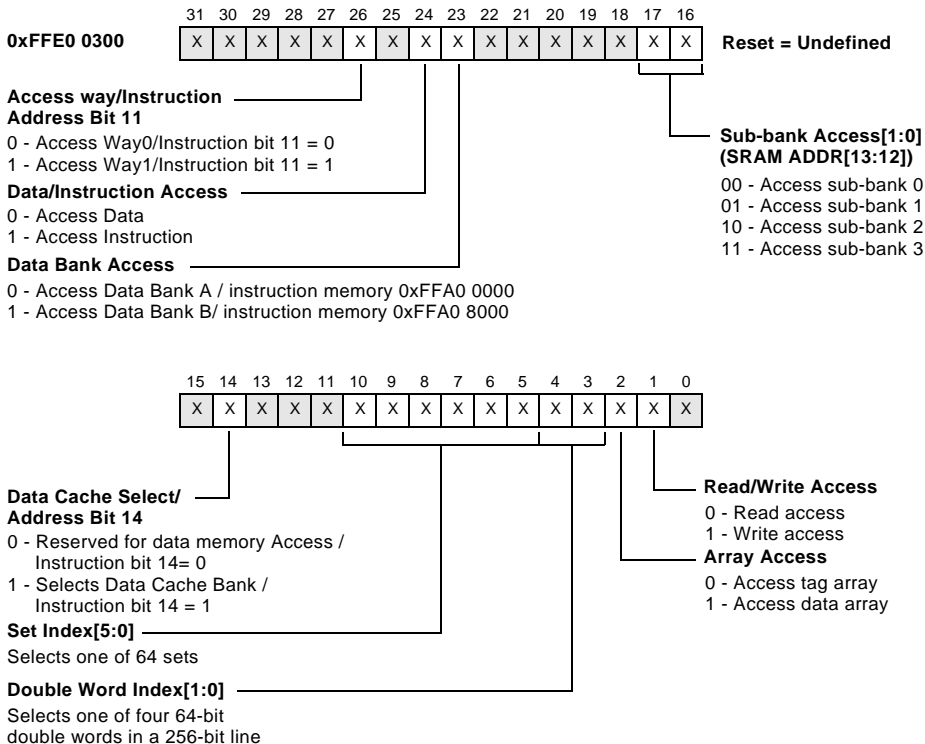
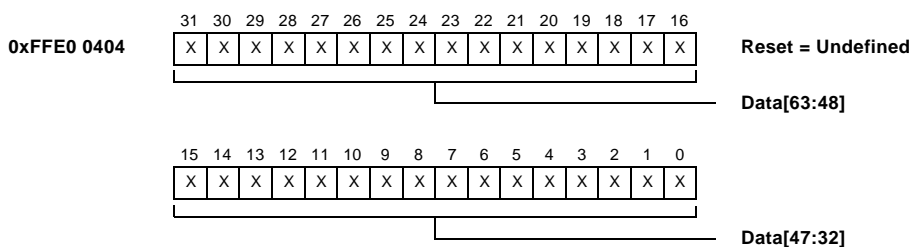


Figure 3-13. Data Test Command Register

Data Test Data 1 Register (DTEST_DATA1)

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.

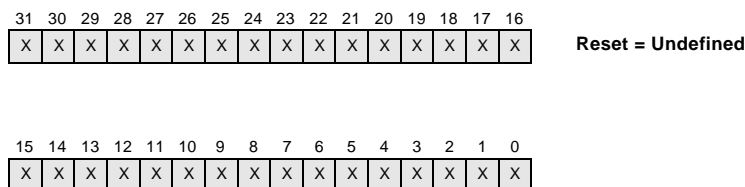


Figure 3-14. Data Test Data 1 Register

Data Test Data 0 Register (DTEST_DATA0)

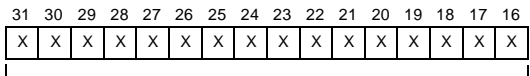
The data test data 0 register (DTEST_DATA0) stores the lower 32 bits of the 64-bit data to be written, or it contains the lower 32 bits of the destination for the 64-bit data read.

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 Registers

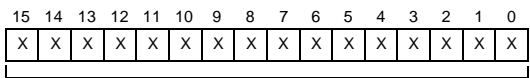
Data Test Data 0 Register (DTEST_DATA0)

0xFFE0 0400



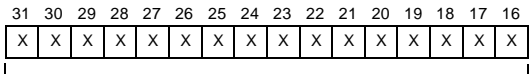
Reset = Undefined

Data[31:16]



Data[15:0]

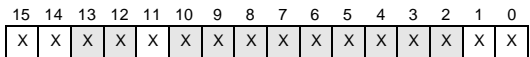
Used to access the L1 cache tag arrays. The address tag consists of the upper 18 bits and bit 11 of the physical address. See "Cache Lines" on page 3-15.



Reset = Undefined

Tag[19:4]

Physical address



Tag[3:2]

Physical address

Tag

Physical address

LRU

0 - Way0 is the least recently used
1 - Way1 is the least recently used

Valid

0 - Cache line invalid
1 - Cache line valid

Dirty

0 - Cache line unmodified since it was copied from source memory
1 - Cache line modified after it was copied from source memory

Figure 3-15. Data Test Data 0 Register

On-Chip Level 2 (L2) Memory

Configured as SRAM, the on-chip Level 2 (L2) memory of the ADSP-BF54x processor provides 128 Kbyte of low latency, high bandwidth storage capacity. For systems that use some ADSP-BF54x processor L1 memory as cache, the on-chip L2 SRAM memory system can help provide deterministic, bounded memory access times.

Simultaneous access to the multi-banked, on-chip memory other than L1 architecture from the cores and system DMA can occur in parallel, provided they access different banks. A fixed-priority arbitration scheme resolves conflicts. The on-chip system DMA controllers share a dedicated 32-bit data path into the memory other than L1 system. This interface operates at `SCLK` frequency. Dedicated L2 access from the processor core is also supported.

The processor core has a dedicated, low latency, 64-bit data path into the L2 SRAM memory. At a core clock frequency of 600 MHz, the peak data transfer rate across this interface is 4.8G byte/second.

On-Chip L2 Bank Access

The L2 is divided into eight separate 16K sub-banks. Two L2 access ports, a processor core port and a system port, are provided to allow concurrent access to the L2, provided the two ports access different memory sub-banks. If simultaneous access to the same memory sub-bank is attempted, collision detection logic in the L2 provides arbitration. This is a fixed priority arbiter; the DMA port always has the highest priority, unless the core is granted access to the sub-bank for a burst transfer. In this case, the L2 finishes the burst transfer before the system bus is granted access.

Latency

When cache is enabled, the bus between the core and L2 is fully pipelined for contiguous burst transfers. The cache line fill from on-chip memory behaves the same for instruction and data fetches. Operations that miss the cache trigger a cache line replacement. This replacement fills one 256-bit (32-byte) line with four 64-bit reads. Under this condition, the L1 cache line fills from the L2 SRAM in $9+2+2+2=15$ cycles. In other words, after nine core cycles, the first 64-bit (8-byte) fill is available for the processor. Figure 3-16 shows an example of L2 latency with cache on.

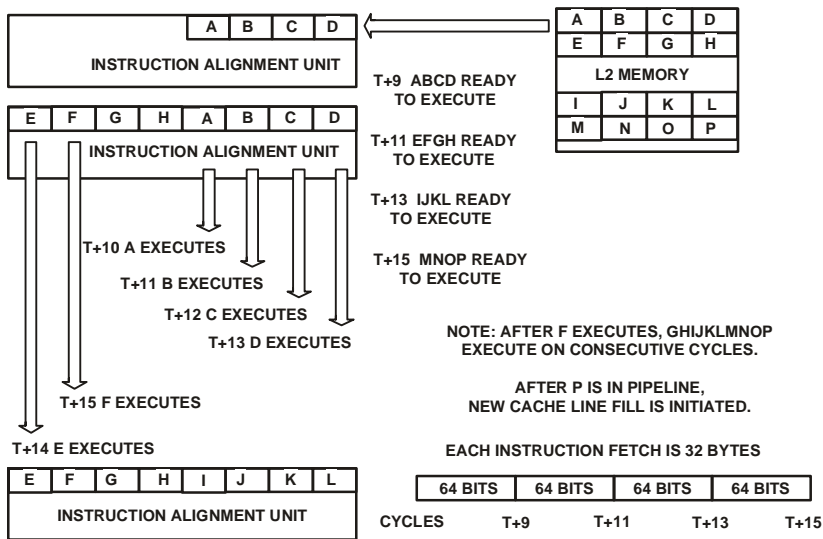


Figure 3-16. L2 Latency With Cache On

In this example, at the end of 15 core cycles, 32 bytes of instructions or data have been brought into cache and are available to the sequencer. If all the instructions contain 16 bits, sixteen instructions are brought into cache at the end of 15 cycles. In addition, the first instruction that is part

of the cache line fill executes on the tenth cycle; the second instruction executes on the eleventh cycle, and the third instruction executes on the twelfth cycle—all of them in parallel with the cache line fill.

Each cache line fill is aligned on a 32-byte boundary. When the requested instruction or data is not 32-byte aligned, the requested item is always loaded in the first read; each read is forwarded to the core as the line is filled. Sequential memory accesses miss the cache only when they reach the end of a cache line.

When on-chip L2 is configured as non-cacheable, instruction fetches and data fetches occur in 64-bit fills. In this case, each fill takes seven core cycles to complete. As shown in [Figure 3-17 on page 3-52](#), on-chip L2 is configured as non-cacheable. To illustrate the concept of L2 latency with cache off, simple instructions are used that do not require additional external data fetches. In this case, consecutive instructions are issued on consecutive cycles if multiple instructions are brought into the core in a given fetch.

On-Chip Level 2 (L2) Memory

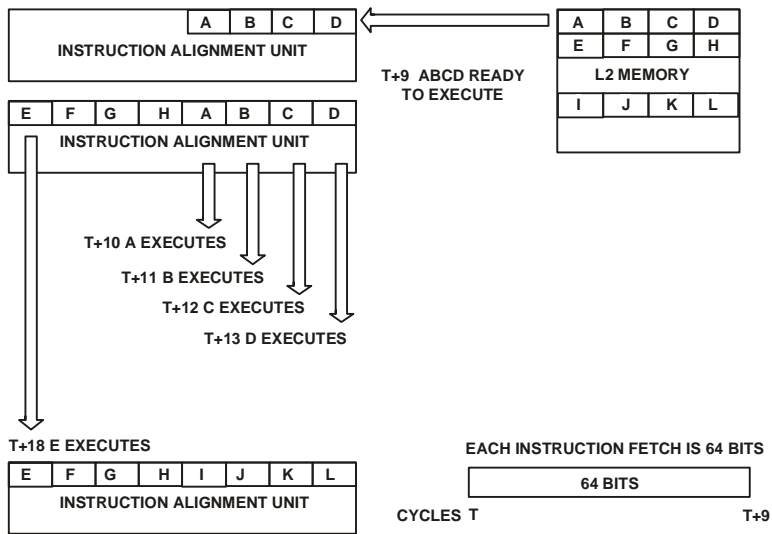


Figure 3-17. L2 Latency With Cache Off

One Time Programmable Memory

The ADSP-BF54x processor also includes an on-chip OTP memory array which provides 64K bits of non-volatile memory that can be programmed by the customer only one time. It includes the array and logic to support read access and programming. A mechanism for error correction is provided. Additionally, its pages can be write protected. The OTP is not part of the Blackfin processor linear memory map. OTP memory is not accessed directly using the Blackfin processor memory map, rather, it is accessed through four 32-bit wide registers (OTP_DATA0-3) which act as the OTP memory read/write buffer.

Because OTP memory usage is required for usage of the security features of the ADSP-BF54x processor, OTP memory is described in [Chapter 16, “One-Time Programmable Memory.”](#) Note that OTP memory has many other uses besides support for security.

External Memory

The external memory space is shown in [Figure 3-1 on page 3-4](#). One of the memory regions is dedicated to two banks of SDRAM support. The size of each SDRAM bank is programmable and can range in size from 16M byte to 256M byte. The start address of the bank is 0x0000 0000.

Each of the next four banks contains 64M byte and is dedicated to support asynchronous memories. The start address of the asynchronous memory bank is 0x2000 0000.

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 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 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 3-2 on page 3-10](#) and [Figure 3-9 on page 3-29](#).


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

Memory Protection and Properties

- 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.
- 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 memory in this page 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 can fit into the available on-chip CPLB MMRs. When this happens, a memory-based

Memory Protection and Properties

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 look aside 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 “[System Interrupts](#)” on page 4-1 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 `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 16 CPLBs must be disabled using:

- The enable DCPLB (ENDCPLB) bit in the DMEM_CONTROL register for data descriptors, or
- The enable ICPLB (ENICPLB) bit in the IMEM_CONTROL register for instruction descriptors

The CPLB replacement policy and algorithm 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 3-18](#), 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.

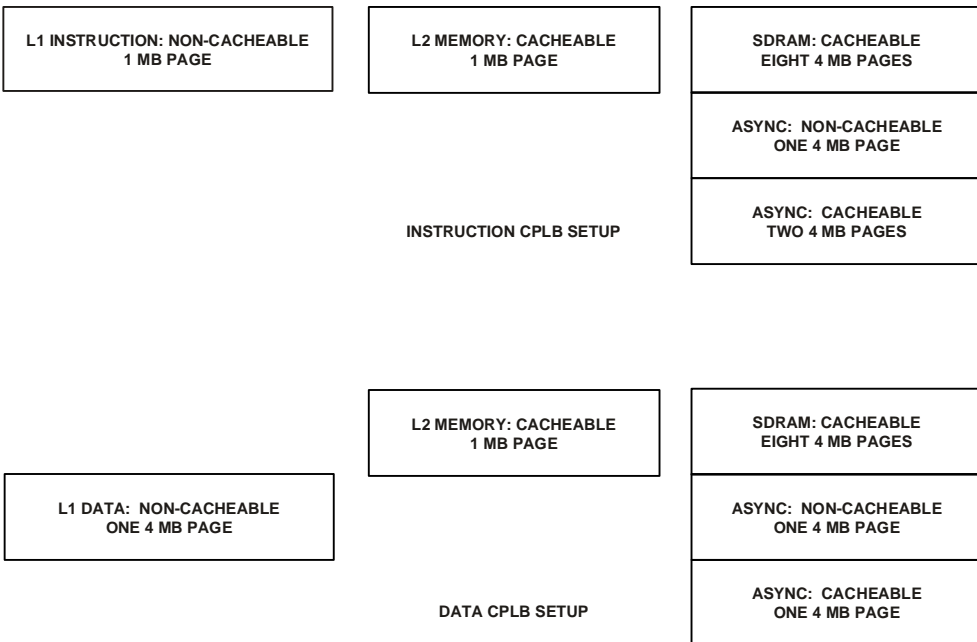


Figure 3-18. Examples of Protected Memory Regions

ICPLB Data Registers (ICPLB_DATAx)

Figure 3-19 describes the ICPLB data registers. Table 3-6 lists the ICPLB data register memory-mapped addresses.

i To ensure proper behavior and future compatibility, all reserved bits in this register must be set to 0 whenever this register is written.

ICPLB Data Registers (ICPLB_DATAx)

For memory-mapped addresses, see Table 3-6.

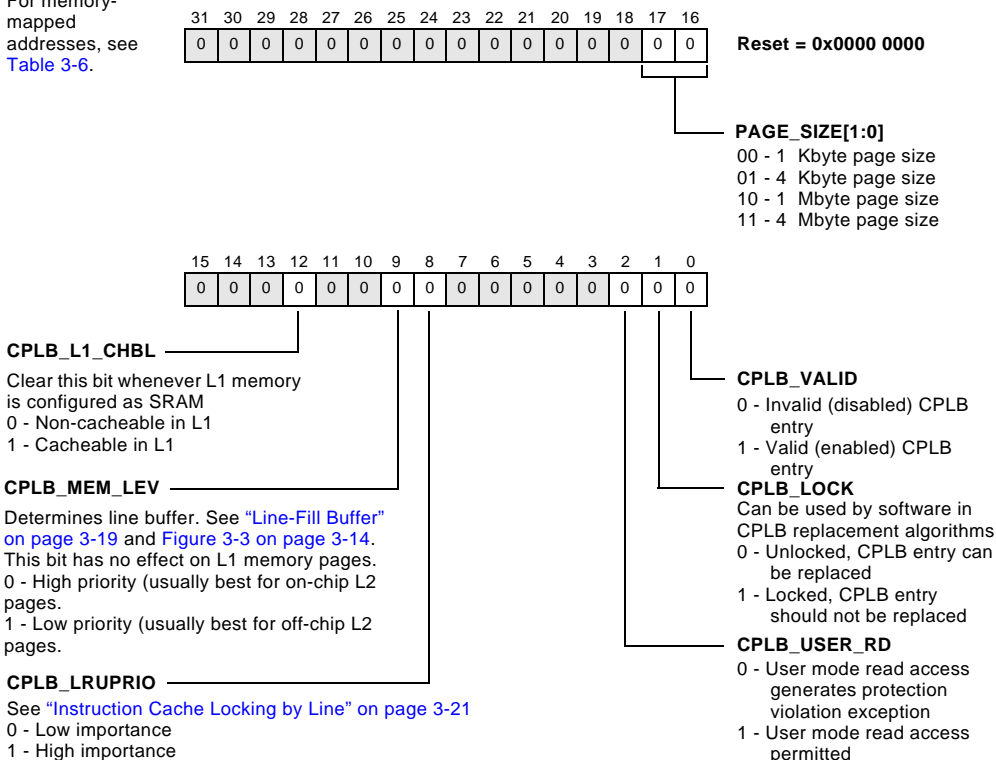


Figure 3-19. ICPLB Data Registers

Memory Protection and Properties

Table 3-6. ICPLB Data Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
ICPLB_DATA0	0xFFE0 1200
ICPLB_DATA1	0xFFE0 1204
ICPLB_DATA2	0xFFE0 1208
ICPLB_DATA3	0xFFE0 120C
ICPLB_DATA4	0xFFE0 1210
ICPLB_DATA5	0xFFE0 1214
ICPLB_DATA6	0xFFE0 1218
ICPLB_DATA7	0xFFE0 121C
ICPLB_DATA8	0xFFE0 1220
ICPLB_DATA9	0xFFE0 1224
ICPLB_DATA10	0xFFE0 1228
ICPLB_DATA11	0xFFE0 122C
ICPLB_DATA12	0xFFE0 1230
ICPLB_DATA13	0xFFE0 1234
ICPLB_DATA14	0xFFE0 1238
ICPLB_DATA15	0xFFE0 123C

DCPLB Data Registers (DCPLB_DATAx)

Figure 3-20 shows the DCPLB data registers. Table 3-7 lists the DCPLB data register memory-mapped addresses.

DCPLB Data Registers (DCPLB_DATAx)

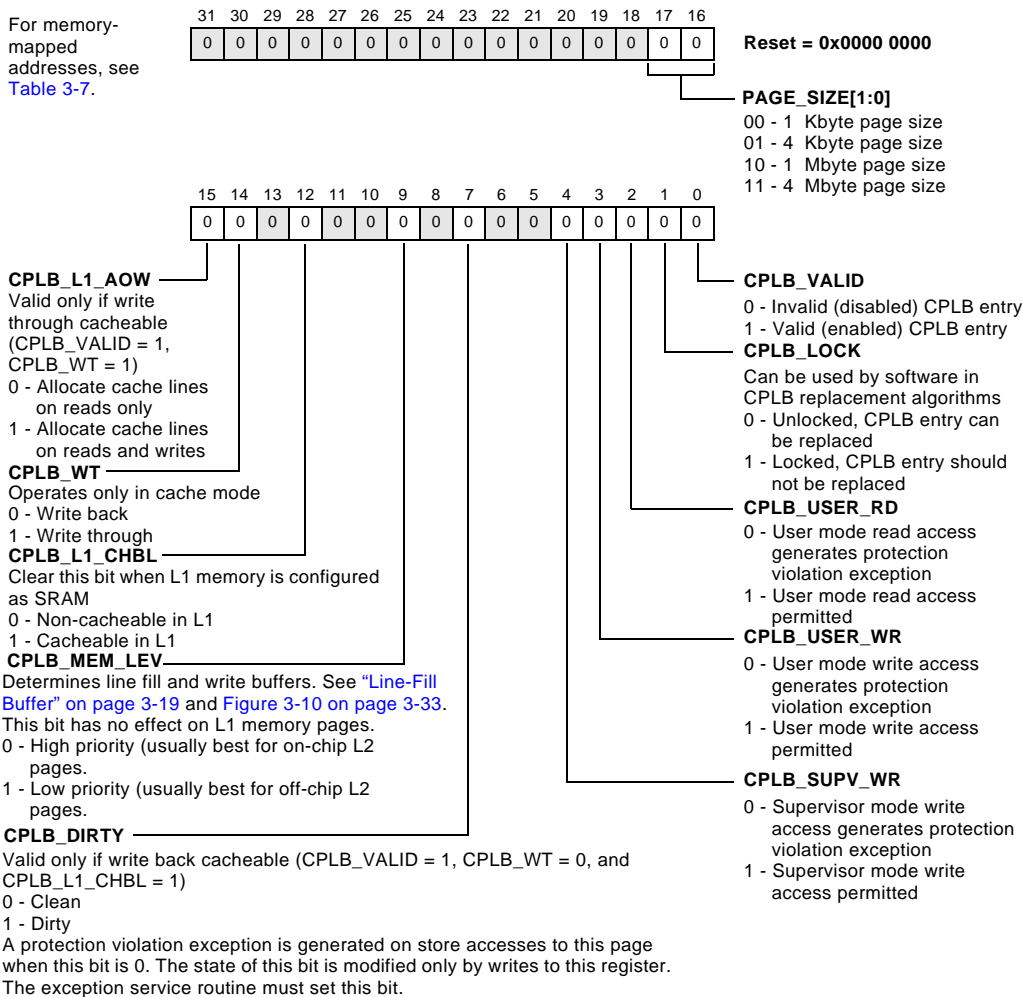


Figure 3-20. DCPLB Data Registers

Memory Protection and Properties



To ensure proper behavior and future compatibility, all reserved bits in this register must be set to 0 whenever this register is written.

Table 3-7. DCPLB Data Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DCPLB_DATA0	0xFFE0 0200
DCPLB_DATA1	0xFFE0 0204
DCPLB_DATA2	0xFFE0 0208
DCPLB_DATA3	0xFFE0 020C
DCPLB_DATA4	0xFFE0 0210
DCPLB_DATA5	0xFFE0 0214
DCPLB_DATA6	0xFFE0 0218
DCPLB_DATA7	0xFFE0 021C
DCPLB_DATA8	0xFFE0 0220
DCPLB_DATA9	0xFFE0 0224
DCPLB_DATA10	0xFFE0 0228
DCPLB_DATA11	0xFFE0 022C
DCPLB_DATA12	0xFFE0 0230
DCPLB_DATA13	0xFFE0 0234
DCPLB_DATA14	0xFFE0 0238
DCPLB_DATA15	0xFFE0 023C

DCPLB Address Registers (DCPLB_ADDRx)

Figure 3-21 shows the DCPLB address registers. Table 3-8 lists the DCPLB address register memory-mapped addresses.

DCPLB Address Registers (DCPLB_ADDRx)

For memory-mapped addresses, see Table 3-8.

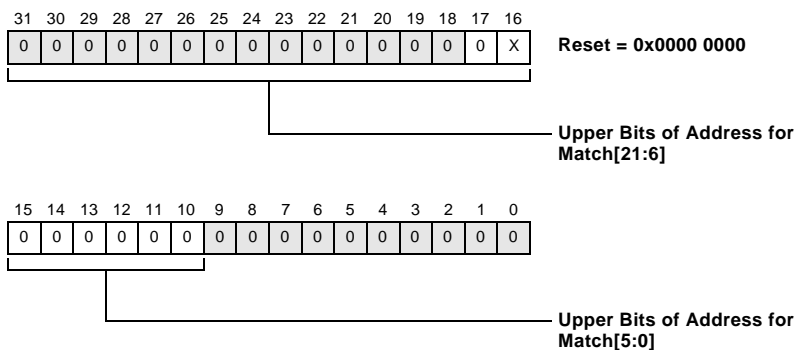


Figure 3-21. DCPLB Address Registers

Memory Protection and Properties

Table 3-8. DCPLB Address Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DCPLB_ADDR0	0xFFE0 0100
DCPLB_ADDR1	0xFFE0 0104
DCPLB_ADDR2	0xFFE0 0108
DCPLB_ADDR3	0xFFE0 010C
DCPLB_ADDR4	0xFFE0 0110
DCPLB_ADDR5	0xFFE0 0114
DCPLB_ADDR6	0xFFE0 0118
DCPLB_ADDR7	0xFFE0 011C
DCPLB_ADDR8	0xFFE0 0120
DCPLB_ADDR9	0xFFE0 0124
DCPLB_ADDR10	0xFFE0 0128
DCPLB_ADDR11	0xFFE0 012C
DCPLB_ADDR12	0xFFE0 0130
DCPLB_ADDR13	0xFFE0 0134
DCPLB_ADDR14	0xFFE0 0138
DCPLB_ADDR15	0xFFE0 013C

ICPLB Address Registers (ICPLB_ADDRx)

Figure 3-22 shows the ICPLB address registers. Table 3-9 lists the ICPLB address register memory-mapped addresses.

ICPLB Address Registers (ICPLB_ADDRx)

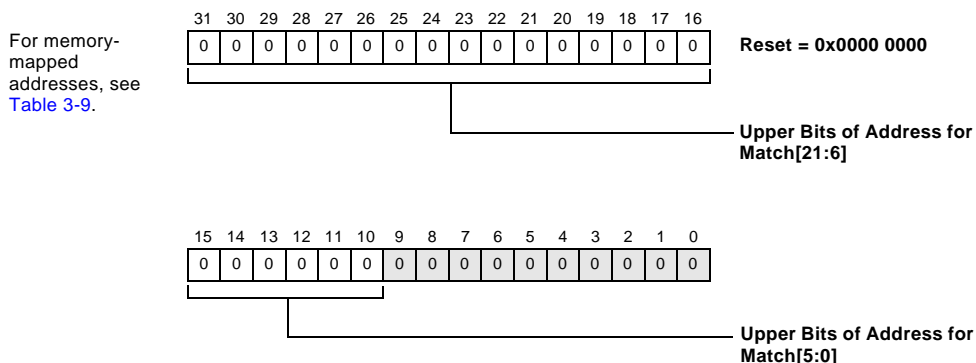


Figure 3-22. ICPLB Address Registers

Memory Protection and Properties

Table 3-9. ICPLB Address Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
ICPLB_ADDR0	0xFFE0 1100
ICPLB_ADDR1	0xFFE0 1104
ICPLB_ADDR2	0xFFE0 1108
ICPLB_ADDR3	0xFFE0 110C
ICPLB_ADDR4	0xFFE0 1110
ICPLB_ADDR5	0xFFE0 1114
ICPLB_ADDR6	0xFFE0 1118
ICPLB_ADDR7	0xFFE0 111C
ICPLB_ADDR8	0xFFE0 1120
ICPLB_ADDR9	0xFFE0 1124
ICPLB_ADDR10	0xFFE0 1128
ICPLB_ADDR11	0xFFE0 112C
ICPLB_ADDR12	0xFFE0 1130
ICPLB_ADDR13	0xFFE0 1134
ICPLB_ADDR14	0xFFE0 1138
ICPLB_ADDR15	0xFFE0 113C

CPLB Status Registers

Bits in the DCPLB status register (DCPLB_STATUS) and ICPLB status register (ICPLB_STATUS) identify the CPLB entry that 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.

DCPLB Status Register (DCPLB_STATUS)

The `FAULT_DAG`, `FAULT_USERSUPV`, and `FAULT_RW` bits in the DCPLB status register (`DCPLB_STATUS`) identify the CPLB entry that triggered the CPLB-related exception (see [Figure 3-23](#)).

DCPLB Status Register (DCPLB_STATUS)

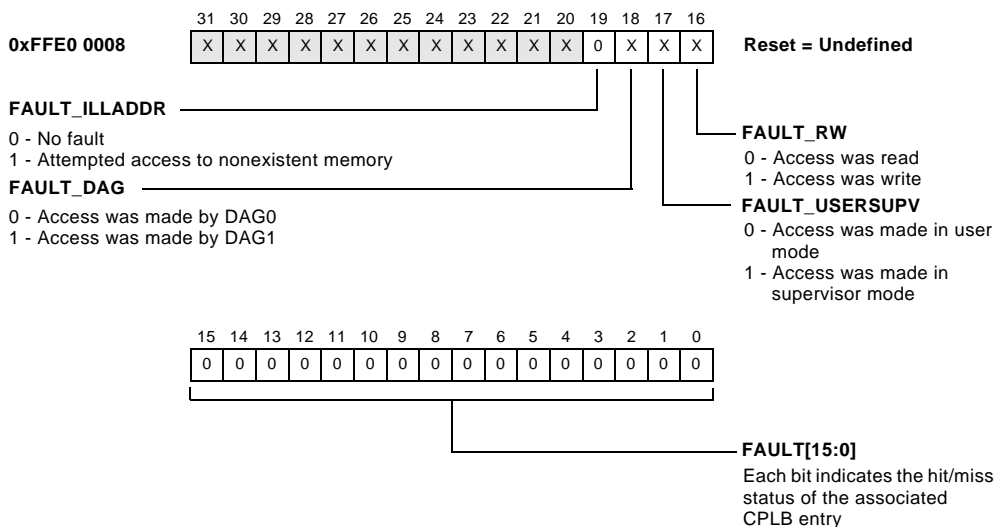


Figure 3-23. DCPLB Status Register

Memory Protection and Properties

ICPLB Status Register (ICPLB_STATUS)

The `FAULT_USERSUPV` bit in the ICPLB status register (`ICPLB_STATUS`) is used to identify the CPLB entry that triggered the CPLB-related exception (see [Figure 3-24](#)).

ICPLB Status Register (ICPLB_STATUS)

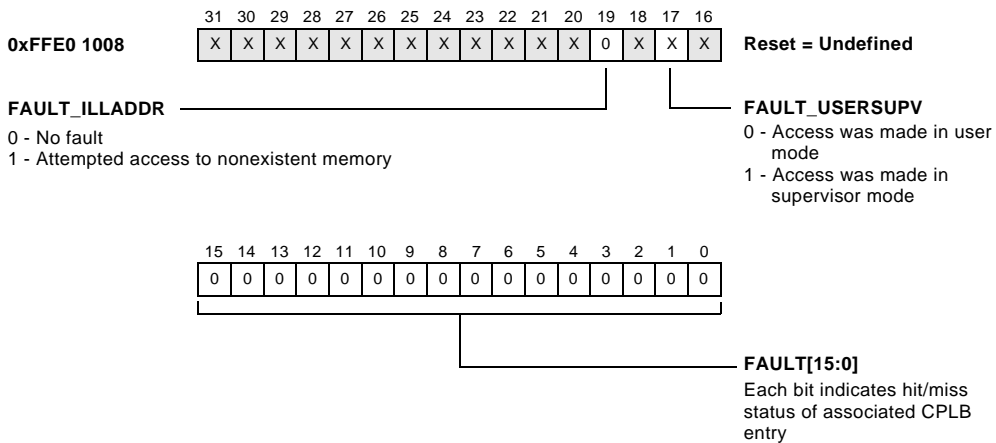


Figure 3-24. ICPLB Status Register

CPLB Fault Address Registers

The DCPLB fault address register (`DCPLB_FAULT_ADDR`) and ICPLB fault address register (`ICPLB_FAULT_ADDR`) hold the address that caused a fault in the L1 data memory or L1 instruction memory, respectively.



The `DCPLB_FAULT_ADDR` and `ICPLB_FAULT_ADDR` registers are valid only while in the faulting exception service routine.

DCPLB Fault Address Register (DCPLB_FAULT_ADDR)

Figure 3-25 lists the DCPLB fault address register.

DCPLB Address Register (DCPLB_FAULT_ADDR)

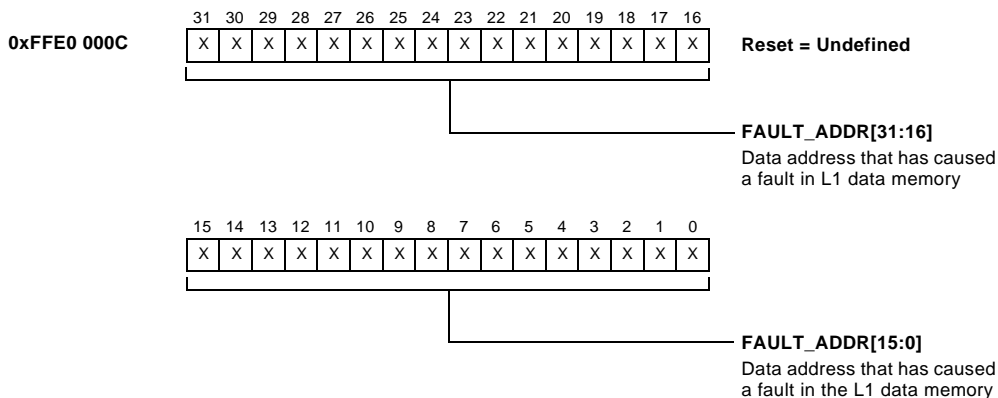


Figure 3-25. DCPLB Fault Address Register

Memory Transaction Model

ICPLB Fault Address Register (ICPLB_FAULT_ADDR)

Figure 3-26 lists the ICPLB fault address register..

ICPLB Fault Address Register (ICPLB_FAULT_ADDR)

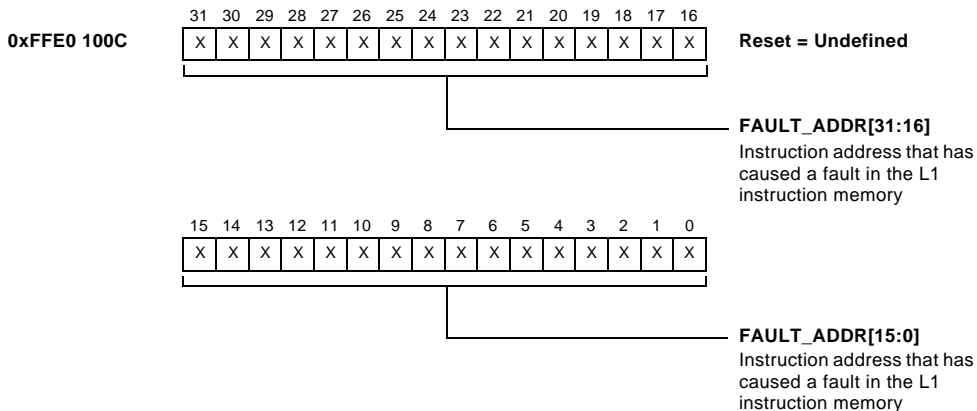


Figure 3-26. ICPLB Fault Address Register

Memory Transaction Model

Both internal and external memory locations are accessed in little endian byte order. Figure 3-27 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.

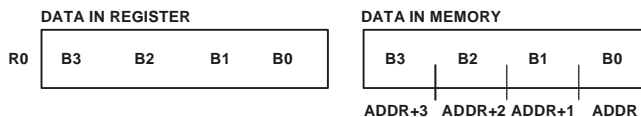


Figure 3-27. Data Stored in Little Endian Order

Figure 3-28 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 $addr$).

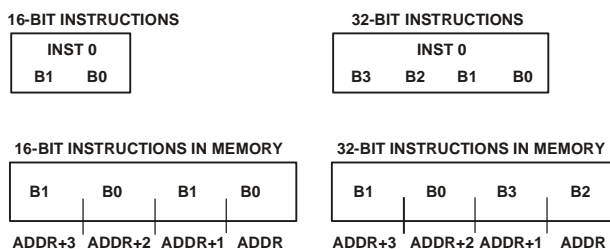


Figure 3-28. Instructions Stored in Little Endian Order

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 $addr+2$).

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.

Load/Store Operation

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. The unrelated instructions execute in parallel while the processor waits for the memory system to return the data. 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 will complete before the returned data is used by a subsequent instruction.
- Load operations using data previously written will use the updated values.
- Store operations will 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 is 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.

Load/Store Operation

When writing to or reading from nonmemory 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, `CSYNC` or `SSYNC`, respectively.

The `CSYNC` instruction ensures all pending core operations have completed and the core buffer (between the processor core and the L1 memories) is 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:

```
IF CC JUMP away_from_here
csync;
r0 = [p0];
away_from_here:
```

In the example code, the `CSYNC` instruction ensures:

- The conditional branch (`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 `CSYNC` completes.
- The load is not fetched from memory speculatively.

The `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 `CSYNC`, the `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,

```

        IF CC JUMP away_from_here
        R0 = [P2];
        ...
away_from_here:

```

Load/Store Operation

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 returns 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 [“System Interrupts” on page 4-1](#)). 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 3-43](#)), and/or explicit line invalidation through the core MMRs (see [“Data Test Registers” on page 3-44](#)).

Atomic Operations


The processor provides a single atomic operation: `TESTSET`. Atomic operations are used to provide noninterruptible 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 ensures 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 4 Gbyte 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 3-75](#).

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 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 provides a summary of all core MMRs. Appendix B 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 3-1](#) shows the instructions required to manipulate a generic core MMR.

Listing 3-1. 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 */
```

Terminology

```
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 interruptable.

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 is 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 the line is valid and the data contained in the line matches that in 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 3-4 on page 3-16](#)).

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.

Terminology

valid. A state bit, stored with the tag, indicating 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 3-4 on page 3-16](#)).

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).

4 SYSTEM INTERRUPTS

This chapter discusses the system interrupt controller (SIC), which is specific to the ADSP-BF54x processor derivatives. While this chapter does refer to features of the core event controller (CEC), it does not cover all aspects of it. Refer to the appropriate *Blackfin Processor Programming Reference* for more information on the CEC.

The chapter includes the following sections:

- [“Overview” on page 4-1](#)
- [“Interfaces” on page 4-2](#)
- [“Description of Operation” on page 4-6](#)
- [“Programming Model” on page 4-22](#)
- [“System Interrupt Controller Registers” on page 4-26](#)
- [“Programming Examples” on page 4-44](#)

Overview

This chapter describes the system peripheral interrupts, including setup and clearing of interrupt requests.

Interfaces

Features

The Blackfin processor architecture provides a two-level interrupt processing scheme:

- The core event controller (CEC) runs in the `CCLK` clock domain. It interacts closely with the program sequencer and manages the event vector table (EVT). The CEC processes not only core-related interrupts such as exceptions, core errors, and emulation events, it also supports software interrupts.
- The system interrupt controller (SIC) runs in the `SCLK` clock domain. It masks, groups, and prioritizes interrupt requests signalled by on-chip or off-chip peripherals and forwards them to the CEC.

Interfaces

[Figure 4-1](#), [Figure 4-2](#), and [Figure 4-3](#) provide an overview of how the individual peripheral interrupt request lines connect to the SIC. They also show how the 12 interrupt assignment registers (`SIC_IARx`) control the assignment to the 9 available peripheral request inputs of the CEC.



The memory-mapped `ILAT`, `IMASK`, and `IPEND` registers are part of the CEC controller.

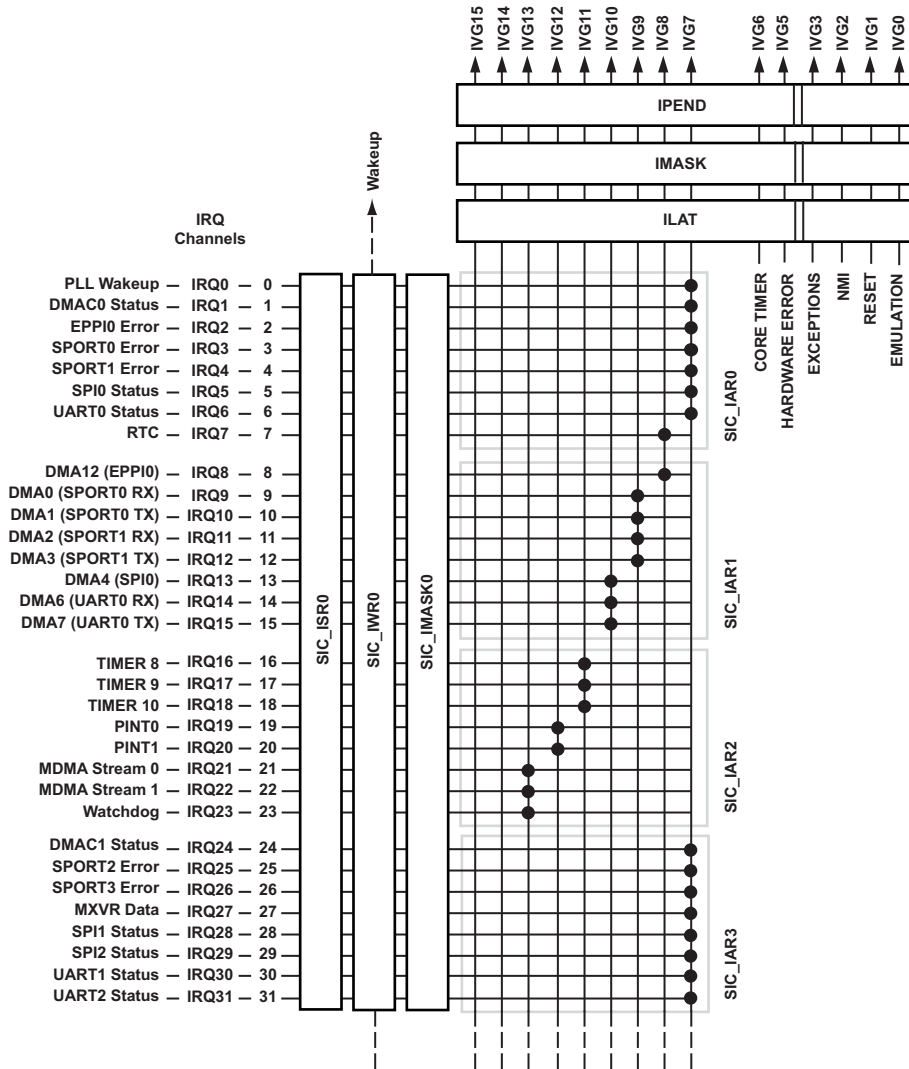


Figure 4-1. Interrupt Routing Overview-1 of 3

Interfaces

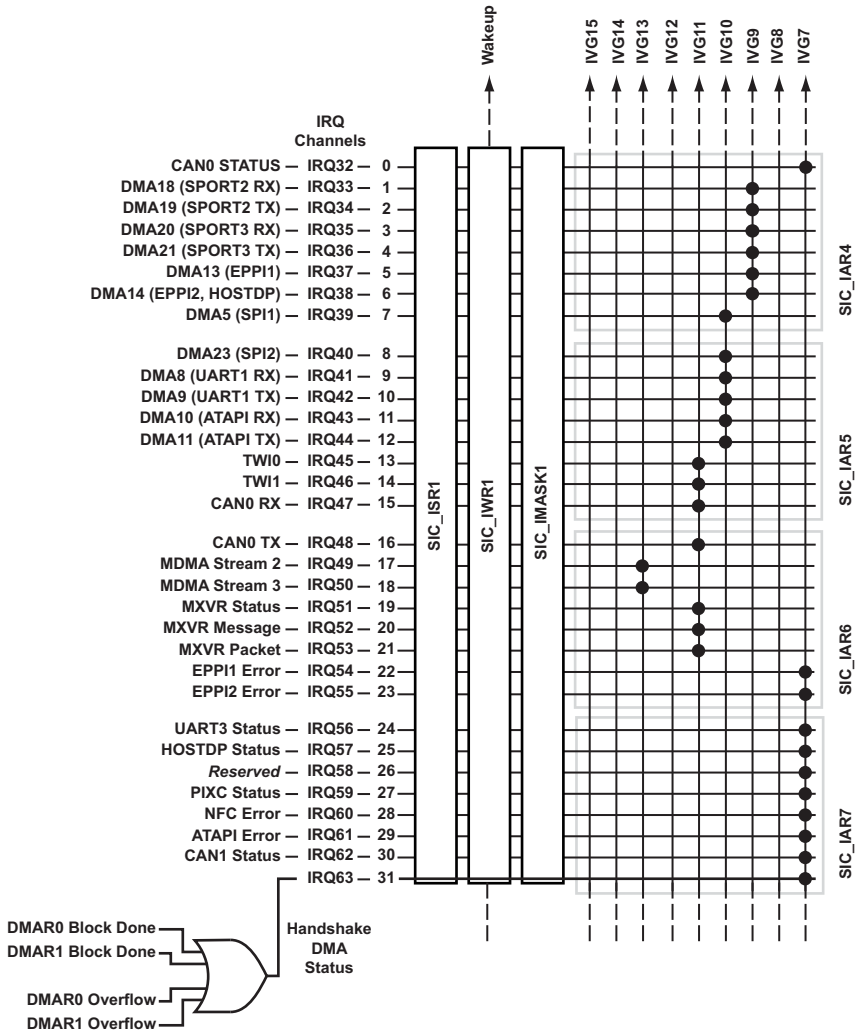


Figure 4-2. Interrupt Routing Overview-2 of 3

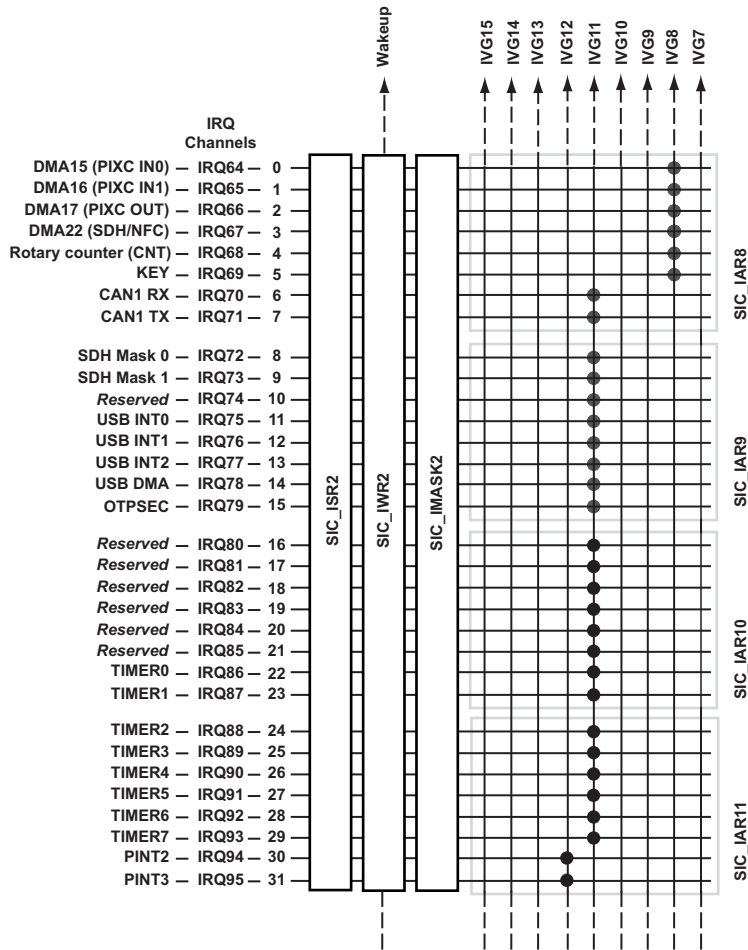


Figure 4-3. Interrupt Routing Overview-3 of 3

Description of Operation

The following sections describe the operation of the system interrupts.

Events and Sequencing

The processor employs a two-level event control mechanism. The processor SIC works with the CEC to prioritize and control all system interrupts. The SIC 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 SIC.

The CEC of the processor manages five types of activities or events:

- Emulation
- Reset
- Nonmaskable interrupts (NMI)
- Exceptions
- Interrupts

Note the word *event* describes all five types of activities. The CEC manages fifteen different events in all: emulation, reset, NMI, exception, and eleven interrupts.

An interrupt is an event that changes the 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 CEC supports nine general-purpose interrupts (IVG7 – IVG15) in addition to the dedicated interrupt and exception events that are described in [Table 4-1](#).

Table 4-1. System and Core Event Mapping

Peripheral Interrupt Source	Event Source	Core Event Name
Core events	Emulation (highest priority)	EMU
	Reset	RST
	NMI	NMI
	Exception	EVX
	Reserved	–
	Hardware error	IVHW
	Core timer	IVTMR

Description of Operation

Table 4-1. System and Core Event Mapping (Cont'd)

Peripheral Interrupt Source	Event Source	Core Event Name
System Interrupts	PLL Wakeup Interrupt DMAC0 Status (generic) DMAC1 Status (generic) EPPI0 Error Interrupt EPPI1 Error Interrupt EPPI2 Error Interrupt SPORT0 Error Interrupt SPORT1 Error Interrupt SPORT2 Error Interrupt SPORT3 Error Interrupt MXVR Synchronous Data Interrupt SPI0 Status Interrupt SPI1 Status Interrupt SPI2 Status Interrupt UART0 Status Interrupt UART1 Status Interrupt UART2 Status Interrupt UART3 Status Interrupt HOSTDP Status Interrupt PIXC Status Interrupt NFC Status Interrupt ATAPI Status Interrupt CAN0 Status Interrupt CAN1 Status Interrupt DMAR0 Block Done DMAR1 Block Done DMAR0 Overflow DMAR1 Overflow	IVG7
	Real-Time Clock Interrupt DMA12 Interrupt (EPPI0) DMA15 Interrupt (PIXC IN0) DMA16 Interrupt (PIXC IN1) DMA17 Interrupt (PIXC OUT) DMA22 Interrupt (SDH/NFC) Rotary Counter Interrupt Keypad Interrupt	IVG8

Table 4-1. System and Core Event Mapping (Cont'd)

Peripheral Interrupt Source	Event Source	Core Event Name
System Interrupts, continued	DMA0 Interrupt (SPORT0 RX) DMA1 Interrupt (SPORT0 TX) DMA2 Interrupt (SPORT1 RX) DMA3 Interrupt (SPORT1 TX) DMA18 Interrupt (SPORT2 RX) DMA19 Interrupt (SPORT2 TX) DMA20 Interrupt (SPORT3 RX) DMA21 Interrupt (SPORT3 TX) DMA13 Interrupt (EPPI1) DMA14 Interrupt (EPPI2,HOSTDP)	IVG9
	DMA4 Interrupt (SPI0) DMA6 Interrupt (UART0 RX) DMA7 Interrupt (UART0 TX) DMA5 Interrupt (SPI1) DMA23 Interrupt (SPI2) DMA8 Interrupt (UART1 RX) DMA9 Interrupt (UART1 TX) DMA10 Interrupt (ATAPI RX) DMA11 Interrupt (ATAPI TX)	IVG10
	Timer 8 Interrupt Timer 9 Interrupt Timer 10 Interrupt TWI0 Interrupt TWI1 Interrupt CAN0 RX Interrupt CAN0 TX Interrupt CAN1 RX Interrupt CAN1 TX Interrupt SDH Interrupt 0 SDH Interrupt 1 USB Interrupt 0 (USB_INT0) USB Interrupt 1 (USB_INT1) USB Interrupt 2 (USB_INT2) USB DMA Interrupt (USB_DMAINT) OTPSEC Interrupt	IVG11

Description of Operation

Table 4-1. System and Core Event Mapping (Cont'd)

Peripheral Interrupt Source	Event Source	Core Event Name
System Interrupts, continued	MXVR Asynchronous Packet Interrupt MXVR Control Message Interrupt MXVR Status Interrupt Timer 0 Interrupt Timer 1 Interrupt Timer 2 Interrupt Timer 3 Interrupt Timer 4 Interrupt Timer 5 Interrupt Timer 6 Interrupt Timer 7 Interrupt	
	Pin Interrupt 0 (PINT0) Pin Interrupt 1 (PINT1) Pin Interrupt 2 (PINT2) Pin Interrupt 3 (PINT3)	IVG12
	MDMA Stream 0 MDMA Stream 1 MDMA Stream 2 MDMA Stream 3 Software Watchdog Timer Interrupt	IVG13

It is common for applications to reserve the lowest or the two lowest priority interrupts (IVG14 and IVG15) for software interrupts, leaving eight or seven prioritized interrupt inputs (IVG7 – IVG13) for peripheral purposes. Refer to [Table 4-1](#).



The system interrupt to core event mappings shown in [Table 4-1](#) are the default values at reset and can be changed by software.

System Peripheral Interrupts

To service the rich set of peripherals, the SIC has 96 interrupt request inputs and 9 interrupt request outputs which go to the CEC. The primary function of the SIC is to mask, group, and prioritize interrupt requests and to forward them to the nine general-purpose interrupt inputs of the

CEC (IVG7–IVG15). Additionally, the SIC controller can enable individual peripheral interrupts to wake up the processor from idle or power-down state.

The nine general-purpose interrupt inputs (IVG7–IVG15) of the core event controller have fixed priority. The IVG0 channel has the highest priority and IVG15 has the lowest priority. Therefore, the interrupt assignment in the SIC_IARx registers not only groups peripheral interrupts, but it also programs their priority by assigning them to individual IVG channels. However, 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 (SIC_IARx) settings, as detailed in [Figure 4-7 on page 4-28](#) through [Figure 4-18 on page 4-33](#). 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.


The core timer has a dedicated input to the CEC controller. Its interrupts are not routed through the SIC controller and always have higher priority than requests from all other peripherals.

The system interrupt mask register (SIC_IMASKx, shown in [Figure 4-19 on page 4-35](#) through [Figure 4-21 on page 4-37](#)) allows software to mask any peripheral interrupt source at the SIC level. This functionality is independent of whether the particular interrupt is enabled at the peripheral itself. At reset, the contents of SIC_IMASKx are all 0s to mask off all peripheral interrupts. Turning off a system interrupt mask and enabling the particular interrupt is performed by writing a 1 to a bit location in the SIC_IMASKx register.

Description of Operation

The SIC includes a read-only system interrupt status register (`SIC_ISRx`) with individual bits which correspond to one of the peripheral interrupt sources. See [Figure 4-24 on page 4-40](#). When the SIC detects the interrupt, the bit is asserted. When the SIC detects that the peripheral interrupt input is deasserted, the respective bit in the system interrupt status register is cleared. Note for some peripherals, such as programmable flag asynchronous input interrupts, many cycles of latency may pass from the time an interrupt service routine initiates the clearing of the interrupt (usually by writing a system MMR) to the time the SIC senses that the interrupt is 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 the `SIC_ISRx` register 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's 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` seldom, if ever, needs to be interrogated.

The `SIC_ISRx` register is not affected by the state of the system interrupt mask register (`SIC_IMASKx`) and can be read at any time. Writes to the `SIC_ISRx` register have no effect on its contents.

Peripheral DMA channels are mapped in a fixed manner to the peripheral interrupt IDs. However, the assignment between peripherals and DMA channels is freely programmable with the `DMAx_PERIPHERAL_MAP` registers. [Table 4-2](#), [Figure 4-4](#), and [Figure 4-5](#) show the default DMA assignment. For more information on DMA, see [Chapter 5, “Direct Memory Access”](#). Once a peripheral is assigned to a DMA channel it uses the new DMA channel’s interrupt ID regardless of whether DMA is enabled or not. Therefore, clean `DMAx_PERIPHERAL_MAP` management is required even if the DMA is not used. The default setup should be the best choice for all non-DMA applications.

For dynamic power management, any of the peripherals can be configured to wake up the core from its idled state or from sleep mode to optionally process the interrupt, simply by enabling the appropriate bit in the system interrupt wakeup-enable register (`SIC_IWRx`, refer to [Figure 4-25 on page 4-41](#)). If a peripheral interrupt source is enabled in the `SIC_IWRx` register and the core is idled or in sleep mode, the interrupt causes the DPMC to initiate the core wakeup sequence in order to optionally process the interrupt. Note this mode of operation may add latency to interrupt processing, depending on the power control state. For further discussion of power modes and the idled state of the core, see [Chapter 18, “Dynamic Power Management”](#).

The `SIC_IWRx` register has no effect unless the core is idled or in sleep mode. By default, all interrupts generate a wakeup 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

Description of Operation

SIC_IWR_x register can be read from or written to at any time. To prevent spurious or lost interrupt activity, this register should be written to only when all peripheral interrupts are disabled.

i The wakeup function is independent of the interrupt mask function. If an interrupt source is enabled in the SIC_IWR_x register but masked-off in the SIC_IMASK_x register, the core wakes up if it is idled or in sleep mode, but it does not generate an interrupt.

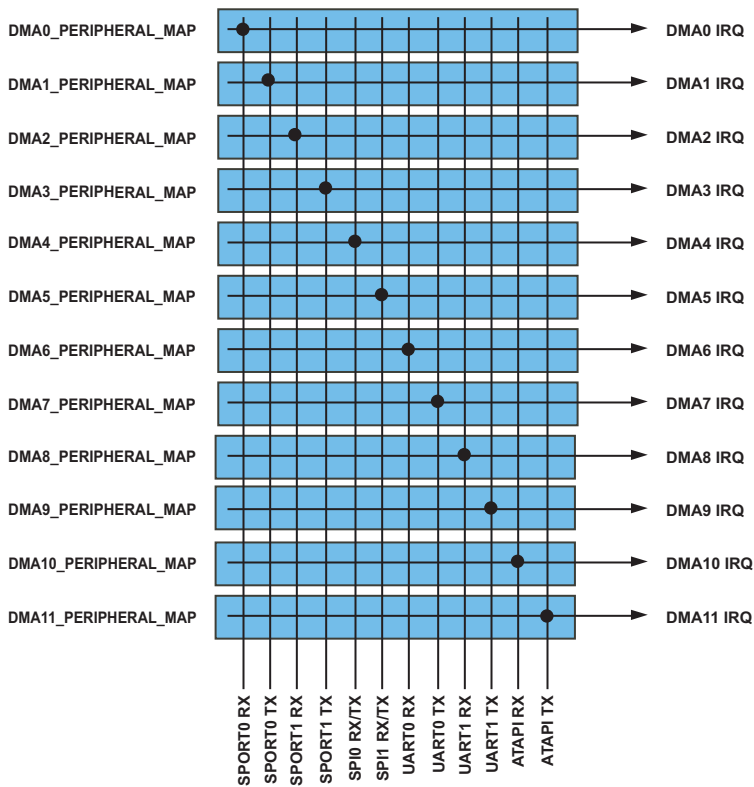


Figure 4-4. Default Peripheral-to-DMA Mapping (DMAC0 Controller)

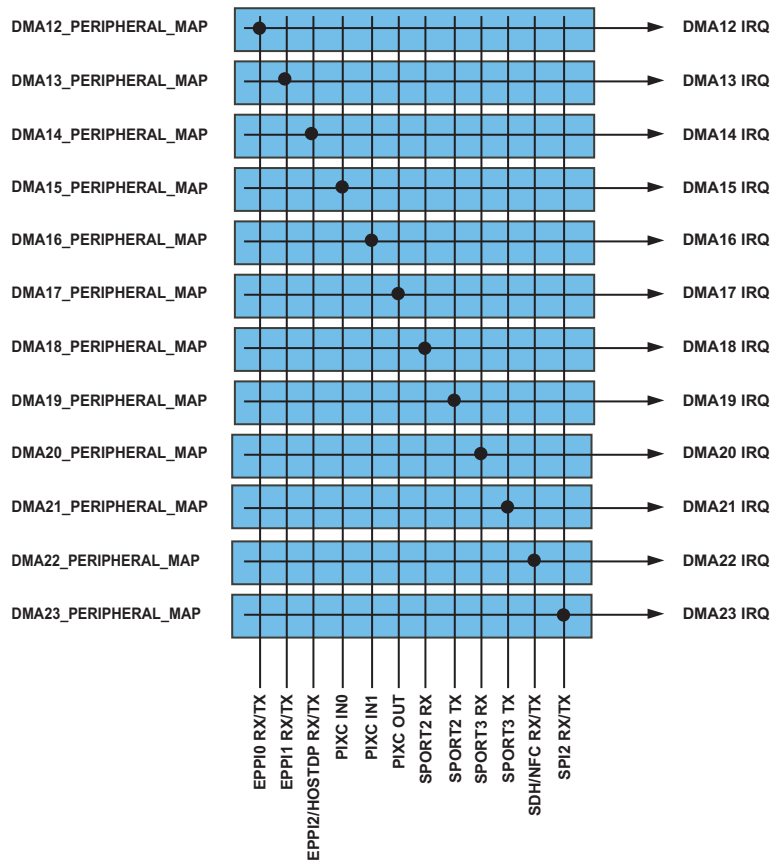


Figure 4-5. Default Peripheral-to-DMA Mapping (DMAC1 Controller)

Description of Operation

Table 4-2 shows the peripheral interrupt events, the default mapping of each event, the peripheral interrupt ID used in the system interrupt assignment registers (SIC_IARx), and the core interrupt ID. See “System Interrupt Assignment (SIC_IARx) Registers” on page 4-27.

Table 4-2. System Interrupt Controller (SIC)

Peripheral Interrupt Event	Peripheral Interrupt ID	Default Mapping	Default Core Interrupt ID	SIC Registers	
PLL Wakeup Interrupt	0	IVG7	0	SIC_IAR0	SIC_IWR0, SIC_ISR0 & SIC_IMASK0
DMAC0 Status (generic)	1	IVG7	0		
EPPI0 Error Interrupt	2	IVG7	0		
SPORT0 Error Interrupt	3	IVG7	0		
SPORT1 Error Interrupt	4	IVG7	0		
SPI0 Status Interrupt	5	IVG7	0		
UART0 Status Interrupt	6	IVG7	0		
Real-Time Clock Interrupt	7	IVG8	1		
DMA12 Interrupt (EPPI0)	8	IVG8	1	SIC_IAR1	
DMA0 Interrupt (SPORT0 RX)	9	IVG9	2		
DMA1 Interrupt (SPORT0 TX)	10	IVG9	2		
DMA2 Interrupt (SPORT1 RX)	11	IVG9	2		
DMA3 Interrupt (SPORT1 TX)	12	IVG9	2		
DMA4 Interrupt (SPI0)	13	IVG10	3		
DMA6 Interrupt (UART0 RX)	14	IVG10	3		
DMA7 Interrupt (UART0 TX)	15	IVG10	3		

Table 4-2. System Interrupt Controller (SIC) (Cont'd)

Peripheral Interrupt Event	Peripheral Interrupt ID	Default Mapping	Default Core Interrupt ID	SIC Registers	
Timer 8 Interrupt	16	IVG11	4	SIC_IAR2	SIC_IWR0, SIC_ISR0 & SIC_IMASK0
Timer 9 Interrupt	17	IVG11	4		
Timer 10 Interrupt	18	IVG11	4		
Pin Interrupt 0 (PINT0)	19	IVG12	5		
Pin Interrupt 1 (PINT1)	20	IVG12	5		
MDMA Stream 0 Interrupt	21	IVG13	6		
MDMA Stream 1 Interrupt	22	IVG13	6		
Software Watchdog Timer Interrupt	23	IVG13	6		
DMAC1 Status (generic)	24	IVG7	0	SIC_IAR3	
SPO2 Error Interrupt	25	IVG7	0		
SPO3 Error Interrupt	26	IVG7	0		
MXVR Synchronous Data Interrupt	27	IVG7	0		
SPI1 Status Interrupt	28	IVG7	0		
SPI2 Status Interrupt	29	IVG7	0		
UART1 Status Interrupt	30	IVG7	0		
UART2 Status Interrupt	31	IVG7	0		

Description of Operation

Table 4-2. System Interrupt Controller (SIC) (Cont'd)

Peripheral Interrupt Event	Peripheral Interrupt ID	Default Mapping	Default Core Interrupt ID	SIC Registers		
CAN0 Status Interrupt	32	IVG7	0	SIC_IAR4	SIC_IWR1, SIC_ISR1 & SIC_IMASK1	
DMA18 Interrupt (SPORT2 RX)	33	IVG9	2			
DMA19 Interrupt (SPORT2 TX)	34	IVG9	2			
DMA20 Interrupt (SPORT3 RX)	35	IVG9	2			
DMA21 Interrupt (SPORT3 TX)	36	IVG9	2			
DMA13 Interrupt (EPPI1)	37	IVG9	2			
DMA14 Interrupt (EPPI2, HOSTDP)	38	IVG9	2			
DMA5 Interrupt (SPI1)	39	IVG10	3			
DMA23 Interrupt (SPI2)	40	IVG10	3			SIC_IAR5
DMA8 Interrupt (UART1 RX)	41	IVG10	3			
DMA9 Interrupt (UART1 TX)	42	IVG10	3			
DMA10 Interrupt (ATAPI RX)	43	IVG10	3			
DMA11 Interrupt (ATAPI TX)	44	IVG10	3			
TWI0 Interrupt	45	IVG11	4			
TWI1 Interrupt	46	IVG11	4			
CAN0 Receive Interrupt	47	IVG11	4			

Table 4-2. System Interrupt Controller (SIC) (Cont'd)

Peripheral Interrupt Event	Peripheral Interrupt ID	Default Mapping	Default Core Interrupt ID	SIC Registers	
CAN0 Transmit Interrupt	48	IVG11	4	SIC_IAR6	SIC_IWR1, SIC_ISR1 & SIC_IMASK1
MDMA Stream 2 Interrupt	49	IVG13	6		
MDMA Stream 3 Interrupt	50	IVG13	6		
MXVR Status Interrupt	51	IVG11	4		
MXVR Control Message Interrupt	52	IVG11	4		
MXVR Asynchronous Packet Interrupt	53	IVG11	4		
EPPI1 Error Interrupt	54	IVG7	0		
EPPI2 Error Interrupt	55	IVG7	0		
UART3 Status Interrupt	56	IVG7	0	SIC_IAR7	
HOSTDP Status Interrupt	57	IVG7	0		
Reserved	58	IVG7	0		
Pixel Compositor (PIXC) Status Interrupt	59	IVG7	0		
NFC Status Interrupt	60	IVG7	0		
ATAPI Status Interrupt	61	IVG7	0		
CAN1 Status Interrupt	62	IVG7	0		
Handshake DMA Status (logical OR of DMAR0 Block Interrupt, DMAR1 Block Interrupt, DMAR0 Overflow Error Interrupt, and DMAR1 Overflow Error Interrupt)	63	IVG7	0		

Description of Operation

Table 4-2. System Interrupt Controller (SIC) (Cont'd)

Peripheral Interrupt Event	Peripheral Interrupt ID	Default Mapping	Default Core Interrupt ID	SIC Registers	
DMA15 Interrupt (PIXC IN0)	64	IVG8	1	SIC_IAR8	SIC_IWR2, SIC_ISR2 & SIC_IMASK2
DMA16 Interrupt (PIXC IN1)	65	IVG8	1		
DMA17 Interrupt (PIXC OUT)	66	IVG8	1		
DMA22 Interrupt (SDH/NFC)	67	IVG8	1		
Rotary Counter (CNT) Interrupt	68	IVG8	1		
Keypad (KEY) Interrupt	69	IVG8	1		
CAN1 RX Interrupt	70	IVG11	4		
CAN1 TX Interrupt	71	IVG11	4		
SDH Mask 0 Interrupt	72	IVG11	4	SIC_IAR9	
SDH Mask 1 Interrupt	73	IVG11	4		
Reserved	74	IVG11	4		
USB_INT0 Interrupt	75	IVG11	4		
USB_INT1 Interrupt	76	IVG11	4		
USB_INT2 Interrupt	77	IVG11	4		
USB_DMAINT Interrupt	78	IVG11	4		
OTPSEC Interrupt	79	IVG11	4		

Table 4-2. System Interrupt Controller (SIC) (Cont'd)

Peripheral Interrupt Event	Peripheral Interrupt ID	Default Mapping	Default Core Interrupt ID	SIC Registers	
Reserved	80	IVG11	4	SIC_IAR10	SIC_IWR2, SIC_ISR2 & SIC_IMASK2
Reserved	81	IVG11	4		
Reserved	82	IVG11	4		
Reserved	83	IVG11	4		
Reserved	84	IVG11	4		
Reserved	85	IVG11	4		
Timer 0 Interrupt	86	IVG11	4	SIC_IAR11	
Timer 1 Interrupt	87	IVG11	4		
Timer 2 Interrupt	88	IVG11	4		
Timer 3 Interrupt	89	IVG11	4		
Timer 4 Interrupt	90	IVG11	4		
Timer 5 Interrupt	91	IVG11	4		
Timer 6 Interrupt	92	IVG11	4		
Timer 7 Interrupt	93	IVG11	4		
Pin Interrupt 2 (PINT2)	94	IVG12	5		
Pin Interrupt 3 (PINT3)	95	IVG12	5		

The peripheral interrupt structure of the processor is flexible. Upon reset, multiple peripheral interrupts share a single, general-purpose interrupt in the core by default, as shown in [Table 4-2](#).



UART2 and UART3 are not assigned to peripheral channels by default. To assign one of these peripherals to a DMA channel, refer to [Table 5-1 on page 5-10](#).

Programming Model

An interrupt service routine that supports multiple interrupt sources must interrogate the appropriate system memory-mapped registers (MMRs) to determine which peripheral generated the interrupt.

Programming Model

The programming model for the system interrupts is described in the following sections.

System Interrupt Initialization

If the default assignments shown in [Table 4-2 on page 4-16](#) are acceptable, then interrupt initialization involves only:

- Initialization of the core event vector table (EVT) vector address entries
- Initialization of the IMASK register
- Unmasking the specific peripheral interrupts in the SIC_IMASKx register that the system requires

System Interrupt Processing Summary

Referring to [Figure 4-6 on page 4-25](#), note when an interrupt (interrupt A) is generated by an interrupt-enabled peripheral:

1. The SIC_ISRx register logs the request and keeps track of system interrupts that are asserted but not yet serviced (that is, an interrupt service routine that has not yet cleared the interrupt).
2. The SIC_IWRx register checks to see if it should wake up the core from an idled or sleep mode state based on this interrupt request.

3. The `SIC_IMASKx` register masks-off or enables interrupts from peripherals at the system level. If interrupt A is not masked, the request proceeds to Step 4.
4. The `SIC_IARx` register, which maps the peripheral interrupts to a smaller set of general-purpose core interrupts (`IVG7-IVG15`), determines the core priority of interrupt A.
5. The `ILAT` bit adds interrupt A to its log of interrupts latched by the core but not yet actively being serviced.
6. the `IMASK` bit 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.
7. The event vector table (EVT) is accessed to look up the appropriate vector for interrupt A's ISR.
8. 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, the `IPEND` bit tracks all pending interrupts, as well as those being presently serviced.
9. When the interrupt service routine for interrupt A is 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.

Programming Model

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 interrupt service routine (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.

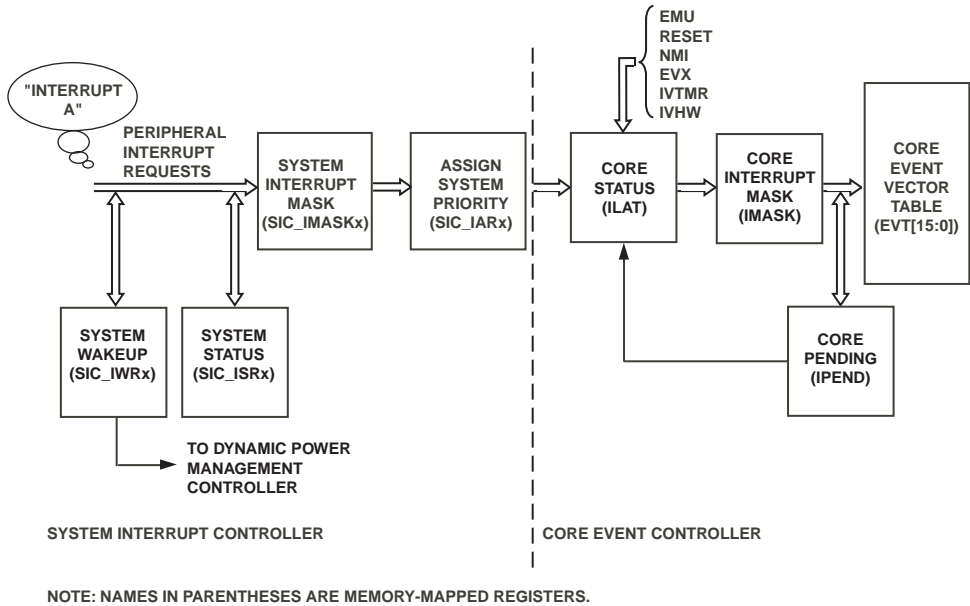


Figure 4-6. Interrupt Processing Block Diagram

System Interrupt Controller Registers

System interrupt controller (SIC) registers can be read from or written to 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.

The SIC registers are described in [Table 4-3](#).

Table 4-3. System Interrupt Controller Registers

Name	Description/ Refer to
SIC_IARx	System Interrupt Assignment registers Listing on page 4-27
SIC_IMASKx	System Interrupt Mask registers Listing on page 4-34
SIC_ISRx	System Interrupt Status registers Listing
SIC_IWRx	System Interrupt Wakeup registers Listing on page 4-40

System Interrupt Assignment (SIC_IARx) Registers

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

Table 4-4. IVG Select Definitions

General-Purpose Interrupt	Value in SIC_IAR
IVG7	0
IVG8	1
IVG9	2
IVG10	3
IVG11	4
IVG12	5
IVG13	6
IVG14	7
IVG15	8

System Interrupt Controller Registers

The system interrupt assignment registers (SIC_IARx) are shown in [Figure 4-7](#) through [Figure 4-18](#).

System Interrupt Assignment Register 0 (SIC_IAR0)

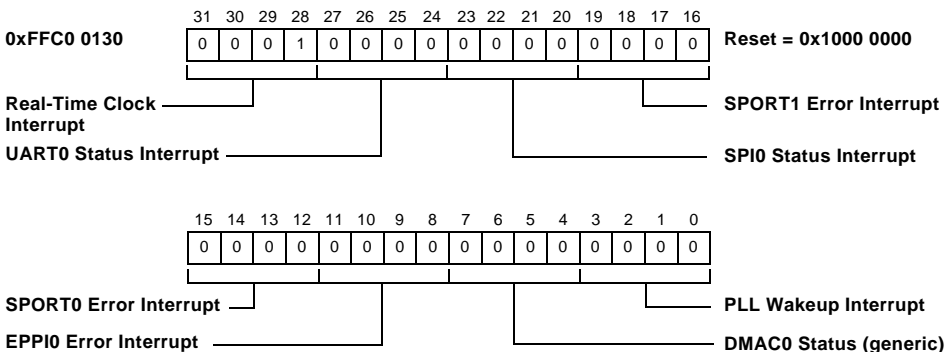


Figure 4-7. System Interrupt Assignment Register 0 (SIC_IAR0)

System Interrupt Assignment Register 1 (SIC_IAR1)

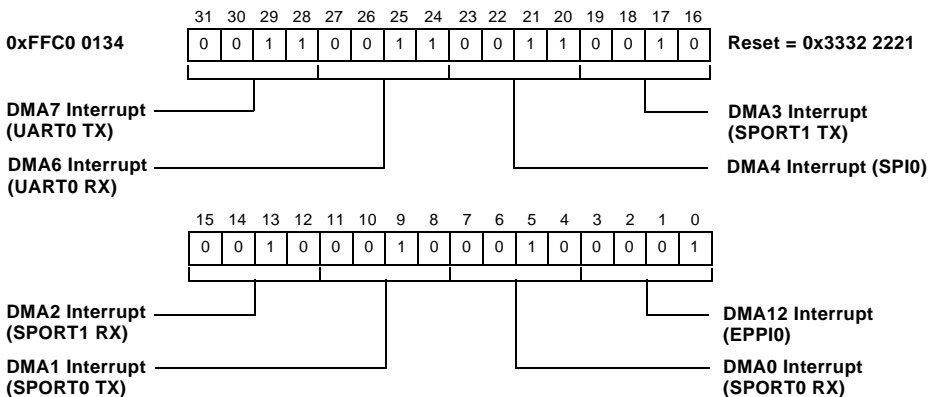


Figure 4-8. System Interrupt Assignment Register 1 (SIC_IAR1)

System Interrupt Assignment Register 2 (SIC_IAR2)

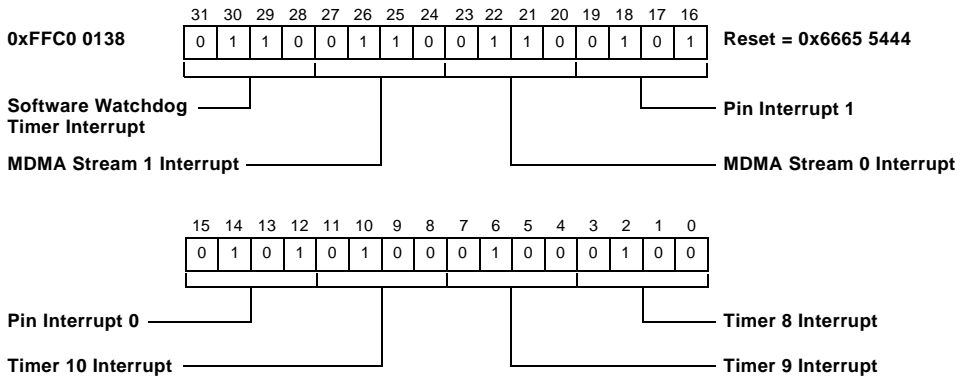


Figure 4-9. System Interrupt Assignment Register 2 (SIC_IAR2)

System Interrupt Assignment Register 3 (SIC_IAR3)

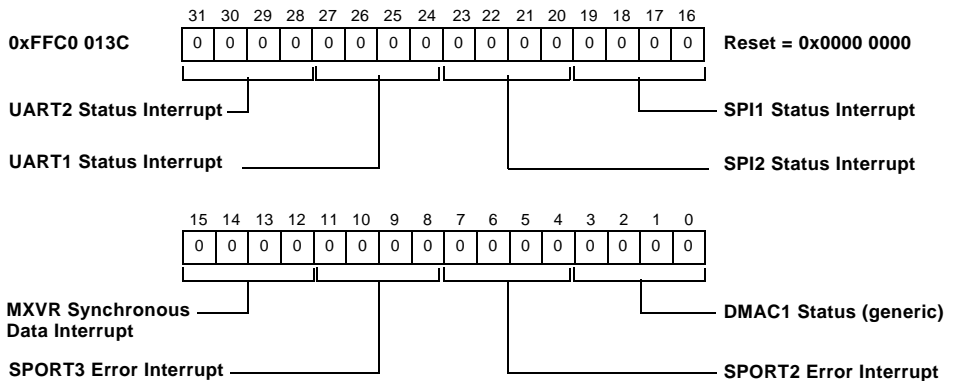


Figure 4-10. System Interrupt Assignment Register 3 (SIC_IAR3)

System Interrupt Controller Registers

System Interrupt Assignment Register 4 (SIC_IAR4)

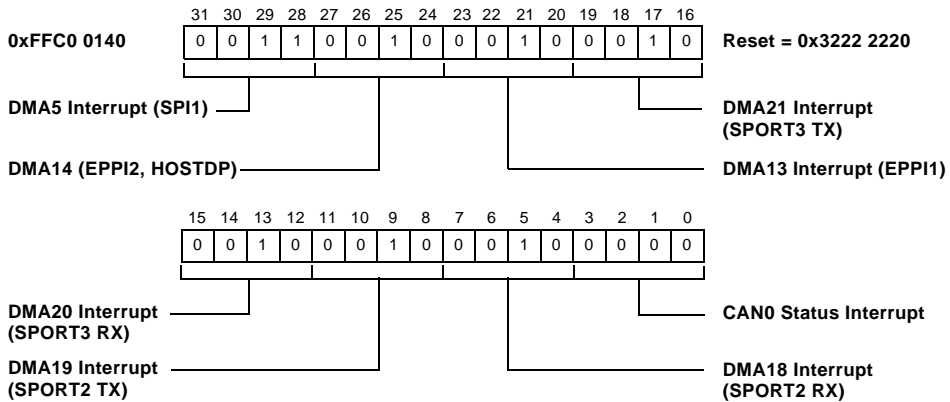


Figure 4-11. System Interrupt Assignment Register 4 (SIC_IAR4)

System Interrupt Assignment Register 5 (SIC_IAR5)

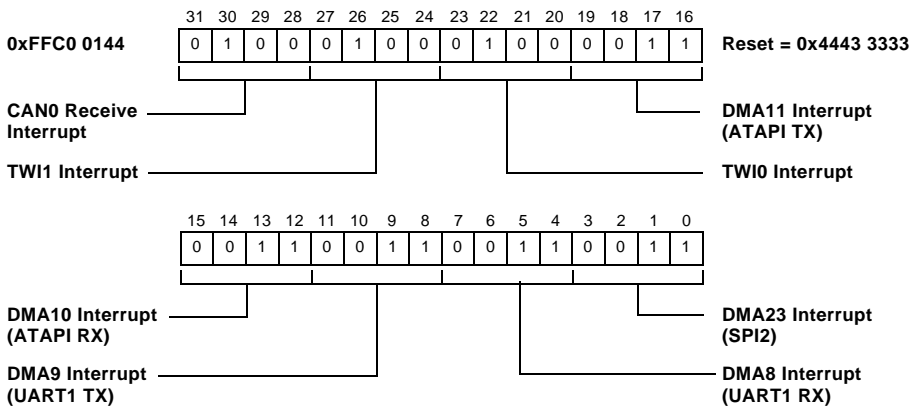


Figure 4-12. System Interrupt Assignment Register 5 (SIC_IAR5)

System Interrupt Assignment Register 6 (SIC_IAR6)

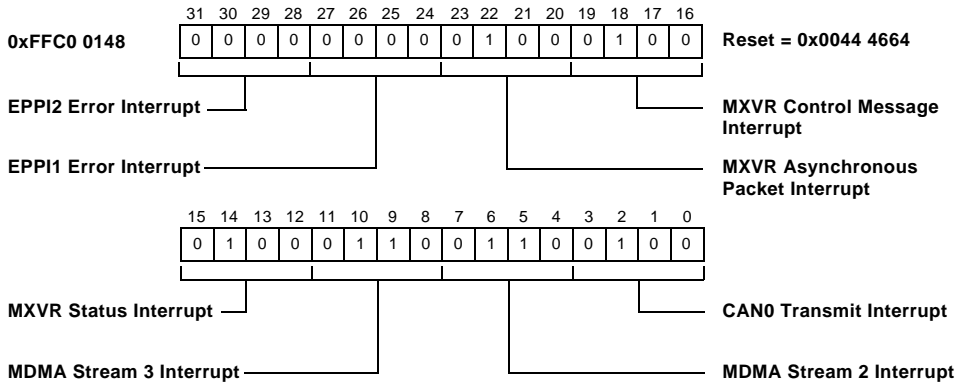


Figure 4-13. System Interrupt Assignment Register 6 (SIC_IAR6)

System Interrupt Assignment Register 7 (SIC_IAR7)

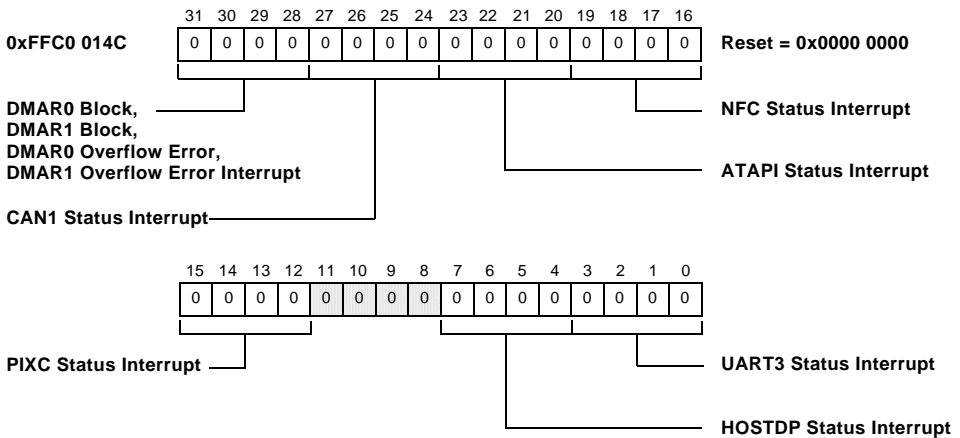


Figure 4-14. System Interrupt Assignment Register 7 (SIC_IAR7)

System Interrupt Controller Registers

System Interrupt Assignment Register 8 (SIC_IAR8)

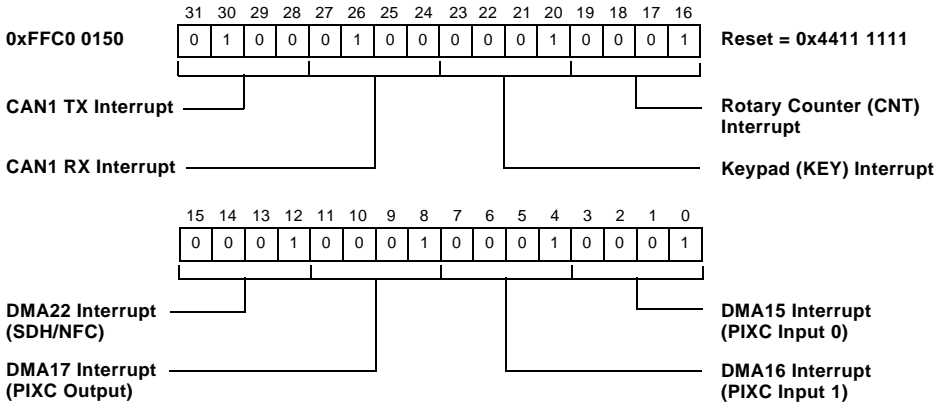


Figure 4-15. System Interrupt Assignment Register 8 (SIC_IAR8)

System Interrupt Assignment Register 9 (SIC_IAR9)

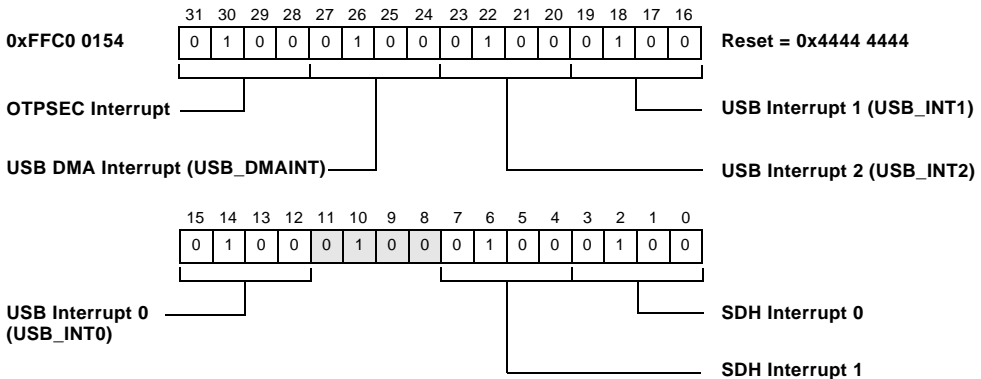


Figure 4-16. System Interrupt Assignment Register 9 (SIC_IAR9)

System Interrupt Assignment Register 10 (SIC_IAR10)

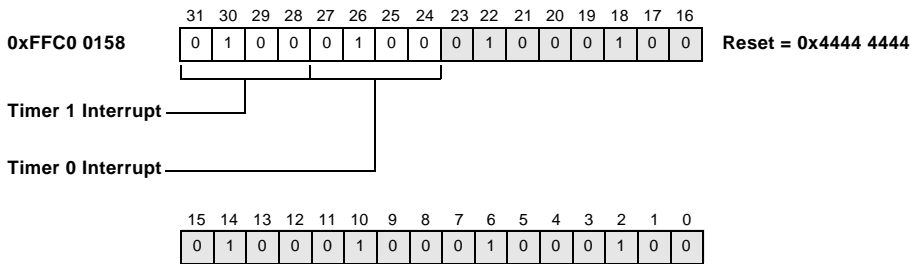


Figure 4-17. System Interrupt Assignment Register 10 (SIC_IAR10)

System Interrupt Assignment Register 11 (SIC_IAR11)

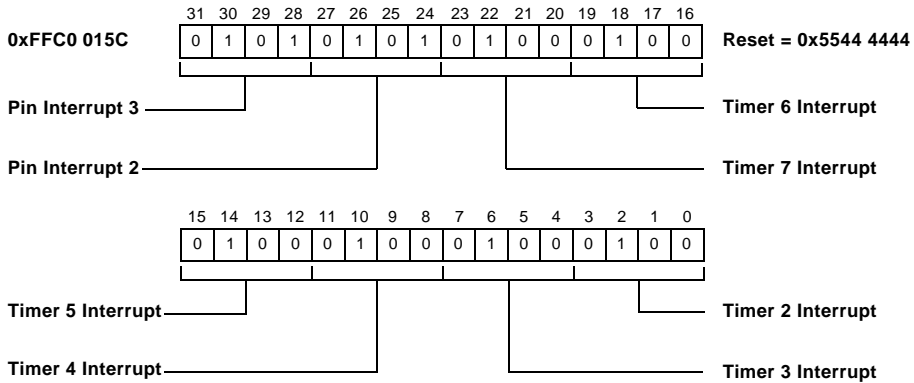


Figure 4-18. System Interrupt Assignment Register 11 (SIC_IAR11)

System Interrupt Controller Registers

System Interrupt Mask (SIC_IMASKx) Registers

The system interrupt mask registers (SIC_IMASKx) are shown in [Figure 4-19](#) through [Figure 4-21](#).

System Interrupt Mask Register 0 (SIC_IMASK0)

For all bits, 0 - Interrupt masked, 1 - Interrupt enabled

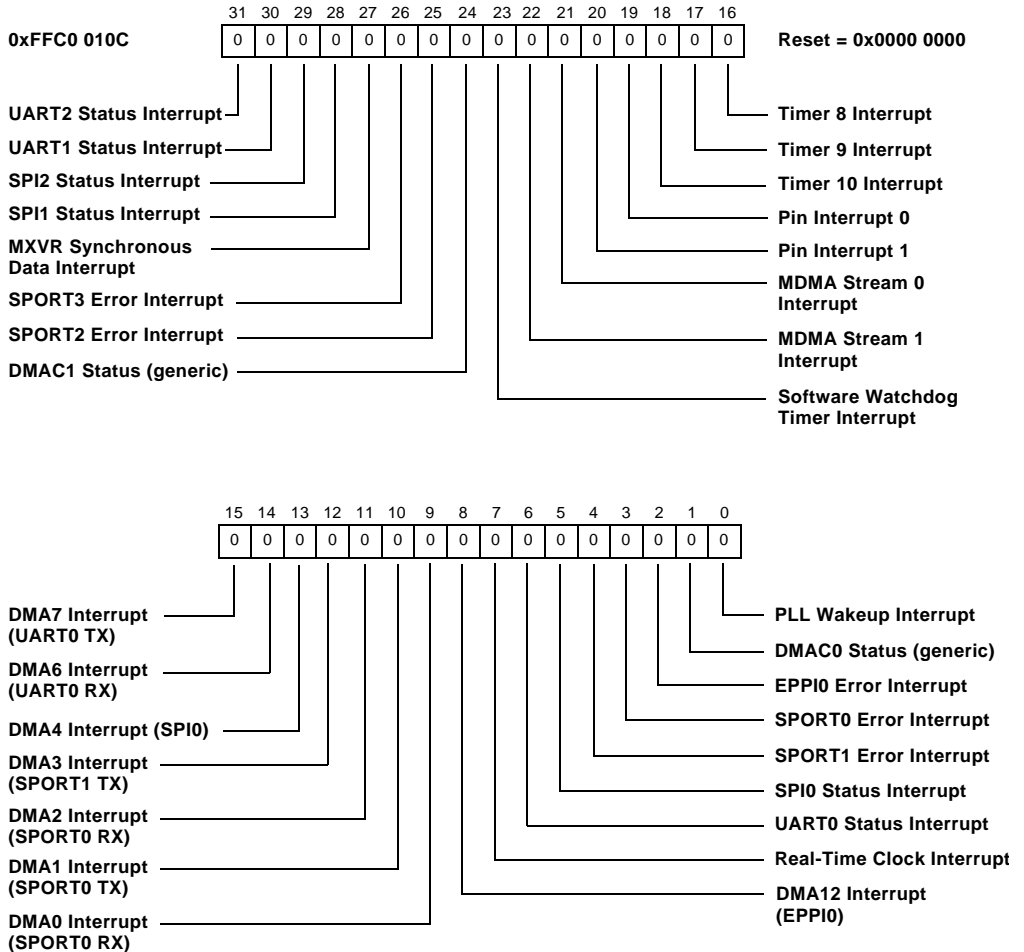


Figure 4-19. System Interrupt Mask Register 0 (SIC_IMASK0)

System Interrupt Controller Registers

System Interrupt Mask Register 1 (SIC_IMASK1)

For all bits, 0 - Interrupt masked, 1 - Interrupt enabled

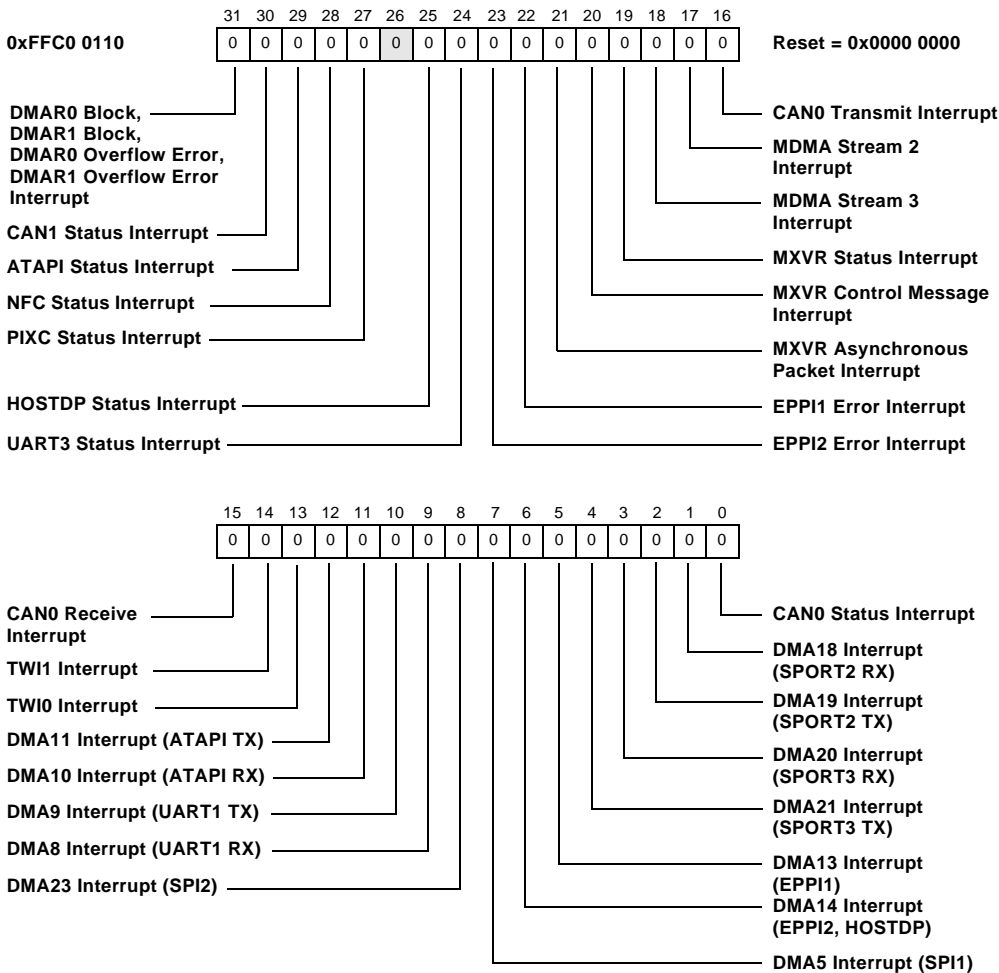


Figure 4-20. System Interrupt Mask Register 1 (SIC_IMASK1)

System Interrupt Mask Register 2 (SIC_IMASK2)

For all bits, 0 - Interrupt masked, 1 - Interrupt enabled

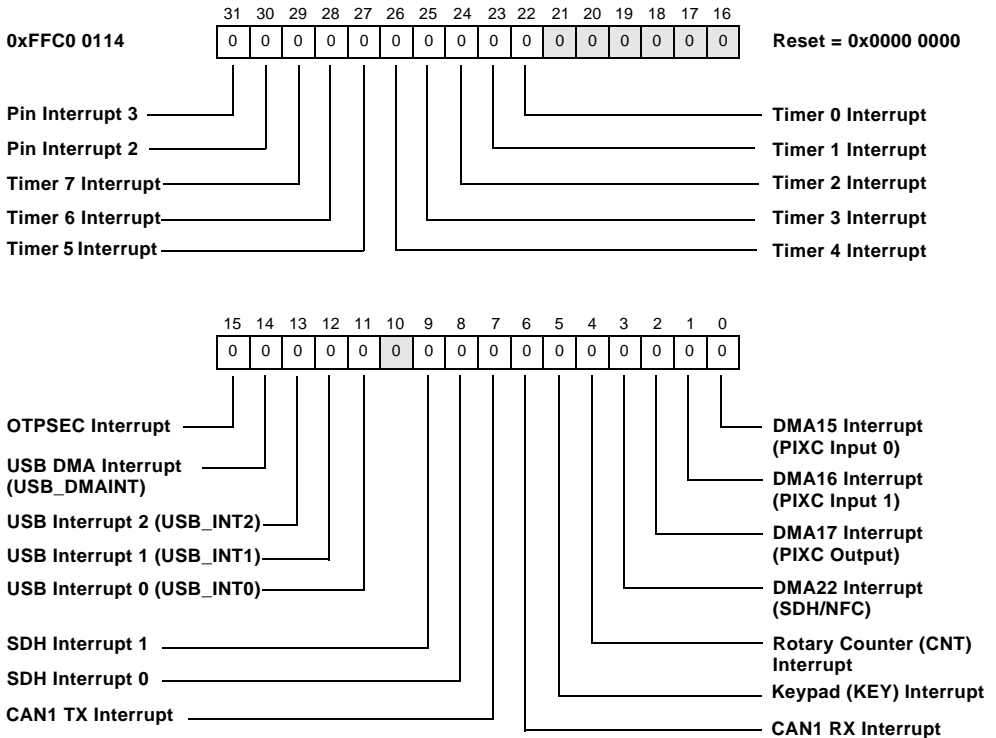


Figure 4-21. System Interrupt Mask Register 2 (SIC_IMASK2)

System Interrupt Status (SIC_ISRx) Registers

The system interrupt status registers (SIC_ISRx) are shown in [Figure 4-22](#), [Figure 4-23](#), and [Figure 4-24](#).

System Interrupt Status Register 0 (SIC_ISR0)

For all bits, 0 - Deasserted, 1 - Asserted

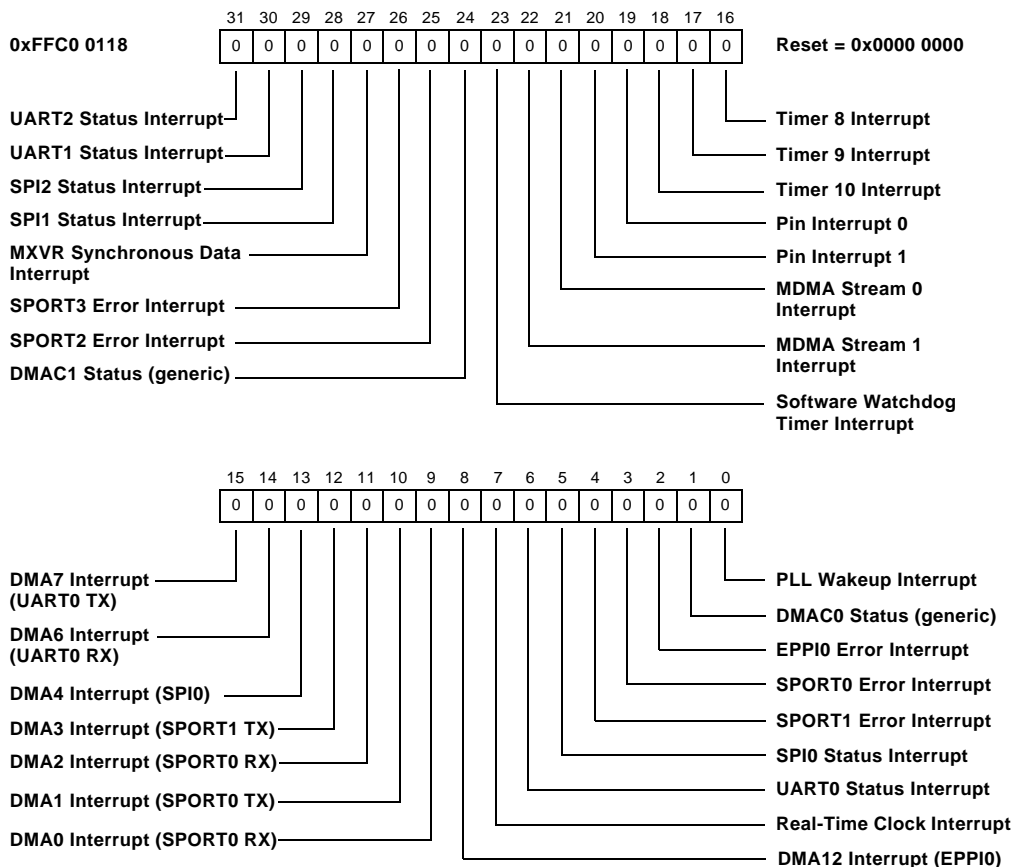


Figure 4-22. System Interrupt Status Register 0 (SIC_ISR0)

System Interrupt Status Register 1 (SIC_ISR1)

For all bits, 0 - Deasserted, 1 - Asserted

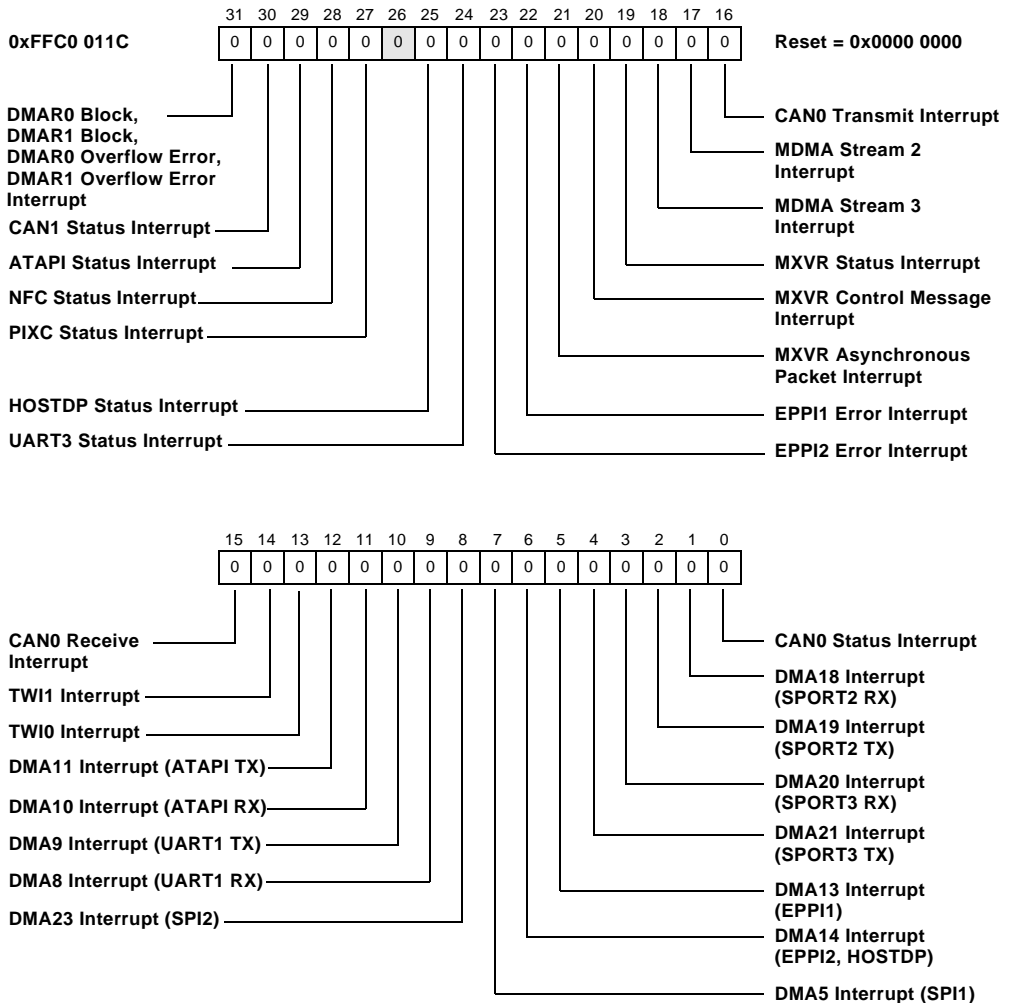


Figure 4-23. System Interrupt Status Register 1 (SIC_ISR1)

System Interrupt Controller Registers

System Interrupt Status Register 2 (SIC_ISR2)

For all bits, 0 - Deasserted, 1 - Asserted

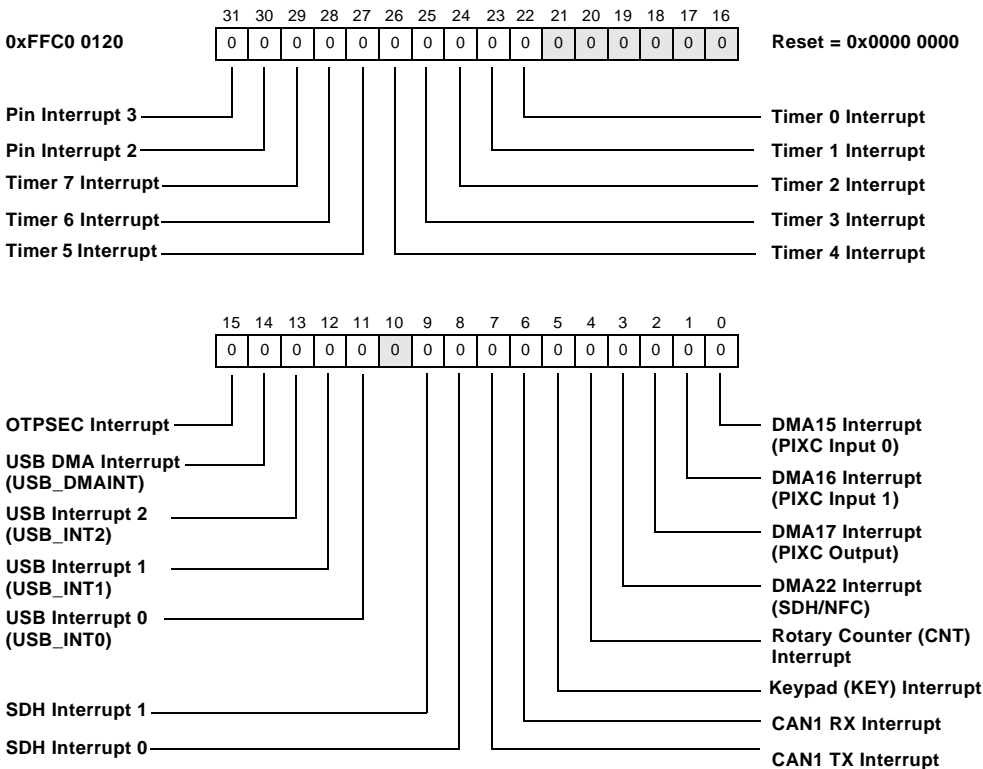


Figure 4-24. System Interrupt Status Register 2 (SIC_ISR2)

System Interrupt Wakeup (SIC_IWRx) Registers

The system interrupt wakeup registers (SIC_IWRx) are shown in [Figure 4-25](#), [Figure 4-26](#), and [Figure 4-27](#).

System Interrupt Wakeup Register 0 (SIC_IWR0)

For all bits, 0 - Wakeup function not enabled, 1 - Wakeup function enabled

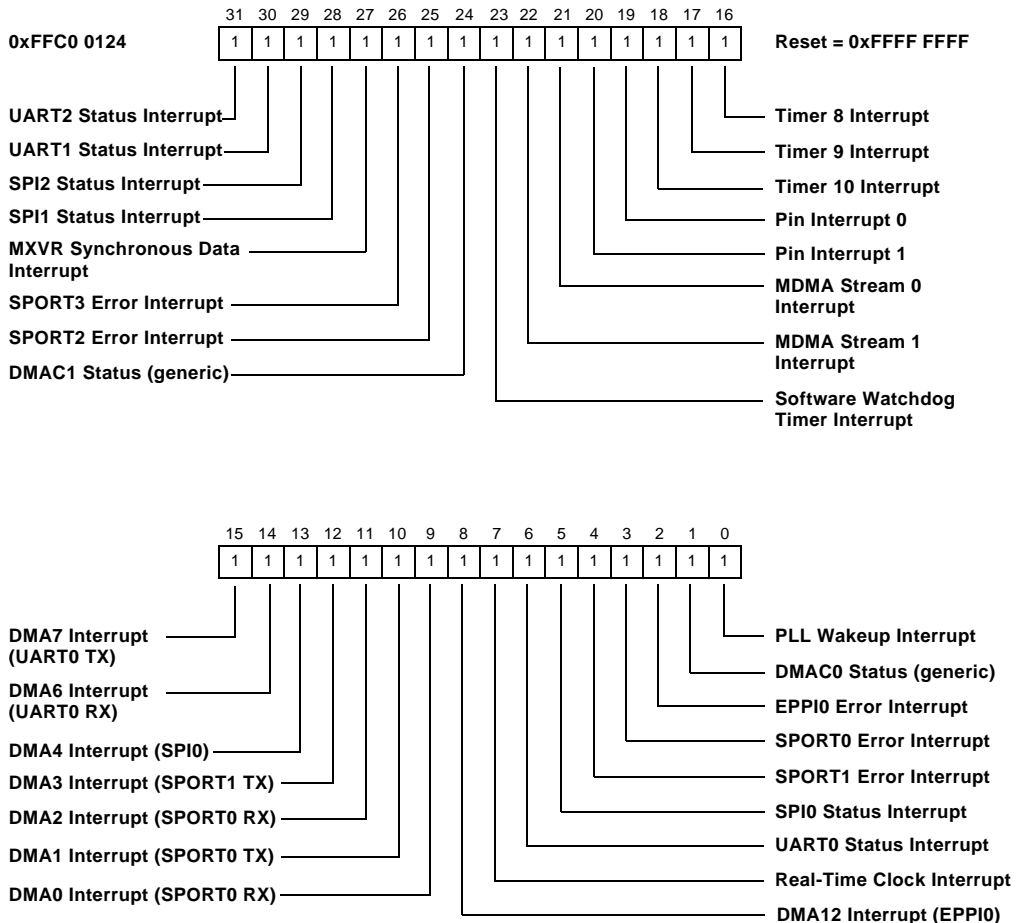


Figure 4-25. System Interrupt Wakeup Register 0 (SIC_IWR0)

System Interrupt Controller Registers

System Interrupt Wakeup Register 1 (SIC_IWR1)

For all bits, 0 - Wakeup function not enabled, 1 - Wakeup function enabled

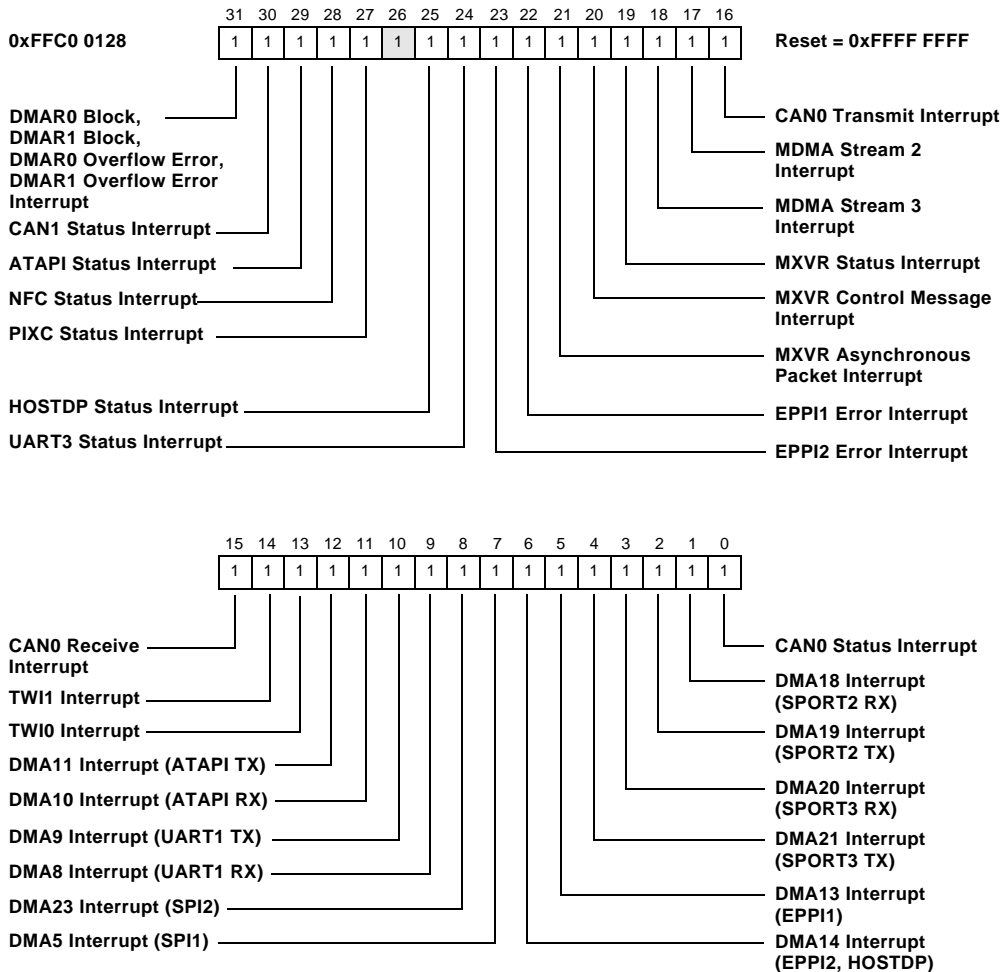


Figure 4-26. System Interrupt Wakeup Register 1 (SIC_IWR1)

System Interrupt Wakeup Register 2 (SIC_IWR2)

For all bits, 0 - Wakeup function not enabled, 1 - Wakeup function enabled

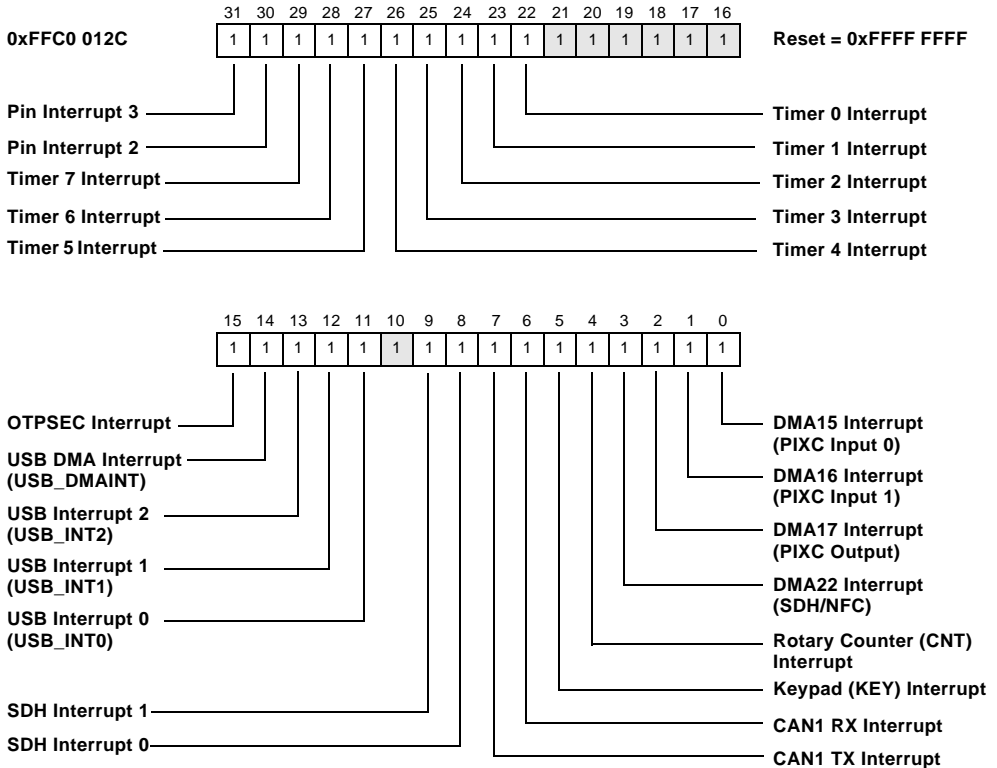


Figure 4-27. System Interrupt Wakeup Register 2 (SIC_IWR2)

Programming Examples

The following sections provide examples for programming system interrupts.

Clearing Interrupt Requests

When the processor services a core event it automatically clears the requesting bit in the ILAT register and no further action is required by the interrupt service routine. It is important to understand that the SIC controller does not provide any interrupt acknowledgment feedback mechanism from the CEC controller back to the peripherals. Although the ILAT bits clear in the same way when a peripheral interrupt is serviced, the signalling peripheral does not release its level-sensitive request until it is explicitly instructed by software. If however, the peripheral keeps requesting, the respective ILAT bit is set again immediately and the service routine is invoked again as soon as its first run terminates by an RTI instruction.

Every software routine that services peripheral interrupts must clear the signalling interrupt request in the respective peripheral. The individual peripherals provide customized mechanisms for how to clear interrupt requests. Receive interrupts, for example, are cleared when received data is read from the respective buffers. Transmit requests typically clear when software (or DMA) writes new data into the transmit buffers. These implicit acknowledge mechanisms avoid the need for cycle-consuming software handshakes in streaming interfaces. Other peripherals such as timers, GPIOs, and error requests require explicit acknowledge instructions, which are typically performed by efficient W1C (write-1-to-clear) operations.

[Listing 4-1](#) shows a representative example of how a GPIO interrupt request might be serviced.

Listing 4-1. Servicing GPIO Interrupt Request

```

#include <defBF549.h>
.section program;
_portg_a_isr:
    /* push used registers */
    [--sp] = (r7:7, p5:5);
    /* clear interrupt request on GPIO pin PG2 */
    /* no matter whether used A or B channel */
    p5.l = lo(PORTGIO_CLEAR);
    p5.h = hi(PORTGIO_CLEAR);
    r7 = PG2;
    w[p5] = r7;

    /* place user code here */

    /* sync system, pop registers and exit */
    ssync;
    (r7:7, p5:5) = [sp++];
    rti;
_portg_a_isr.end:

```

The `WIC` instruction shown in this example may require several `SCLK` cycles to complete, depending on system load and instruction history. The program sequencer does not wait until the instruction completes and continues program execution immediately. The `SSYNC` instruction ensures that the `WIC` command indeed cleared the request in the GPIO peripheral before the `RTI` instruction executes. However, the `SSYNC` instruction does not guarantee that the release of interrupt request has also been recognized by the CEC controller, which may require a few more `CCLK` cycles depending on the `CCLK-to-SCLK` ratio. In service routines consisting of a few instructions only, two `SSYNC` instructions are recommended between the clear command and the `RTI` instruction. However, one `SSYNC` instruction is typically sufficient if the clear command performs in the very beginning

Programming Examples

of the service routine, or the `SSYNC` instruction is followed by another set of instructions before the service routine returns. Commonly, a pop-multiple instruction is used for this purpose as shown in [Listing 4-1](#).

The level-sensitive nature of peripheral interrupts enables more than one of them to share the same IVG channel and therefore the same interrupt priority. This is programmable using the assignment registers. Then a common service routine typically interrogates the `SIC_ISRx` register to determine the signalling interrupt source. If multiple peripherals are requesting interrupts at the same time, it is up to the service routine to either service all requests in a single pass or to service them one by one. If only one request is serviced and the respective request is cleared by software before the `RTI` instruction executes, the same service routine is invoked another time because the second request is still pending. While the first approach may require fewer cycles to service both requests, the second approach enables higher priority requests to be serviced more quickly in a non-nested interrupt system setup.

5 DIRECT MEMORY ACCESS

This chapter describes the direct memory access (DMA) controllers. The features common to all the DMA channels, as well as how DMA operations are set up are also described. 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 2-23](#).

This chapter does not cover the DMA controllers associated with the USB and MXVR peripherals. For this information, refer to the appropriate peripheral chapter.

The chapter includes the following sections:

- [“Overview and Features” on page 5-2](#)
- [“DMA Controller Overview” on page 5-6](#)
- [“Modes of Operation” on page 5-17](#)
- [“Functional Description” on page 5-25](#)
- [“Programming Model” on page 5-61](#)
- [“DMA Registers” on page 5-74](#)
- [“Programming Examples” on page 5-129](#)

Overview and Features

The processor uses DMA to transfer data between 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 controller carries out the data transfers independent of processor activity.

The processor has two DMA controllers: DMAC0 has a 16-bit data bus, while DMAC1 has a 32-bit data bus.

The DMA controllers can perform several types of data transfers:

- Peripheral DMA transfers data between memory and on-chip peripherals. The processor has 24 peripheral DMA channels that support 21 peripherals.
 - SPORT0, SPORT1, SPORT2, and SPORT3 (dedicated DMA channels for each transmit and receive function)
 - UART0, UART1, UART2 and UART3 (dedicated DMA channels for each transmit and receive function)
 - EPPI0, EPPI1, and EPPI2/HOSTDP (each transmit and receive pair shares one DMA channel)
 - Pixel compositor (PIXC) (two dedicated DMA channels for inputs, one for output)
 - NFC/SDH (transmit and receive channels share one DMA channel)
 - ATAPI (dedicated DMA channels for transmit and receive)
 - SPI0, SPI1, and SPI2 (each transmit and receive pair shares one DMA channel)

- Memory DMA (MDMA) transfers data between memory and memory. The processor has four MDMA modules, each consisting of independent memory read and memory write channels.
- Handshaking memory DMA (HMDMA) transfers data between off-chip peripherals and memory. This enhancement of the MDMA channels enables external hardware to control the timing of individual data transfers or block transfers.

All DMAs can transport data to and from on-chip and off-chip memories, including L1, L2, boot ROM, and DDR SDRAM. The L1 scratchpad memory cannot be accessed by DMA.

DMA transfers on the processor can be descriptor-based or register-based. 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. 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.

Examples of DMA styles supported by flex descriptors include:

- A single linear buffer that stops on completion (`FLOW = stop mode`)
- A linear buffer with strides equal 1 or greater, zero or negative (`DMAx_X_MODIFY` register)
- A circular, auto-refreshing buffer that interrupts on each full buffer
- A similar buffer that interrupts on fractional buffers (for example, 1/2, 1/4) (2D DMA)

Overview and Features

- 1D DMA, using a set of identical ping-pong buffers defined by a linked ring of 3-word descriptors, each containing a link pointer and a 32-bit address }
- 1D DMA, using a linked list of 5-word descriptors containing a link pointer, a 32-bit address, the length of the buffer, and the DMA configuration.
- 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

The following functions can be served by DMA channels:

- EPPI2–0 receive
- EPPI2–0 transmit
- Host DMA receive/transmit
- PIXC image data (read from memory)
- PIXC overlay data (read from memory)
- PIXC results (write to memory)
- SPORT3–0 receive
- SPORT3–0 transmit
- UART3–0 receive
- UART3–0 transmit
- SPI2–0 receive
- SPI2–0 transmit

- NFC receive/transmit
- SDH receive/transmit
- ATAPI receive
- ATAPI transmit
- MDMA3-0 destination
- MDMA3-0 source

DMA Controller Overview

Figure 5-1 and Figure 5-2 provide block diagrams of the DMA controllers.

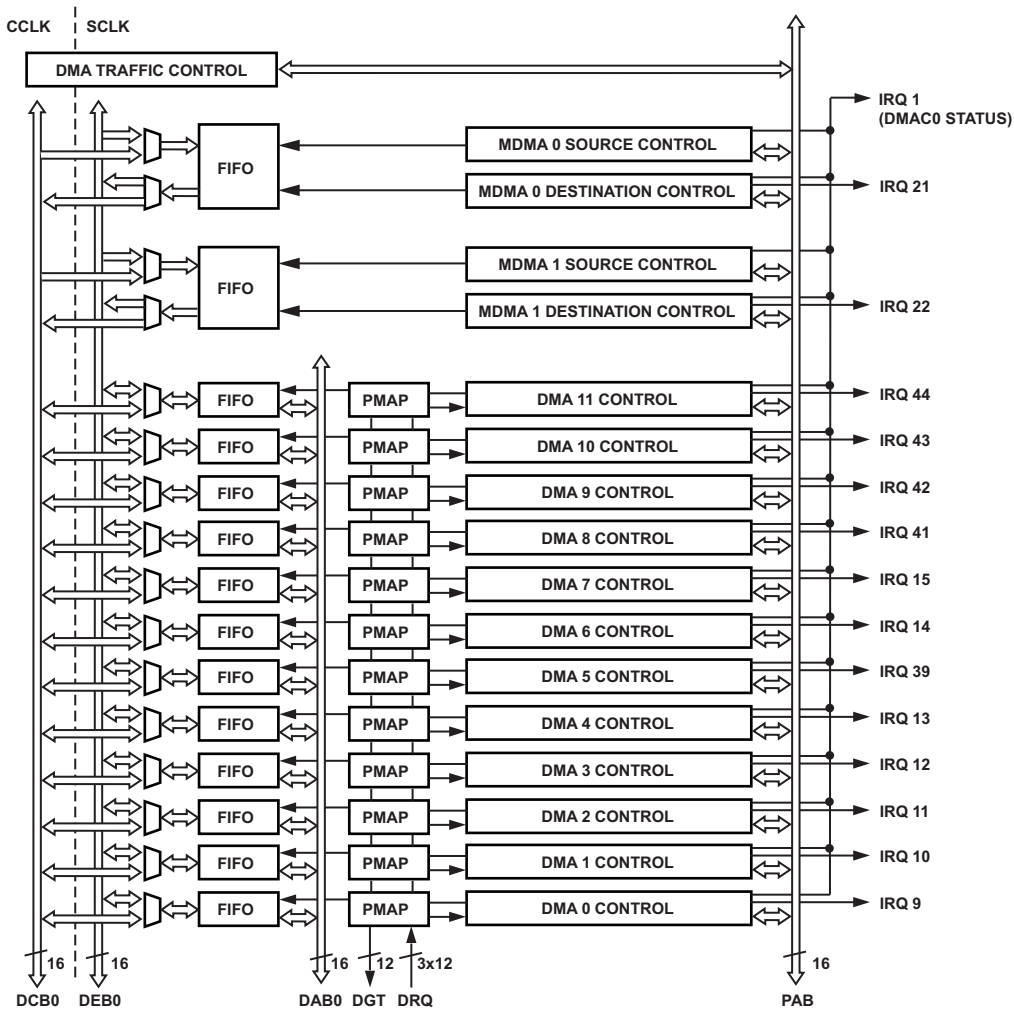


Figure 5-1. DMAC0 Controller Block Diagram

In the figures, DRQ = DMA request (see [Table 5-21 on page 5-117](#)) and DGT = DMA grant.

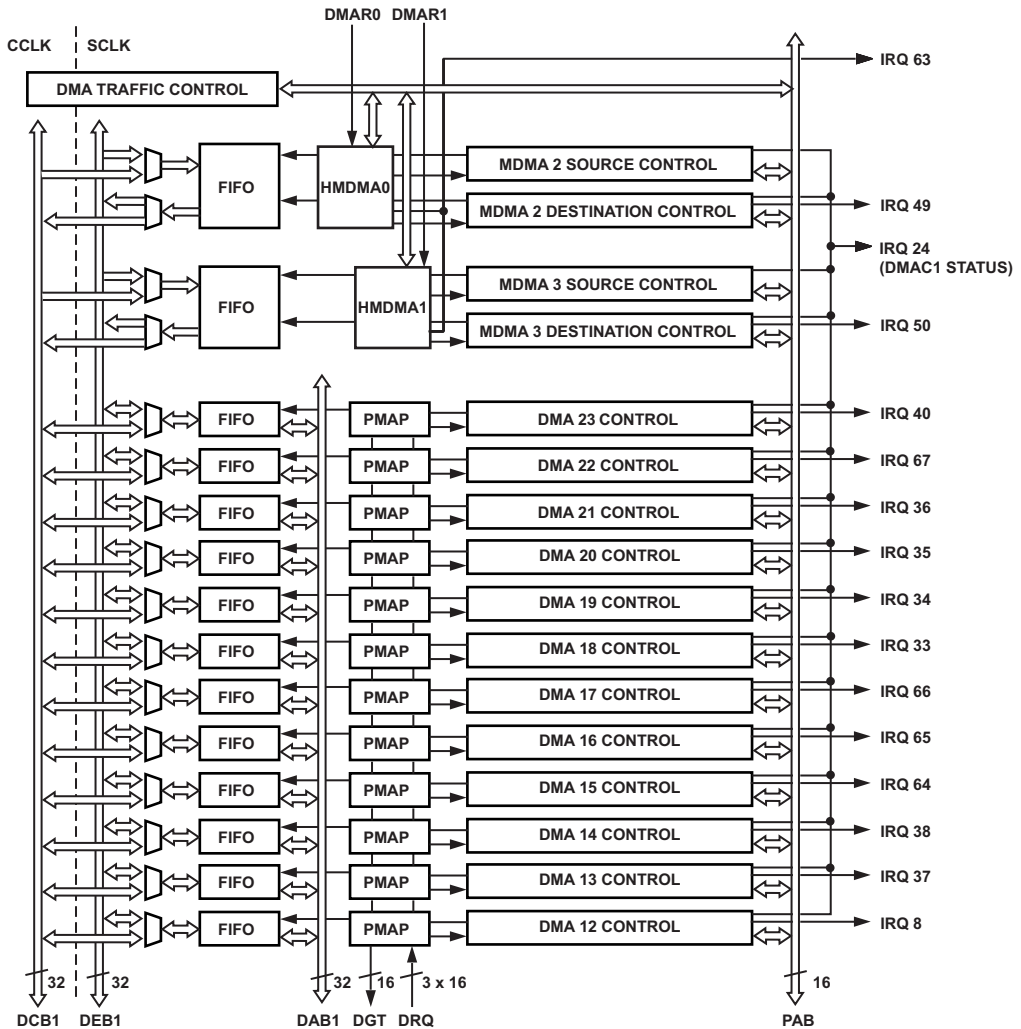


Figure 5-2. DMAC1 Controller Block Diagram

External Interfaces

The DMA does not connect external memories and devices directly. Rather, data is passed through the EBIU port. Any kind of device that is supported by the EBIU can also be accessed by peripheral DMA or memory DMA operation. This is typically flash memory, SRAM, DDR SDRAM, FIFOs, or memory-mapped peripheral devices.

Handshaking MDMA operation is supported by two MDMA request input pins, `DMAR0` and `DMAR1`. The `DMAR0` pin controls transfer timing on the MDMA2 source or destination channel. The `DMAR1` pin controls the source or destination channel of MDMA3. With these pins, external FIFO devices, ADC or DAC converters, or other streaming or block-processing devices can use the MDMA channels to exchange their data or data buffers with the Blackfin processor memory.

Both `DMARx` pins reside on port H and are multiplexed with MXVR signals. To enable their function, `PH5` and/or `PH6` must be set in the `PORTH_FER` registers and the related bit fields in the `PORTH_MUX` register must be set to “b#01”. The `REP` bit in the respective `HMDMAX_CONTROL` register controls whether the `DMARx` inputs trigger on falling or rising edges of the connect strobe.

Internal Interfaces

[Figure 2-1 on page 2-3](#) of the “Chip Bus Hierarchy” chapter shows the dedicated DMA buses used by the DMA controllers to interconnect L1 and L2 memories, the on-chip peripherals, and the EBIU port.

The 16-bit DMA core bus (`DCB0`) allows `DMAC0` to access either a dedicated port of L1 memory or the on-chip memory other than L1. A 32-bit DMA core bus (`DCB1`) allows `DMAC1` to access either a dedicated port of L1 memory or the on-chip memory other than L1. These buses, along with `DCB2` from MXVR and `DCB3` from USB, operate at the system

clock (`SCLK`) frequency. Internal arbitration is performed between accesses on these four buses and translates the requests into the core clock (`CCLK`) domain for either memory other than L1 or L1 memory.

The 16-bit DMA access bus (`DAB0`) connects `DMAC0` to the following on-chip peripherals: `SPORT0`, `SPORT1`, `SPI0`, `SPI1`, `UART0`, `UART1`, `ATAPI`.

The 32-bit DMA access bus (`DAB1`) connects `DMAC1` to the following on-chip peripherals: `EPPI0`, `EPPI1`, `EPPI2`, `HOSTDP`, `PIXC`, `SPORT2`, `SPORT3`, `UART2`, `UART3`, `SDH`, `NFC`, and `SPI2`. Both `DAB` buses operate at `SCLK` frequency.

The 16-bit DMA external bus (`DEB0`) connects the `DMAC0` to the `EBIU` port. The 32-bit DMA external bus (`DEB1`) connects `DMAC1` to the `EBIU` port.

Transferred data can be 8, 16, or 32 bits wide. `DMAC0`, however, connects only to 16-bit buses. `MDMA0` and `MDMA1` reside on `DMAC0`, while `MDMA2` and `MDMA3` reside on `DMAC1`.

In terms of `DCB` bus performance, L2 memory resembles L1 memory for the purposes of performance on the `DCB` buses.

Memory DMA can pass data every `SCLK` cycle between L1 or memory other than L1 and the `EBIU`. Transfers originating from L1 or memory other than L1 and targeting L1 or memory other than L1 require two cycles, as the `DCB` bus is used for both source and destination transfer. Similarly, transfers between two off-chip devices require `EBIU` and `DEB` resources twice. Peripheral DMA transfers can be performed every other `SCLK` cycle.

For more details on DMA performance see [“DMA Performance” on page 5-51](#).

Peripheral DMA

As can be seen in [Figure 5-1 on page 5-6](#) and [Figure 5-2 on page 5-7](#), the DMA controllers each feature 12 channels that perform transfers between peripherals and on-chip or off-chip memories. The user has full control over the mapping of DMA channels and peripherals. The default configuration, shown in [Table 5-1](#), can be changed by altering the 4-bit PMAP field in the `DMAX_PERIPHERAL_MAP` registers for the peripheral DMA channels.

Table 5-1. Default Mapping of Peripheral to DMA

DMA Channel	DMA Controller	PMAP Default ^{1, 2}	Peripheral Mapped by Default
DMA0	DMAC0	0x0	SPORT0 receive
DMA1	DMAC0	0x1	SPORT0 transmit
DMA2	DMAC0	0x2	SPORT1 receive
DMA3	DMAC0	0x3	SPORT1 transmit
DMA4	DMAC0	0x4	SPI0 receive/transmit
DMA5	DMAC0	0x5	SPI1 receive/transmit
DMA6	DMAC0	0x6	UART0 receive
DMA7	DMAC0	0x7	UART0 transmit
DMA8	DMAC0	0x8	UART1 receive
DMA9	DMAC0	0x9	UART1 transmit
DMA10	DMAC0	0xA	ATAPI receive
DMA11	DMAC0	0xB	ATAPI transmit
DMA12	DMAC1	0x0	EPPI0 receive/transmit
DMA13	DMAC1	0x1	EPPI1 receive/transmit
DMA14	DMAC1	0x2	EPPI2/Host DMA receive/transmit
DMA15	DMAC1	0x3	PIXC image data (read from memory)
DMA16	DMAC1	0x4	PIXC overlay data (read from memory)

Table 5-1. Default Mapping of Peripheral to DMA (Cont'd)

DMA Channel	DMA Controller	PMAP Default ^{1, 2}	Peripheral Mapped by Default
DMA17	DMAC1	0x5	PIXC output data (write to memory)
DMA18	DMAC1	0x6	SPORT2 receive
DMA19	DMAC1	0x7	SPORT2 transmit
DMA20	DMAC1	0x8	SPORT3 receive
DMA21	DMAC1	0x9	SPORT3 transmit
DMA22	DMAC1	0xA	SDH/NFC receive/transmit
DMA23	DMAC1	0xB	SPI2 receive/transmit
–	DMAC1	0xC	Note ³
–	DMAC1	0xD	Note ³
–	DMAC1	0xE	Note ³
–	DMAC1	0xF	Note ³

1 Host DMA and EPPI2 share a PMAP assignment on DMAC1. Host DMA is given the channel when it is enabled. Otherwise EPPI2 is given the channel.


2 NFC (NAND flash controller) and SDH (secure digital host) share a PMAP assignment on DMAC1. For more information on enabling the NFC, see [“DMA Controller 1 Peripheral Multiplexer \(DMAC1_PERIMUX\) Register” on page 5-129](#).

3 UART2 and UART3 are not assigned to peripheral channels by default. To assign one of these peripherals to a DMA channel, program the selected DMA channel with the following PMAP value: 0xC for UART2 RX, 0xD for UART2 TX, 0xE for UART3 RX, or 0xF for UART3 TX


The default configuration works in most cases, but there are some cases where remapping the assignment can be helpful, because of the DMA channel priorities. In the default configuration, when competing for any of the system buses, DMA0 has higher priority than DMA1, and so on.

DMA Controller Overview

DMA11 has the lowest priority of the peripheral DMA channels on DMAC0. Similarly, DMA12 is the highest priority peripheral DMA channel on DMAC1, and DMA23 is the lowest.

 Memory DMA channels are present on both DMA controllers. On a per DMA controller basis, memory DMA is treated as the lowest priority. However, memory DMA channels on the higher priority DMA controller will have higher priority than the peripheral DMA channels on the lower priority DMA controller.


There are control bits in the `SYSCR` register which can change the priorities of DMAC0 and DMAC1 for L1 and for L2. For more information, see [Table 2-1 on page 2-13](#) and [Table 2-5 on page 2-22](#).

 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 on DMAC0) 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.

The twelve peripheral DMA channels in each controller work completely independently from each other. The transfer timing is controlled by the mapped peripheral.

Every DMA channel features its own 4-deep FIFO that decouples DAB activity from DCB and DEB availability. DMA interrupt and descriptor fetch timing is aligned with the memory-side (DCB/DEB side) of the FIFO. The user does, however, have an option to align interrupts with the peripheral side (DAB side) of the FIFO for transmit operations.

Refer to the SYNC bit in the DMA_x_CONFIG register for details (see “DMA Configuration (DMA_x_CONFIG and MDMA_{yy}_CONFIG) Registers” on page 5-82).

 On DMAC1, 32-bit DMA mode (WDSIZE1-0 = “b#10” in DMA_x_CONFIG) is not supported for SPORT2, SPORT3, UART2, UART3, and SPI2. However, SPORT2 and SPORT3 data word lengths can still be set to up to 32 bits.

Memory DMA

This section describes the four MDMA controllers, which provide memory-to-memory DMA transfers among the various memory spaces. These include L1 and L2 memories, as well as external synchronous/asynchronous memories.

Each MDMA controller contains a DMA FIFO used to transfer data to and from either L1, L2, or the DCB and DEB buses. MDMA0 and MDMA1 have an 8-word by 16-bit FIFO, whereas MDMA2 and MDMA3 have an 8-word by 32-bit FIFO. Typically, memory DMA is used to transfer data between external memory and internal memory. It also supports DMA from boot ROM on the DEB bus. The FIFO can also be used to hold DMA data transferred between two L1 or memory other than L1 locations or between two external memory locations.

Each MDMA controller provides two DMA channels:


- A source channel (for reading from memory)
- A destination channel (for writing to memory)

DMA Controller Overview

A memory-to-memory transfer always requires the source and the destination channel to be enabled. The four channels on each DMA controller are hardwired for DMA priorities 12 through 15. Each source/destination channel forms a “stream,” and these two streams are hardwired for DMA priorities 12 through 15:

- Priority 12: MDMA0 destination (DMAC0) or MDMA2 destination (DMAC1)
- Priority 13: MDMA0 source (DMAC0) or MDMA2 source (DMAC1)
- Priority 14: MDMA1 destination (DMAC0) or MDMA3 destination (DMAC1)
- Priority 15: MDMA1 source (DMAC0) or MDMA3 source (DMAC1)

MDMA0 takes precedence over MDMA1, and MDMA2 takes precedence over MDMA3, unless round-robin scheduling is used or priorities become urgent as programmed by the DRQ bit field in the HMDMA_CONTROL register.


 It is illegal to program a source channel for memory write or a destination channel for memory read.

The channels support 8-, 16-, and 32-bit memory DMA transfers, but both ends of MDMA0 and MDMA1 connect to 16-bit buses. Source and destination channel 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 the MDMA FIFO for a given stream are granted priority at the same time. The source DMA engine fills the FIFO, while the destination DMA engine empties it. The FIFO depth allows the burst transfers of the external access bus (EAB) and DMA access bus (DAB) 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 transfer count is the same.

To start an MDMA transfer operation, the MMRs for the source and destination channels are written, each in a manner similar to peripheral DMA.

 Note the `DMAx_CONFIG` register for the source channel must be written before the `DMAx_CONFIG` register for the destination channel. Also note that an interrupt (if enabled) is generated only upon the completion of the destination work unit, not the source work unit.

There are default sets of arbitration priorities between the different DMA controllers. These arbitration priorities are described in [“DMA-Related Buses” on page 2-17](#).

The priorities between DMAC0 and DMAC1 with respect to each other are also programmable at each of the bus interfaces (DEB to external memory, DCB to the core memory and SysBus to memory other than L1).

A peripheral DMA on either DMA controller uses a subset of its DMA controller bandwidth for a variety of reasons, including data packing/unpacking. Additionally, the fact that a peripheral runs at some fraction of the `SCLK` rate allows other peripherals to access the various DMA buses as well.

In contrast, a memory DMA channel pair on a given controller can transfer data on every `SCLK` cycle if no other DMA activity occurs on the same DMA controller. This throughput difference can cause bandwidth issues with respect to other DMA controllers. For example, memory DMAs on

DMA Controller Overview

the higher priority DMA controller will hold off transfers from peripherals on the lower priority DMA controller. This transfer holdoff is most apparent at the external memory interface.

To help with this scenario, please refer to the arbitration options in [“DMA-Related Buses” on page 2-17](#). In addition, please refer to the descriptions of the `DEB_ARB_PRIORITY`, `DEB0_URGENT`, `DEB1_URGENT`, and `DEB2_URGENT` bits in the `DDR_QUEUE` register (see [Table 6-4 on page 6-17](#)) for additional control information.

Handshaked Memory DMA Mode

Handshaked operation applies only to memory DMA channels on DMAC1.

Normally, memory DMA transfers are performed at maximum speed. Once started, data is transferred in a continuous manner until either the data count expires or the MDMA is stopped. In handshake mode, the MDMA does not transfer data automatically when enabled; it waits for an external trigger on the MDMA request input signals. The `DMAR0` input is associated with MDMA2 and the `DMAR1` input with MDMA3. Once a trigger event is detected, a programmable portion of data is transferred and then the MDMA stalls again and waits for the next trigger.

Handshake operation is not only useful to control the timing of memory-to-memory transfers, it also enables the MDMA to operate with asynchronous FIFO-style devices connected to the EBIU port. The Blackfin processor acknowledges a DMA request by a proper number of read or write operations. It is up to the device connected to any of the \overline{AMSx} strobes to deassert or pulse the request signal and to decrement the number of pending requests accordingly.

Depending on HMDMA operating mode, an external DMA request may trigger individual data word transfers or block transfers. A block can consist of up to 65535 data words. For best throughput, DMA requests can be pipelined. The HMDMA controllers feature a request counter to decouple request timing from the data transfers.

See [“Handshaked Memory DMA Operation” on page 5-46](#) for a functional description.

Modes of Operation

The following sections describe the DMA operations - register-based, two-dimensional, and descriptor-based.

Register-Based DMA Operation

Register-based DMA is the traditional kind of DMA operation. Software writes source or destination address and length of the data to be transferred into memory-mapped registers and then starts DMA operation.

For basic operation the software performs these steps:

- Write the source or destination address to the 32-bit `DMAx_START_ADDR` register.
- Write the number of data words to be transferred to the 16-bit `DMAx_X_COUNT` register.
- Write the address modifier to the 16-bit `DMAx_X_MODIFY` register. This is the two's-complement value added to the address pointer after every transfer. Typically, this register is set to 0x0004 for 32-bit DMA transfers, to 0x0002 for 16-bit transfers, and to 0x0001 for byte transfers.

Modes of Operation

- Write the operation mode to the `DMAx_CONFIG` register. These bits in particular need to be changed as needed:
 - The `DMAEN` bit enables the DMA channel.
 - The `WNR` bit controls the DMA direction. DMAs that read from memory keep this bit cleared, for example, transmitting peripheral DMAs and the source channel of memory DMAs. Receiving DMAs and the destination for memory DMAs set this bit, because they write to memory.
 - The `WDSIZE` bit controls the data word width for the transfer. It can be 8, 16, or 32 bits wide.
 - The `DI_EN` bit enables an interrupt when the DMA operation has finished.
 - Set the `FLOW` field to 0x0 for stop mode or 0x1 for autobuffer mode.

Once the `DMAEN` bit is set, the DMA channel starts its operation. While running, the `DMAx_CURR_ADDR` and the `DMAx_CURR_X_COUNT` registers can be monitored to determine the current progress of the DMA operation.

The `DMAx_IRQ_STATUS` register signals whether the DMA has finished (`DMA_DONE` bit), whether a DMA error has occurred (`DMA_ERR` bit), and whether the DMA is currently running (`DMA_RUN` bit). The `DMA_DONE` and the `DMA_ERR` bits also function as interrupt latch bits. They must be cleared by write-1-to-clear (W1C) operations by the interrupt service routine.

Stop Mode

In stop mode, the DMA operation is executed only once. If started, the DMA channel transfers the desired number of data words and stops itself again when finished. If the DMA channel is no longer used, software

clears the `DMAEN` enable bit to disable a paused channel. Stop mode is entered if the `FLOW` bit field in the DMA channel's `DMAX_CONFIG` register is 0. The `NDSIZE` field must always be 0 in this mode.

For receive (memory write) operation, the `DMA_RUN` bit functions almost the same as the inverted `DMA_DONE` bit. For transmit (memory read) operation, however, the two bits have different timing. Refer to the description of the `SYNC` bit for details.

Autobuffer Mode

In autobuffer mode, the DMA operates repeatedly in a circular manner. If all data words have been transferred, the address pointer `DMAX_CURR_ADDR` is reloaded automatically by the `DMAX_START_ADDR` value. An interrupt may also be generated.

Autobuffer mode is entered if the `FLOW` field in the `DMAX_CONFIG` register is 1. The `NDSIZE` bit must be 0 in autobuffer mode.

Two-Dimensional DMA Operation

Register-based and descriptor-based DMA can operate in one-dimensional mode or two-dimensional mode.

In two-dimensional (2D) mode the `DMAX_X_COUNT` register is accompanied by the `DMAX_Y_COUNT` register, supporting arbitrary row and column sizes up to 64K bytes x 64K bytes elements, as well as arbitrary `DMAX_X_MODIFY` and `DMAX_Y_MODIFY` values up to $\pm 32\text{K}$ bytes. Furthermore, `DMAX_Y_MODIFY` values can be negative, allowing implementation of interleaved data streams. The `DMAX_X_COUNT` and `DMAX_Y_COUNT` values specify the row and column sizes, where a `DMAX_X_COUNT` value 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 (`WDSIZE[1:0]` in `DMAX_CONFIG`). Misalignment causes a DMA error.

Modes of Operation

The `DMAX_X_MODIFY` value is the byte-address increment that is applied after each transfer that decrements the `DMAX_CURR_X_COUNT` register. The `DMAX_X_MODIFY` value is not applied when the inner loop count is ended by decrementing the `DMAX_CURR_X_COUNT` value from 1 to 0, except that it is applied on the final transfer when the `DMAX_CURR_Y_COUNT` value is 1 and `DMAX_CURR_X_COUNT` decrements from 1 to 0.

The `DMAX_Y_MODIFY` value is the byte-address increment that is applied after each decrement of the `DMAX_CURR_Y_COUNT` register. However, the `DMAX_Y_MODIFY` value is not applied to the last item in the array on which the outer loop count (`DMAX_CURR_Y_COUNT`) also expires by decrementing from 1 to 0.

After the last transfer completes, registers `DMAX_CURR_Y_COUNT = 1`, `DMAX_CURR_X_COUNT = 0`, and `DMAX_CURR_ADDR` are equal to the last item's address plus `DMAX_X_MODIFY`. Note if the DMA channel is programmed to refresh automatically (autobuffer mode), then these registers is loaded from `DMAX_X_COUNT`, `DMAX_Y_COUNT`, and `DMAX_START_ADDR` upon the first data transfer.

The `DI_SEL` configuration bit enables DMA interrupt requests every time the inner loop rolls over. If `DI_SEL` is cleared, but `DI_EN` is still set, only one interrupt is generated after the outer loop completes.

Examples of Two-Dimensional DMA

Example 1: Retrieve a 16×8 block of bytes from a video frame buffer of size ($N \times M$) pixels:

```
DMAX_X_MODIFY = 1
DMAX_X_COUNT = 16
DMAX_Y_MODIFY = N-15 (offset from the end of one row to the start
of another)
DMAX_Y_COUNT = 8
```


This produces the following code offset from the start address:

```
0, 1, 2, ... 15,
N, N + 1, ... N + 15,
2N, 2N + 1, ... 2N + 15, ...
7N, 7N + 1, ... 7N + 15,
```

Example 2: Receive a video datastream of bytes,
(R,G,B pixels) \times (N \times M image size):

```
DMAx_X_MODIFY = (N * M)
DMAx_X_COUNT = 3
DMAx_Y_MODIFY = 1 - 2(N * M) (negative)
DMAx_Y_COUNT = (N * M)
```

This produces the following code offset 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,
```

Descriptor-Based DMA Operation

In descriptor-based DMA operation, software does not set up DMA sequences by writing directly into DMA controller registers. Rather, software keeps DMA configurations, called descriptors, in memory. On demand, the DMA controller loads the descriptor from memory and overwrites the affected DMA registers by its own control. Descriptors can be fetched from L1 memory using the DCB bus, from memory other than L1, or from external memory using the DEB bus.

Modes of Operation

A descriptor describes what kind of operation should be performed next by the DMA channel. This includes the DMA configuration word as well as data source/destination address, transfer count, and address modify values. A DMA sequence controlled by one descriptor is called a work unit.

Optionally, an interrupt can be requested at the end of any work unit by setting the `DI_EN` bit in the configuration word of the respective descriptor.

A DMA channel is started in descriptor-based mode by first writing the 32-bit address of the first descriptor into the `DMAx_NEXT_DESC_PTR` register (or the `DMAx_CURR_DESC_PTR` register in case of descriptor array mode) and then performing a write to the configuration register `DMAx_CONFIG` that sets the `FLOW` field to either `0x04`, `0x6`, or `0x7` and enables the `DMAEN` bit. This causes the DMA controller to immediately fetch the descriptor from the address pointed to by the `DMAx_NEXT_DESC_PTR` register. The fetch overwrites the `DMAx_CONFIG` register again. If the `DMAEN` bit is still set, the channel starts DMA processing.

The `DFETCH` bit in the `DMAx_IRQ_STATUS` register tells whether a descriptor fetch is ongoing on the respective DMA channel, whereas the `DMAx_CURR_DESC_PTR` register points to the descriptor value that is to be fetched next.

Descriptor List Mode

Descriptor list mode is selected by setting the `FLOW` bit field in the DMA channel's `DMAx_CONFIG` register to either `0x6` (small descriptor mode) or `0x7` (large descriptor mode). In this mode multiple descriptors form a chained list. Every descriptor contains a pointer to the next descriptor. When the descriptor is fetched, this pointer value is loaded into the `DMAx_NEXT_DESC_PTR` register of the DMA channel. In large descriptor mode this pointer is 32 bits wide. Therefore, the next descriptor may reside in any address space accessible through the DCB and DEB buses. In small descriptor mode this pointer is just 16 bits wide. For this reason, the

next descriptor must reside in the same 64K byte address space as the first one, because the upper 16 bits of the `DMAx_NEXT_DESC_PTR` register are not updated.

Descriptor list modes are started by writing first to the `DMAx_NEXT_DESC_PTR` register and then to the `DMAx_CONFIG` register.

Descriptor Array Mode

Descriptor array mode is selected by setting the `FLOW` bit field in the DMA channel's `DMAx_CONFIG` register to `0x4`. In this mode, the descriptors do not contain further descriptor pointers. The initial `DMAx_CURR_DESC_PTR` value is written by software. It points to an array of descriptors. The individual descriptors are assumed to reside next to each other and, therefore, their address is known.

Variable Descriptor Size

In any descriptor-based mode, the `NDSIZE` field in the configuration word specifies how many 16-bit words of the next descriptor need to be loaded on the next fetch. In descriptor-based operation, `NDSIZE` field must be nonzero. The descriptor size can be any value from one entry (the lower 16 bits of `DMAx_START_ADDR` register only) to nine entries (all the DMA parameters). [Table 5-2](#) illustrates how a descriptor must be structured in memory. The values have the same order as the corresponding MMR addresses.

If, for example, a descriptor is fetched in array mode with `NDSIZE = 0x5`, the DMA controller fetches the 32-bit start address, the DMA configuration word and the `XCNT` and `XMOD` values. However, it does not load the `YCNT` and `YMOD` values. This might be the case if the DMA operates in one-dimensional mode or if the DMA is in two-dimensional mode, but the `YCNT` and `YMOD` values do not need to change.

Modes of Operation

All the other registers not loaded from the descriptor retain their prior values, although the `DMAx_CURR_ADDR`, `DMAx_CURR_X_COUNT`, and `DMAx_CURR_Y_COUNT` registers are reloaded between the descriptor fetch and the start of DMA operation.

Table 5-2 shows the offsets for descriptor elements in the three modes described above. Note the names in the table describe the descriptor elements in memory, not the actual MMRs into which they are eventually loaded.

Table 5-2. Parameter Registers and Descriptor Offsets

Descriptor Offset	Descriptor Array Mode	Small Descriptor List Mode	Large Descriptor List Mode
0x0	SAL	NDPL	NDPL
0x2	SAH	SAL	NDPH
0x4	DMACFG	SAH	SAL
0x6	XCNT	DMACFG	SAH
0x8	XMOD	XCNT	DMACFG
0xA	YCNT	XMOD	XCNT
0xC	YMOD	YCNT	XMOD
0xE		YMOD	YCNT
0x10			YMOD



Every descriptor fetch consumes bandwidth on either the DCB bus or DEB bus and the external memory interface, so it is best to keep the size of descriptors as small as possible.

Mixing Flow Modes

The `FLOW` mode of a DMA is not a global setting. If the DMA configuration word is reloaded with a descriptor fetch, the `FLOW` and `NDSIZE` bit fields can also be altered. A small descriptor might be used to loop back to the first descriptor if a descriptor array is used in an endless manner.

If the descriptor chain is not endless and the DMA is required to stop after a certain descriptor is processed, the last descriptor is typically processed in stop mode, that is, its `FLOW` and `NDSIZE` fields are 0, but its `DMAEN` bit is still set.

Functional Description

The following sections provide a functional description of DMA - operation flow, errors, control commands, handshaked memory and performance.

DMA Operation Flow

[Figure 5-3](#) and [Figure 5-4](#) describe the DMA flow.

DMA Startup

This section discusses starting DMA “from scratch.” This is similar to starting it after it is 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 the `DMAx_NEXT_DESC_PTR` and `DMAx_START_ADDR` registers, because they might not otherwise be accessed, depending on the chosen `FLOW` mode of operation. Also note that the `DMAx_X_MODIFY` and `DMAx_Y_MODIFY` are not preset to a default value at reset.

Functional Description

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 `DMAx_NEXT_DESC_PTR` register (or `DMAx_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, `DMAx_X_MODIFY` and `DMAx_Y_MODIFY`). The contents of `NDSIZE` and `FLOW` in the `DMAx_CONFIG` register 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 `DMAx_CONFIG` with `DMAEN = 1`.

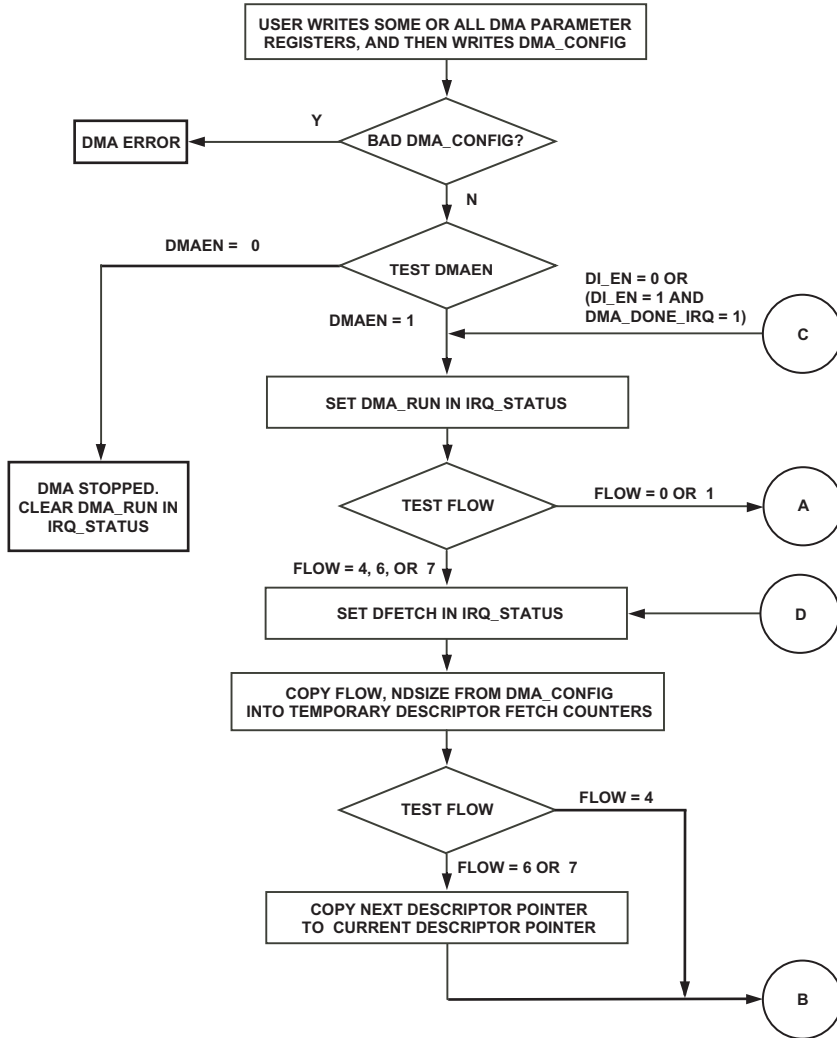


Figure 5-3. DMA Flow, From DMA Controller's Point of View (1 of 2)

Functional Description

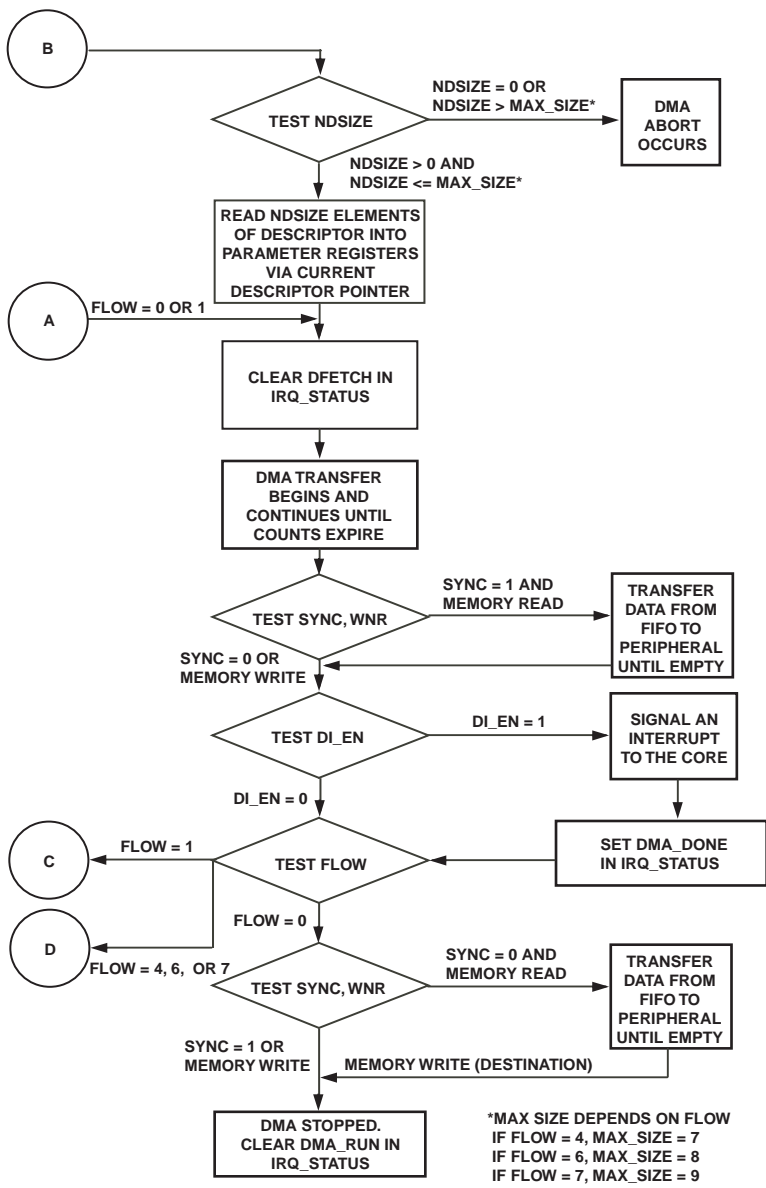


Figure 5-4. DMA Flow, From DMA Controller's Point of View (2 of 2)

When the `DMAx_CONFIG` register is written directly by software, 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 is stopped (`FLOW = 0`).

When the descriptor fetch is complete and `DMAEN = 1`, the `DMACFG` descriptor element that was read into the `DMAx_CONFIG` register assumes control. Before this point, the direct write to `DMAx_CONFIG` register had control. In other words, the `WDSIZE`, `DI_EN`, `DI_SEL`, `SYNC`, and `DMA2D` fields are taken from the `DMACFG` value in the descriptor read from memory, while these field values initially written to the `DMAx_CONFIG` register are ignored.

As [Figure 5-3 on page 5-27](#) and [Figure 5-4 on page 5-28](#) show, at startup, the `FLOW` and `NDSIZE` bits in `DMAx_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.

If the `FLOW` value specifies small or large descriptor list modes, the `DMAx_NEXT_DESC_PTR` register is copied into `DMAx_CURR_DESC_PTR` register. Then, fetches of new descriptor elements from memory are performed, indexed by `DMAx_CURR_DESC_PTR` register, which is incremented after each fetch. If `NDPL` and/or `NDPH` is part of the descriptor, then these values are loaded into `DMAx_NEXT_DESC_PTR` register, but the fetch of the current descriptor continues using `DMAx_CURR_DESC_PTR` register. After completion of the descriptor fetch, `DMAx_CURR_DESC_PTR` register 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 `DMAx_CURR_DESC_PTR` register does not occur. Instead, descriptor fetch indexing begins with the value in `DMAx_CURR_DESC_PTR` register.

Functional Description

If `DMACFG` is not part of the descriptor, the previous `DMAX_CONFIG` settings (as written by MMR access at startup) control the work unit operation. If `DMACFG` register is part of the descriptor, then the `DMAX_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's `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 `DMAX_IRQ_STATUS` register indicate the state of the DMA channel. After a write to `DMAX_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, DMA data transfer operation begins. The DMA channel immediately attempts to fill its FIFO, subject to channel priority—a memory write (RX) DMA channel begins accepting data from its peripheral, and a memory read (TX) DMA channel begins memory reads, provided the channel wins the grant for bus access.

When the DMA channel performs its first data memory access, its address and count computations take their input operands from the start registers (`DMAX_START_ADDR`, `DMAX_X_COUNT`, `DMAX_Y_COUNT`), and write results back to the current registers (`DMAX_CURR_ADDR`, `DMAX_CURR_X_COUNT`, `DMAX_CURR_Y_COUNT`). Note also that the current registers are not valid until the first memory access is performed, which may be some time after the channel is started by the write to the `DMAX_CONFIG` register. The current registers are loaded automatically from the appropriate descriptor elements, overwriting their previous contents, as follows:

- `DMAX_START_ADDR` is copied to `DMAX_CURR_ADDR`
- `DMAX_X_COUNT` is copied to `DMAX_CURR_X_COUNT`
- `DMAX_Y_COUNT` is copied to `DMAX_CURR_Y_COUNT`

Then DMA data transfer operation begins, as shown in [Figure 5-4 on page 5-28](#).

DMA Refresh

On completion of a work unit, the DMA controller:

- Completes the transfer of all data between memory and the DMA unit.
- If `SYNC = 1` and `WNR = 0` (memory read). Selects a synchronized transition. Transfers all data to the peripheral before continuing.
- If enabled by `DI_EN`, signals an interrupt to the core and sets the `DMA_DONE` bit in the channel's `DMAx_IRQ_STATUS` register.
- If `FLOW = 0` (stop) only. Stops operation by clearing the `DMA_RUN` bit in `DMAx_IRQ_STATUS` after any data in the channel's DMA FIFO is transferred to the peripheral.
- During the fetch in `FLOW` modes 4, 6, and 7, the DMA controller sets the `DFETCH` bit in `DMAx_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), then loads a new descriptor from memory into DMA registers by way of the contents of `DMAx_CURR_DESC_PTR`, while incrementing `DMAx_CURR_DESC_PTR`. The descriptor size comes from the `NDSIZE` field of the `DMAx_CONFIG` value prior to the beginning of the fetch.

If `FLOW = 6` (descriptor list small), then copies the 32-bit `DMAx_NEXT_DESC_PTR` into `DMAx_CURR_DESC_PTR`. Next, fetches a descriptor from memory into DMA registers through the new contents of `DMAx_CURR_DESC_PTR`, while incrementing `DMAx_CURR_DESC_PTR`. The first descriptor element loaded is a new 16-bit value for the lower 16 bits of `DMAx_NEXT_DESC_PTR`, followed

Functional Description

by the rest of the descriptor elements. The high 16 bits of `DMAx_NEXT_DESC_PTR` will 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 64K byte range of memory.

if `FLOW = 7` (descriptor list large), then copies the 32-bit `DMAx_NEXT_DESC_PTR` into `DMAx_CURR_DESC_PTR`. Next, fetches a descriptor from memory into DMA registers through the new contents of `DMAx_CURR_DESC_PTR`, while incrementing `DMAx_CURR_DESC_PTR`. The first descriptor element loaded is a new 32-bit value for the full `DMAx_NEXT_DESC_PTR`, followed by the rest of the descriptor elements. The high 16 bits of `DMAx_NEXT_DESC_PTR` may differ from their former value. This supports a fully flexible descriptor list which can be located anywhere in internal or external memory.

Note if it is necessary to link from a descriptor chain whose descriptors are in one 64K byte 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 64K byte range. All the other descriptors located together in the same 64K byte areas may use `FLOW = 6`.

- If `FLOW = 4, 6, or 7` (descriptor array, descriptor list small, or descriptor list large, respectively), then the DMA controller clears the `DFETCH` bit in the `DMAX_IRQ_STATUS` register.
- If `FLOW = any value but 0 (Stop)`, then the DMA controller begins the next work unit, contending with other channels for priority on the memory buses. On the first memory transfer of the new work unit, the DMA controller updates the current registers from the start registers:

`DMAX_CURR_ADDR` loaded from `DMAX_START_ADDR`
`DMAX_CURR_X_COUNT` loaded from `DMAX_X_COUNT`
`DMAX_CURR_Y_COUNT` loaded from `DMAX_Y_COUNT`

The `DFETCH` bit in `DMAX_IRQ_STATUS` is then cleared, after which the DMA transfer begins again, as shown in [Figure 5-4 on page 5-28](#).


Work Unit Transitions

Transitions from one work unit to the next are controlled by the `SYNC` bit in the `DMAX_CONFIG` register of the work units. In general, continuous transitions have lower latency at the cost of restrictions on changes of data format or addressed memory space in the two work units. These latency gains and data restrictions arise from the way the DMA FIFO pipeline is handled while the next descriptor is fetched. In continuous transitions (`SYNC = 0`), the DMA FIFO pipeline continues to transfer data to and from the peripheral or destination memory during the descriptor fetch and/or when the DMA channel is paused between descriptor chains.

Synchronized transitions (`SYNC = 1`), on the other hand, provide better real-time synchronization of interrupts with peripheral state and greater flexibility in the data formats and memory spaces of the two work units, at

Functional Description

the cost of higher latency in the transition. In synchronized transitions, the DMA FIFO pipeline is drained to the destination or flushed (RX data discarded) between work units.

 Work unit transitions for MDMA streams are controlled by the `SYNC` bit of the MDMA source channel's `DMAx_CONFIG` register. The `SYNC` bit of the MDMA destination channel is reserved and must be 0. In transmit (memory read) channels, the `SYNC` bit of the last descriptor prior to the transition controls the transition behavior. In contrast, in receive channels, the `SYNC` bit of the first descriptor of the next descriptor chain controls the transition.

DMA Transmit and MDMA Source

In DMA transmit (memory read) and MDMA source channels, the `SYNC` bit controls the interrupt timing at the end of the work unit and the handling of the DMA FIFO between the current and next work unit.

If `SYNC = 0`, a continuous transition is selected. In a continuous transition, just after the last data item is read from memory, these four operations all start in parallel:

- The interrupt (if any) is signalled.
- The `DMA_DONE` bit in the `DMAx_IRQ_STATUS` register is set.
- The next descriptor begins to be fetched.
- The final data items are delivered from the DMA FIFO to the destination memory or peripheral.

This allows the DMA to provide data from the FIFO to the peripheral “continuously” during the descriptor fetch latency period.

When `SYNC = 0`, the final interrupt (if enabled) occurs when the last data is read from memory. This interrupt is at the earliest time that the output memory buffer may safely be modified without affecting the previous data transmission. Up to four data items may still be in the DMA FIFO,

however, and not yet at the peripheral, so the DMA interrupt should not be used as the sole means of synchronizing the shutdown or reconfiguration of the peripheral following a transmission.

i If `SYNC = 0` (continuous transition) on a transmit (memory read) descriptor, the next descriptor is required to have the same data word size, read/write direction, and source memory (internal versus external) as the current descriptor.

If `SYNC = 0` selects continuous transition on a work unit in `FLOW = STOP` mode with interrupt enabled, the interrupt service routine may already run while the final data is still draining from the FIFO to the peripheral. This is indicated by the `DMA_RUN` bit in the `DMAx_IRQ_STATUS` register; if it is 1, the FIFO is not empty yet. Do not start a new work unit with different word size or direction while `DMA_RUN = 1`. Further, if the channel is disabled (by writing `DMAEN = 0`), the data in the FIFO is lost.

If `SYNC = 1`, a synchronized transition is selected, in which the DMA FIFO is first drained to the destination memory or peripheral before any interrupt is signalled and before any subsequent descriptor or data is fetched. This incurs greater latency, but provides direct synchronization between the DMA interrupt and the state of the data at the peripheral.

For example, if `SYNC = 1` and `DI_EN = 1` on the last descriptor in a work unit, the interrupt occurs when the final data is transferred to the peripheral, allowing the service routine to properly switch to non-DMA transmit operation. When the interrupt service routine is invoked, the `DMA_DONE` bit is set and the `DMA_RUN` bit is cleared.


A synchronized transition also allows greater flexibility in the format of the DMA descriptor chain. If `SYNC = 1`, the next descriptor may have any word size or read/write direction supported by the peripheral and may come from either memory space (internal as opposed to external). This can be useful in managing MDMA work unit queues, since it is no longer necessary to interrupt the queue between dissimilar work units.

Functional Description

DMA Receive

In DMA receive (memory write) channels, the `SYNC` bit controls the handling of the DMA FIFO between descriptor chains (not individual descriptors), when the DMA channel is paused. The DMA channel pauses after descriptors with `FLOW = STOP` mode, and may be restarted (for example, after an interrupt) by writing the channel's `DMAx_CONFIG` register with `DMAEN = 1`.


If the `SYNC` bit is 0 in the new work unit's `DMAx_CONFIG` value, a continuous transition is selected. In this mode, any data items received into the DMA FIFO while the channel was paused are retained, and they are the first items written to memory in the new work unit. This mode of operation provides lower latency at work unit transitions and ensures that no data items are dropped during a DMA pause, at the cost of certain restrictions on the DMA descriptors.

 If the `SYNC` bit is 0 on the first descriptor of a descriptor chain after a DMA pause, the DMA word size of the new chain must not change from the word size of the previous descriptor chain active before the pause, unless the DMA channel is reset between chains by writing the `DMAEN` bit to 0 and then to 1.

If the `SYNC` bit is 1 in the new work unit's `DMAx_CONFIG` value, a synchronized transition is selected. In this mode, only the data received from the peripheral by the DMA channel after the write to the `DMAx_CONFIG` register is delivered to memory. Any prior data items transferred from the peripheral to the DMA FIFO before this register write are discarded. This provides direct synchronization between the data stream received from the peripheral and the timing of the channel restart (when the `DMAx_CONFIG` register is written).

For receive DMAs, the `SYNC` bit has no effect in transitions between work units in the same descriptor chain (that is, when the previous descriptor's `FLOW` mode was not `STOP`, so that DMA channel did not pause).


If a descriptor chain begins with a SYNC bit of 1, there is no restriction on DMA word size of the new chain in comparison to the previous chain.

-  The DMA word size must not change between one descriptor and the next in any DMA receive (memory write) channel within a single descriptor chain, regardless of the SYNC bit setting. In other words, if a descriptor has WNR = 1 and FLOW = 4, 6, or 7, then the next descriptor must have the same word size. For any DMA receive (memory write) channel, there is no restriction on changes of memory space (internal versus external) between descriptors or descriptor chains. DMA transmit (memory read) channels may have such restrictions (see [“DMA Transmit and MDMA Source” on page 5-34](#)).

Stopping 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 autobuffer (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 DMAEN bit is 0. A write of 0 to the entire register always terminates DMA gracefully (without DMA abort).

-  If a channel is stopped abruptly by writing DMAx_CONFIG to 0 (or any value with DMAEN = 0), the user must ensure that any memory read or write accesses in the pipelines have completed before enabling the channel again. If the channel is enabled again before an “orphan” access from a previous work unit completes, the state of the DMA interrupt and FIFO is unspecified. This can generally be handled by ensuring that the core allocates several idle cycles in

Functional Description

a row in its usage of the relevant memory space to allow up to three pending DMA accesses to issue, plus allowing enough memory access time for the accesses themselves to complete.

DMA Errors (Aborts)

The DMA controllers flag conditions that cause DMA processes to end abnormally (that is, abort). This functionality is provided as a tool for system development and debug, as a way to detect DMA-related programming errors. DMA errors (aborts) are detected by the DMA channel module in the cases listed below. When a DMA error occurs, the channel is immediately stopped (`DMA_RUN` goes to 0) and any prefetched data is discarded. In addition, a `DMA_ERROR` interrupt is asserted.

There is only one `DMA_ERROR` interrupt for a DMA controller, which is asserted whenever any of the channels has detected an error condition.

The `DMA_ERROR` interrupt handler must do these things for each channel:

- Read each channel's `DMAX_IRQ_STATUS` register to look for a channel with the `DMA_ERR` bit set (bit 1).
- Clear the problem with that channel (for example, fix register values).
- Clear the `DMA_ERR` bit (write `DMAX_IRQ_STATUS` with bit 1 = 1).

The following error conditions are detected by the DMA hardware and result in a DMA abort interrupt.

- The configuration register contains invalid values:
 - Incorrect `WDSIZE` value (`WDSIZE = b#11`)
 - Bit 15 not set to 0

- Incorrect FLOW value (FLOW = 2, 3, or 5)
- NDSIZE value does not agree with FLOW.
See [Table 5-3 on page 5-40](#).
- A disallowed register write occurred while the channel was running. Only the DMA_x_CONFIG and DMA_x_IRQ_STATUS registers can be written when DMA_RUN = 1.
- An address alignment error occurred during any memory access. For example, DMA_x_CONFIG register WDSIZE = 1 (16 bit) but the least significant bit (LSB) of the address is not equal to 0, or WDSIZE = 2 (32 bit) but the two LSBs of the address are not equal to b#00.
- A memory space transition was attempted (internal-to-external or vice versa). For example, the current DMA address (DMA_x_CURR_ADDR) crossed the 0xF000 0000 boundary, or the current descriptor pointer (DMA_x_CURR_DESC_PTR) crossed the 0xF000 0000 boundary.
- 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 (signaled by the external memory interface).

Some prohibited situations are not detected by the DMA hardware. No DMA abort is signaled for these situations:

- DMA_x_CONFIG direction bit (WNR) does not agree with the direction of the mapped peripheral.
- DMA_x_CONFIG direction bit does not agree with the direction of the MDMA channel.
- DMA_x_CONFIG word size (WDSIZE) is not supported by the mapped peripheral.

Functional Description

- `DMAx_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_COUNT = 1`.

Table 5-3. Legal NDSIZE Values

FLOW	NDSIZE	Note
0	0	
1	0	
4	$0 < \text{NDSIZE} \leq 7$	Descriptor array, no descriptor pointer fetched
6	$0 < \text{NDSIZE} \leq 8$	Descriptor list, small descriptor pointer fetched
7	$0 < \text{NDSIZE} \leq 9$	Descriptor list, large descriptor pointer fetched

DMA Control Commands

Advanced peripherals on the processor, such as the USB and MXVR port are capable of managing some of their own DMA operations, thus dramatically improving real-time performance and relieving control and interrupt demands on the Blackfin processor core. These peripherals may communicate to the DMA controllers using DMA control commands, which are transmitted from the peripheral to the associated DMA channel over internal DMA request buses. These request buses consist of three wires per DMA-management-capable peripheral. The DMA control commands extend the set of operations available to the peripheral beyond the simple “request data” command used by peripherals in general. Refer to the appropriate peripheral chapter for a description on how that peripheral uses DMA control commands.

Note that while these DMA control commands are not visible to or controlled by the user, their use by a peripheral has implications for the structure of the DMA transfers which that peripheral can support. It is important that application software be written to comply with certain restrictions regarding work units and descriptor chains (described later in this section) so that the peripheral operates properly whenever it issues DMA control commands.

MDMA channels do not service peripherals and therefore do not support DMA control commands.

The DMA control commands are shown in [Table 5-4](#).

Table 5-4. DMA Control Commands

Code	Name	Description
b#000	NOP	No operation
b#001	Restart	Restarts the current work unit from the beginning
b#010	Finish	Finishes the current work unit and starts the next
b#011	Interrupt	Immediately sets the DMA completion interrupt in the associated DMA peripheral channel
b#100	Request Data	Typical DMA data request
b#101	Request Data Urgent	Urgent DMA data request
b#110	Request Register Load	Request/continue transfer of DMA channel control register values by way of DAB.
b#111	-	Reserved

Functional Description

Additional information for the control commands includes:

- **Restart**

The restart control command causes the current work unit to interrupt processing and start over, using the addresses and counts from `DMAx_START_ADDR`, `DMAx_X_COUNT`, and `DMAx_Y_COUNT`. No interrupt is signalled.

If a channel programmed for transmit (memory read) receives a restart control command, the channel momentarily pauses while any pending memory reads initiated prior to the restart command are completed.

During this period of time, the channel does not grant DMA requests. Once all pending reads have been flushed from the channel's pipelines, the channel resets its counters and FIFO, and starts prefetch reads from memory. DMA data requests from the peripheral are granted as soon as new prefetched data is available in the DMA FIFO. The peripheral can then use the restart command to reattempt a failed transmission of a work unit.

If a channel programmed for receive (memory write) receives a restart control command, the channel stops writing to memory, discards any data held in its DMA FIFO, and resets its counters and FIFO. As soon as this initialization is complete, the channel again grants DMA write requests from the peripheral. The peripheral can thus use the restart command to abort transfer of received data into a work unit, and reuse the memory buffer for a later data transfer.

- **Finish**

The finish control command causes the current work unit to terminate processing and move on to the next. An interrupt is signalled as usual, if selected by the `DI_EN` bit. The peripheral can thus use the finish command to partition the DMA stream into work units

on its own, perhaps as a result of parsing the data currently passing through its supported communication channel, without direct real-time control by the processor.

If a channel programmed for transmit (memory read) receives a finish control command, the channel momentarily pauses while any pending memory reads initiated prior to the finish command are completed. During this period of time, the channel does not grant DMA requests. Once all pending reads are flushed from the channel's pipelines, the channel signals an interrupt (if enabled), and begins fetching the next descriptor (if any). DMA data requests from the peripheral are granted as soon as new prefetched data is available in the DMA FIFO.

If a channel programmed for receive (memory write) receives a finish control command, the channel stops granting new DMA requests while it drains its FIFO. Any DMA data received by the DMA controller prior to the finish command is written to memory. When the FIFO reaches an empty state, the channel signals an interrupt (if enabled) and begins fetching the next descriptor (if any). Once the next descriptor is fetched, the channel initializes its FIFO, and then resumes granting DMA requests from the peripheral.

- **Interrupt**

This command immediately sets the DMA completion interrupt in the `DMAX_IRQ_STATUS` register of the associated DMA peripheral channel.

- **Request Data**

The request data control command is identical to the DMA request operation of peripherals which are not DMA-management-capable.

Functional Description

- **Request Data Urgent**

The request data urgent control command behaves identically to the DMA request control command, except that while it is asserted the DMA channel performs its memory accesses with urgent priority. This includes both data and descriptor-fetch memory accesses. A DMA-management-capable peripheral might use this control command if an internal FIFO is approaching a critical condition, for example.

- **Request Register Load**

This command pertains exclusively to the HOSTDTP on DMA14 peripheral on the ADSP-BF54x Blackfin processor. The command allows a “DAB-mastering” peripheral to load values directly into its DMA channel control registers by way of the DAB bus. Refer to [Chapter 8, “Host DMA Port”](#) for more information on how this command is used in conjunction with Host DMA port operation.

The DMA channel must be enabled (`DMAx_CONFIG` register’s `DMA_EN` bit =1) to use the request register load command to be used. This command cannot be used to enable a disabled channel, nor may it be used to write the channel’s next descriptor pointer.

On the first (non-granted) cycle when the peripheral does not assert request register load, the DMA channel will cease loading register values and initiates processing the work unit they specify.

The DMA channel FIFO is not reinitialized when processing begins. Therefore, any transmit or receive data present in the FIFO remains in place, unless otherwise configured by the `DMAx_CONFIG` register’s `SYNC` bit (bit 5).

Restrictions

The proper operation of the 4-location DMA channel FIFO leads to certain restrictions in the sequence of DMA control commands.

Transmit Restart or Finish

No restart or finish control command may be issued by a peripheral to a channel configured for memory read unless both (a) the peripheral has already performed at least one DMA transfer in the current work unit, and (b) the current work unit has more than four items remaining in $\text{DMAx_CURR_X_COUNT}/\text{DMAx_CURR_Y_COUNT}$ (thus not yet read from memory.) Otherwise, the current work unit may already have completed memory operations and can no longer be restarted or finished properly.

If the $\text{DMAx_CURR_X_COUNT}/\text{DMAx_CURR_Y_COUNT}$ of the current work unit is sufficiently large that it is always at least five more than the maximum data count prior to any restart or finish command, the above restriction is satisfied. This implies that any work unit which might be managed by restart or finish commands must have $\text{DMAx_CURR_X_COUNT}/\text{DMAx_CURR_Y_COUNT}$ values representing at least five data items.

Note in particular that if the $\text{DMAx_CURR_X_COUNT}/\text{DMAx_CURR_Y_COUNT}$ registers are programmed to 0 (representing 65,536 transfers, the maximum value) the channel operates properly for 1D work units up to 65,531 data items or 2D work units up to 4,294,967,291 data items.

Receive Restart or Finish

No restart or finish control command may be issued by a peripheral to a channel configured for memory write unless either (a) the peripheral already performed at least five DMA transfers in the current work unit, or (b) the previous work unit terminated by a finish control command and the peripheral performed at least one DMA transfer in the current work unit. If five data transfers performed, then at least one data item is written to memory in the current work unit, which implies that the current work unit's descriptor fetch completed before the data grant of the fifth item. Otherwise, if less than five data items are transferred, it is possible that all of them are still in the DMA FIFO and that the previous work unit is still in the process of completion and transition between work units.

Functional Description

Similarly, if a finish command ended the previous work unit and at least one subsequent DMA data transfer occurred, then the fact that the DMA channel issued the grant guarantees that the previous work unit already completed the process of draining its data to memory and transitioning to the new work unit.

Note that if a peripheral terminates all work units with the finish opcode (effectively assuming responsibility for all work unit boundaries for the DMA channel), then the peripheral need only ensure that it performs a single transfer in each work unit before any restart or finish. This requires, however, that the user programs the descriptors for all work units managed by the channel with `DMAX_CURR_X_COUNT/ DMAX_CURR_Y_COUNTS` representing more data items than the maximum work unit size that the peripheral encounters. For example, `DMAX_CURR_X_COUNT/ DMAX_CURR_Y_COUNTS` of 0 allow the channel to operate properly on 1D work units up to 65,535 data items and 2D work units up to 4,294,967,295 data items.

Handshaked Memory DMA Operation


Both `DMARx` inputs have their own set of control and status registers. Handshake operation for MDMA2 is enabled by the `HMDMAEN` bit in the `HMDMA0_CONTROL` register. Similarly, the `HMDMAEN` bit in the `HMDMA1_CONTROL` register enables handshake mode for MDMA3.

It is important to understand that the handshake hardware works completely independent from the descriptor and autobuffer capabilities of the MDMA, allowing most flexible combinations of logical data organization versus data portioning as required by FIFO deeps, for example. If, however, the connected device requires certain behavior of the address lines, these must be controlled by traditional DMA setup.

Because source and destination channels of a MDMA stream are decoupled by an 8-depth FIFO, they are loosely synchronized to each other. The `DMARx` functionality requires strong synchronization. So, the

HMDMA optionally can be tied to either the destination channel, as by default, or to the source channel, when the `SND` (“source not destination”) bit in the `HMDMA_CONTROLx` register is set. When data is transferred from on-chip memory to off-chip space, one may expect the `HMDMAx` block to control the destination channel. When data is transferred from off-chip space to on-chip space, the `HMDMAx` block should be control the source channel.

The `HMDMAx_BCINIT` registers control how many data transfers are performed upon every DMA request. If set to 1, the peripheral can time every individual data transfer. If greater than 1, the peripheral must feature sufficient buffer size to provide or consume the number of words programmed. Once the transfer is requested, no further handshake can hold off the DMA from transferring the entire block, except by stalling the EBIU accesses by the `ARDY` signal or a complete bus request and grant cycle through the `BR` and `BG` pins. Nevertheless, the peripheral may request a block transfer before the entire buffer is available, by simply taking the minimum transfer time based on wait-state settings into consideration.

 The block count defines how many data transfers are performed by the MDMA engine. A single DMA transfer can cause two read or write operations on the EBIU port if the transfer word size is set to 32 bit in the `MDMA_yy_CONFIG` register (`WDSIZE = b#10`).

Since the block count registers are 16 bits wide, blocks can group up to 65535 transfers.

Once a block transfer is started, the `HMDMAx_BCOUNT` registers return the remaining number of transfers to complete the current block. When the complete block is processed, the `HMDMAx_BCOUNT` register returns zero. Software can force a reload of the `HMDMAx_BCOUNT` from the `HMDMAx_BCINIT` register even during normal operation by writing a 1 to the `RBC` bit in the `HMDMAx_CONTROL` register. Set `RBC` only when the HMDMA module is already active, but the MDMA is not enabled.

Functional Description

Pipelining DMA Requests

The device mastering the DMA request lines is allowed to request additional transfers even before the former transfer has completed. As long as the device can provide or consume sufficient data, it is permitted to pulse the `DMARx` inputs multiple times.

The `HMDMAX_ECOUNT` registers are incremented every time a significant edge is detected on the respective `DMARx` input and are decremented when the MDMA completes the block transfer. These read-only registers use a 16-bit, two's-complement data representation: if they return zero, all requested block transfers have been performed. A positive value signals up to 32767 requests that have not been served yet and indicates that the MDMA is currently processing. Negative values indicate the number of DMA requests ignored by the engine. This feature restrains initial pulses on the `DMARx` inputs at startup.

The `HMDMAX_ECINIT` registers reload the `HMDMAX_ECOUNT` registers every time the handshake mode is enabled, that is, when the `HMDMAEN` bit changes from 0 to 1. If the initial edge count value is 0, the handshake operation starts with a settled request budget. If positive, the engine starts immediately transferring the programmed number (up to 32767) of blocks once enabled, even without detecting any activity on the `DMARx` pins. If negative, the engine disregards the programmed number (up to 32768) significant edges on the `DMARx` inputs before starting normal operation.

Figure 5-5 illustrates how an asynchronous FIFO could be connected. In such a scenario, the `REP` bit is cleared to let the `DMARx` request pin listen to falling edges. The Blackfin processor does not evaluate the full flag such FIFOs usually provide, because asynchronous polling of that signal reduces the system throughput drastically. Moreover, the processor first fills the FIFO by initializing the `HMDMAX_ECINIT` register by the value 1024 which equals the depth of the FIFO. Once enabled, the MDMA automatically transmits 1024 data words. Afterward it continues to transmit only if the FIFO is emptied by its read strobe again.

Most likely, the `HMDMAx_BCINIT` register is programmed to be 1 in this case. In this example, it is recommended to keep the `SND` bit cleared, so that the `HMDMAx` block controls the destination channel of the `MDMA`.

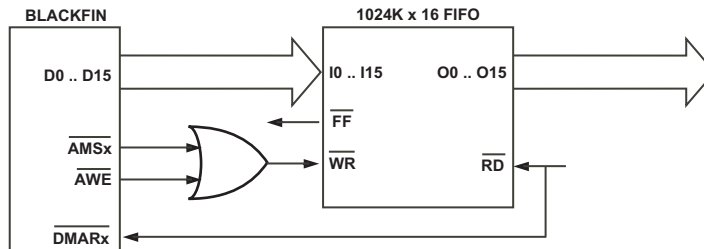


Figure 5-5. Transmit DMA Example Connection

In the receive example shown in [Figure 5-6](#), the Blackfin processor again does not use the FIFO's internal control mechanism. Rather than testing the empty flag, the processor counts the number of data words available in the FIFO by its own `HMDMAx_ECOUNT` register. Theoretically, the `MDMA` could immediately process data as soon as it is written into the FIFO by the write strobe, but the fast `MDMA` engine would read out the FIFO quickly and stall soon if the FIFO was not filled with new data promptly. Streaming applications can balance the FIFO so that the producer is never held off by a full FIFO and the consumer is never held by an empty FIFO. This is accomplished by filling the FIFO half way and then letting both consumer and producer run at the same speed. In this case, the `HMDMAx_ECINIT` register can be written with a negative value, which corresponds to half the FIFO depth. Then, the `MDMA` does not start consuming data as long as the FIFO is not half filled.

Functional Description

In this example, it is recommended to set the `SND` bit, so that the `HMDMAx` block controls the source channel of the MDMA.

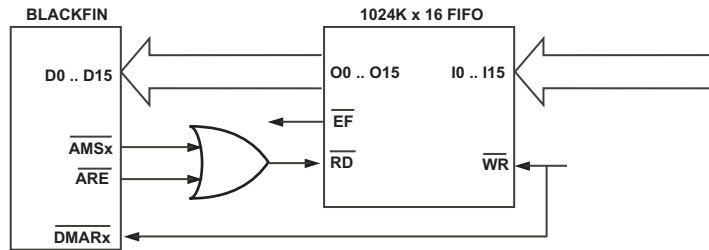


Figure 5-6. Receive DMA Example Connection

On internal system buses, memory DMA channels have lower priority than other DMAs. In busy systems it might happen that the memory DMAs tend to starve. As this is not acceptable when transferring data through high-speed FIFOs, the handshake mode provides a high-water functionality to increase the MDMA's priority. With the `UTE` bit in the `HMDMAx_CONTROL` register set, the MDMA gets higher priority as soon as a (positive) value in the `HMDMAx_ECOUNT` register becomes higher than the threshold held by the `HMDMAx_ECURGENT` register.

HMDMA Interrupts

In addition to the normal MDMA interrupt channels, the handshake hardware provides two new interrupt sources for each `DMARx` input. All interrupt sources (`DMAR0` and `DMAR1` block done, `DMAR0` and `DMAR1` overflow error) are routed to Peripheral Interrupt ID#63. Refer to [Chapter 4, "System Interrupts"](#) for more information. The `HMDMAx_CONTROL` registers provide interrupt enable and status bits. The interrupt status bits require a write-1-to-clear operation to cancel the interrupt request.

The interrupt “block done” signals that a complete MDMA block (as defined by the `HMDMAX_BCINIT` register) is transferred, that is, when the `HMDMAX_BCOUNT` register decrements to zero. While the `BDIE` bit enables this interrupt, the `MBDI` bit can gate it until the edge count also becomes zero, meaning that all requested MDMA transfers are now complete.

The overflow interrupt is generated when the `HMDMA_ECOUNTER` register overflows. Since it can count up to 32767, which is much more than most peripheral devices can support, the Blackfin processor features another threshold register called `HMDMA_ECOVERFLOW`. It resets to `0xFFFF` and is written with any positive value by the user before enabling the function by the `OIE` bit. Then, the overflow interrupt is issued when the value of the `HMDMA_ECOUNTER` register exceeds the threshold in the `HMDMA_ECOVERFLOW` register.

DMA Performance

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.

The Blackfin architecture features several mechanisms to customize system behavior for best performance. This includes DMA channel prioritization, traffic control, and priority treatment of bursted transfers. Nevertheless, the resulting performance of a DMA transfer often depends on application-level circumstances. For best performance, consider these questions when designing the system software:

- What is the required DMA bandwidth?
- Which DMA transfers have real-time requirements and which do not?
- How heavily is the DMA controller competing with the core for on-chip and off-chip resources?

Functional Description

- How often do competing DMA channels require the bus systems to alter direction?
- How often do competing DMA or core accesses cause the DDR SDRAM to open different pages?
- Is there a way to distribute DMA requests smoothly over time?

A key feature of the DMA architecture is the separation of the activity on the peripheral DMA bus (the DMA access bus, DAB) from the activity on the buses between the DMA and memory (the DMA core bus, DCB and the DMA external bus, DEB. [Chapter 2, “Chip Bus Hierarchy”](#) explains the bus architecture.

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.

DMA Throughput

Peripheral DMA channels have a maximum transfer rate of one 16-bit word (on DMAC0) or one 32-bit word (on DMAC1) per two system clocks, per channel, in either direction. As the DAB and DEB buses do, the DMA controllers reside in the `SCLK` domain. The controllers synchronize accesses to and from the DCB bus which is running at `CCLK` rate.

Memory DMA channels have a maximum transfer rate of one 16-bit word (on DMAC0) or one 32-bit word (on DMAC1) per one system clock (`SCLK`), per channel.

When all DMA channels' traffic is taken in the aggregate:

- Transfers between the peripherals and the DMA unit have a maximum rate of one 16-bit transfer per system clock on DMAC0, and one 32-bit transfer per system clock on DMAC1.
- Transfers between the DMA unit and internal memory (L1 or L2) have a maximum rate of one 16-bit transfer per system clock on DMAC0, and one 32-bit transfer per system clock on DMAC1.
- Transfers between the DMA unit and external memory have a maximum rate of one 16-bit transfer per system clock on DMAC0, and one 32-bit transfer per system clock on DMAC1.

Some considerations which limit the actual performance include:

- 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 DDR SDRAM pages, or while filling cache lines.
- Direction change from receive to transmit on the DAB bus imposes a one SCLK cycle 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 DMA registers other than `DMAX_CONFIG`, `DMAX_IRQ_STATUS`, or `DMAX_PERIPHERAL_MAP` stalls all DMA activity for one cycle per 16-bit word transferred. In contrast, MMR accesses to the control/status registers do not cause stalls or wait states.
- Reads from DMA registers other than control/status registers use one PAB bus wait state, delaying the core for several core clocks.

Functional Description

- 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 `DMAEN` changes from 0 to 1 or when the `SYNC` 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 autobuffering.

Efficiency loss caused by excessive direction changes (thrashing) can be minimized by the processor's traffic control features.

The MDMA controllers are clocked by `SCLK`. 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 prefetches 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` (which is typically 7 for internal transfers and 6 for external transfers).

Memory DMA Timing Details

When the destination `DMAX_CONFIG` register is written, MDMA operation starts, after a latency of three `SCLK` cycles.

First, if either MDMA channel is selected to use descriptors, the descriptors are fetched from memory. The destination channel descriptors are fetched first. Then, after a latency of four `SCLK` cycles after the last descriptor word is returned from memory (or typically eight `SCLK` cycles after the fetch of the last descriptor word, due to memory pipelining), the source MDMA channel begins fetching data from the source buffer. The resulting data is deposited in the MDMA channel's 8-location FIFO, and then after a latency of two `SCLK` cycles, the destination MDMA channel begins writing data to the destination memory buffer.

Static Channel Prioritization

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 `PMAP` field in the `DMAX_PERIPHERAL_MAP` registers. The memory DMA streams are always lower static 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. Refer to [Table 5-1](#) for detailed information on priority and mapping of peripherals to DMA.

Temporary DMA Urgency

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.

Functional Description

Occasionally, instantaneous DMA traffic might exceed the available bandwidth, causing congestion. For example, this may occur if L1 or external memory is temporarily stalled, perhaps for a DDR 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.

For DEB bus transfers, all DMA requests to the DDR controller can be marked "urgent" under software control by setting the corresponding `DEBx_URGENT` bit in the `DDR_QUEUE` register. Please refer to ["DDR Arbitration" on page 6-12](#) for more details.

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 become urgent when handshaked operation is enabled and the `DMARx` edge count exceeds the value stored in the `HMDMAX_ECURGENT` register. If handshaked operation is disabled, software can control urgency of requests directly by altering the `DRQ` bit field in the `HMDMAX_CONTROL` register.


When one or more DMA channels express an urgent memory request, two events occur:

- All non-urgent memory requests are decreased in priority by 32, guaranteeing that only urgent requests are granted. The urgent requests compete with each other, if there is more than one, and directional preference among urgent requests is observed.
- The resulting memory transfer is marked for expedited processing in the targeted memory system (L1, L2, 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.

Memory DMA Priority and Scheduling

All MDMA operations within a DMA controller (DMAC0 or DMAC1) have lower precedence than any peripheral DMA operation within that controller. MDMA thus makes effective use of any memory bandwidth unused by peripheral DMA traffic.

 MDMA0 and MDMA1 are always the lowest priority channels in DMAC0, but they have higher priority than all DMAC1 channels by default. Therefore, it is preferable to use MDMA2 and MDMA3 to avoid starving memory bandwidth to DMAC1 peripherals. Refer to [“DCB Arbitration” on page 2-21](#) for a discussion of switching the relative priorities of DMAC0 and DMAC1.

The following discussion about MDMA stream priority and scheduling refers to MDMA streams within a DMA controller, not between DMAC0 and DMAC1.

Functional Description

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 `MDMA_ROUND_ROBIN_PERIOD` register may be programmed to select each stream in turn for a fixed number of transfers.

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 `DMACx_TCPER` register (see [“Static Channel Prioritization” on page 5-55](#)).

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 peripheral DMA 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's selection is then "locked." The `MDMA_ROUND_ROBIN_COUNT` counter field in the `DMACx_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 occurs 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.

Traffic Control

In the Blackfin DMA architecture, 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 through 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

Functional Description

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 “Temporary DMA Urgency” on page 5-55.](#)

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_TCPER` and `DMACx_TCCNT` 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_TCCNT` register. See [“Memory DMA Priority and Scheduling” on page 5-57.](#)

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, only channels 3 and 6 were requesting DAB transfers, and these transfer requests were both “against traffic,” then their effective priorities would become 19 and

22, respectively. 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_TCPER` register to `0x0000`.

Programming Model

Several synchronization and control methods are available for use in development of software tasks which manage peripheral DMA and memory DMA (see also “[Memory DMA](#)” on page 5-13). 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 peripheral DMA and memory DMA 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 `DMAX_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 `DMAX_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 `DMAX_CURR_ADDR`, `DMAX_CURR_DESC_PTR`, or `DMAX_CURR_X_COUNT/DMAX_CURR_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 are first visible to the core by memory read or write instructions. For example, in a DMA memory write operation to external memory, assume a DMA write by channel A is initiated that causes the DDR SDRAM to perform a page open operation which takes 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 is stalled behind the slow operation by channel A. Software monitoring channel B could not safely conclude whether the memory location pointed to by channel B's `DMAX_CURR_ADDR` has or has not been written, based on examination of the `DMAX_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 the length for an MDMA FIFO is eight locations (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).

Additionally, the length of the combined DMA and L1 pipelines to internal memory is approximately six 8- or 16-bit data elements. The length of the DMA and external bus interface unit (EBIU) pipelines is approximately three data elements, when measured 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 `DMAX_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 peripheral DMA 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 `DMAX_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 `DMAX_IRQ_STATUS` bit, it guarantees that the last memory operation of the

Programming Model

work unit is complete and is 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 memory other than L1 or the EBIU that the data is written.

The following examples show methods of synchronizing software with several different styles of DMA.

Single-Buffer DMA Transfers

Synchronization is simple if a peripheral's DMA activity consists of isolated transfers of single buffers. DMA activity is initiated by software writes to the channel's control registers. The user may choose to use a single descriptor in memory, in which case the software only needs to write the `DMAx_CONFIG` and the `DMAx_NEXT_DESC_PTR` registers. Alternatively, the user may choose to write all the MMR registers directly from software, ending with the write to the `DMAx_CONFIG` register.

The simplest way to signal completion of DMA is by an interrupt. This is selected by the `DI_EN` bit in the `DMAx_CONFIG` register, and by the necessary setup of the system interrupt controller. If it is desirable not to use an interrupt, the software can poll for completion by reading the `DMAx_IRQ_STATUS` register and testing the `DMA_RUN` bit. If this bit is zero, the buffer transfer has completed.

Continuous Transfers Using Autobuffering

If a peripheral's DMA data consists of a steady, periodic stream of signal data, DMA autobuffering ($FLOW = 1$) may be an effective option. Here, DMA is transferred from or to a memory buffer with a circular addressing scheme, using either one- or two-dimensional indexing with zero processor and DMA overhead for looping. Synchronization options include:

- 1D, 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.
- 2D, interrupt-driven (double buffering)—the DMA buffer is partitioned into two or more sub-buffers, and interrupts are selected (set $DI_SEL = 1$ in $DMAX_CONFIG$) to signal at the completion of each DMA inner loop. In this way, a traditional double buffer or “ping-pong” scheme could be implemented.

For example, two 512-word sub-buffers inside a 1K word buffer could be used to receive 16-bit peripheral data with these settings:

```

DMAX_START_ADDR = buffer base address
DMAX_CONFIG = 0x10D7 (FLOW = 1, DI_EN = 1, DI_SEL = 1,
DMA2D = 1, WDSIZE = 01, WNR = 1, DMAEN = 1)
DMAX_X_COUNT = 512
DMAX_X_MODIFY = 2 for 16-bit data
DMAX_Y_COUNT = 2 for two sub-buffers
DMAX_Y_MODIFY = 2, same as DMAX_X_MODIFY for contiguous
sub-buffers

```

Programming Model

- 2D, polled—if interrupt overhead is unacceptable but the loose synchronization of address/count register polling is acceptable, a 2D multibuffer synchronization scheme may be used. For example, assume receive data needs to be processed in packets of sixteen 32-bit elements. A four-part 2D DMA buffer can be allocated where each of the four sub-buffers can hold one packet with these settings:

```
DMAX_START_ADDR = buffer base address
DMAX_CONFIG = 0x101B (FLOW = 1, DI_EN = 0, DMA2D = 1,
WDSIZE = 10, WNR = 1, DMAEN = 1)
DMAX_X_COUNT = 16
DMAX_X_MODIFY = 4 for 32-bit data
DMAX_Y_COUNT = 4 for four sub-buffers
DMAX_Y_MODIFY = 4, same as DMAX_X_MODIFY for contiguous
sub-buffers
```

The synchronization core might read `DMAX_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 `DMAX_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.

- 1D unsynchronized FIFO—if a system's design guarantees that the processing of a peripheral's 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 1D autobuffer 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 1D or 2D 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 `DMAx_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.

It is important to remember the meaning of the various fields in the `DMAx_CONFIG` descriptor elements when building a list or array of DMA descriptors. In particular:

- The lower byte of `DMAx_CONFIG` specifies the DMA transfer to be performed by the *current* descriptor (for example, interrupt-enable, 2D mode).
- The upper byte of `DMAx_CONFIG` specifies the format of the *next* descriptor in the chain. The `NDSIZE` and `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 restarted, both bytes of the `DMAx_CONFIG` value written to the DMA channel's `DMAx_CONFIG` register should correspond to the current descriptor.

At a minimum, the `FLOW`, `NDSIZE`, `WNR`, and `DMAEN` fields must all agree with the current descriptor; the `WDSIZE`, `DI_EN`, `DI_SEL`, `SYNC`, and `DMA2D` fields are taken from the `DMAx_CONFIG` value in the descriptor read from memory

Programming Model

(and the field values initially written to the register are ignored). See [“Initializing Descriptors in Memory” on page 5-133](#) in the [“Programming Examples”](#) section for information on how descriptors can be set up.

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 are 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 `NDPH` and `NDPL` members point to the next descriptor, and the last descriptor points to the first.

The code that writes into this descriptor list could use the processor's circular addressing modes (`Ix`, `Lx`, `Mx`, and `Bx` registers), so that it does not need to use comparison and conditional instructions to manage the circular structure. In this case, the `NDPH` and `NDPL` 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:

- Interrupt on every descriptor
- 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 is 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 `DMAX_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 `DMAX_CONFIG` value to the DMA channel's `DMAX_CONFIG` register.

If the counts are unequal, the software instead modifies the next-to-last descriptor's `DMAX_CONFIG` value so that its upper half (`FLOW` and `NDSIZE`) now describes the newly-queued 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 `DMAX_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 `DMAX_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 queued too late

Programming Model

(that is, the modification of the next-to-last descriptor's `DMAx_CONFIG` element occurred after that element was read into the DMA unit.) In this case, the interrupt handler writes the `DMAx_CONFIG` value appropriate for the last descriptor to the DMA channel's `DMAx_CONFIG` register, increment the completed descriptor count, and exit.

Again, this system can fail if the system's interrupt latencies are large enough to cause any of the channel's DMA interrupts to be dropped. An interrupt handler capable of safely synchronizing multiple descriptors' interrupts needs to be complex, performing several MMR 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 `DMAx_CONFIG` word should have a `FLOW` value of zero. If more than one request is received before the DMA queue completion interrupt occurs, the non-interrupt code queues later descriptors, forming a waiting portion of the queue that is disconnected from the active portion of the 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 `DI_EN` 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 `DMAx_CONFIG` register write.

After queuing a new waiting descriptor, the non-interrupt software leaves a message for its interrupt handler in a memory mailbox location containing the desired `DMAx_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 processing by the DMA unit is started, unless careful synchronization measures are taken. In the most straightforward implementation of a descriptor queue, the DMA manager software never modifies 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 `DMAx_CONFIG` register. This single register write restarts the queue, effectively transforming the waiting queue to an active queue. The interrupt handler then passes 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 `DMAx_CONFIG` value of zero, indicating there is no more work to perform, then it passes an appropriate message (for example, zero) back to the non-interrupt software indicating that the queue has stopped. This simple handler can 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 `DMAx_CONFIG` value to the channel's

Programming Model

DMA_x_CONFIG register). If the queue is not stopped, however, the non-interrupt software must not write the DMA_x_CONFIG register (which would cause a DMA error), but instead it should queue the descriptor onto the waiting queue and update its mailbox directed to the interrupt handler.

Software-Triggered Descriptor Fetches

If a DMA is stopped in FLOW = 0 mode, the DMA_RUN bit in the DMA_x_IRQ_STATUS register remains set until the content of the internal DMA FIFOs is completely processed. Once the DMA_RUN bit clears, it is safe to restart the DMA by simply writing again to the DMA_x_CONFIG register. The DMA sequence is repeated with the previous settings.

Similarly, a descriptor-based DMA sequence that is stopped temporarily with a FLOW = 0 descriptor can be continued with a new write to the configuration register. When the DMA controller detects the FLOW = 0 condition by loading the DMACFG field from memory, it has already updated the next descriptor pointer, regardless of whether operating in descriptor array mode or descriptor list mode.

The next descriptor pointer remains valid, if the DMA halts and is restarted. As soon as the DMA_RUN bit clears, software can restart the DMA and force the DMA controller to fetch the next descriptor. To accomplish this, the software writes a value with the DMAEN bit set and with proper values in the FLOW and NDSIZE fields into the configuration register. The next descriptor is fetched if FLOW equals 0x4, 0x6, or 0x7. In this mode of operation, the NDSIZE field should at least span up to the DMACFG field to overwrite the configuration register immediately.

One possible procedure is:


1. Write to `DMAx_NEXT_DESC_PTR` register.
2. Write to `DMAx_CONFIG` register with

```
FLOW = 0x8
NDSIZE >= 0xA
DI_EN = 0
DMAEN = 1.
```

3. Automatically fetched `DMACFG` register has

```
FLOW = 0x0
NDSIZE = 0x0
SYNC = 1 (for transmitting DMAs only)
DI_EN = 1
DMAEN = 1.
```


4. In the interrupt routine, repeat step 2. The `DMAx_NEXT_DESC_PTR` register is updated by the descriptor fetch.

 To avoid polling of the `DMA_RUN` bit, set the `SYNC` bit in case of memory read DMAs (DMA transmit or MDMA source).

If all `DMACFG` fields in a descriptor chain have the `FLOW` and `NDSIZE` fields set to zero, the individual DMA sequences do not start until triggered by software. This is useful when the DMAs need to be synchronized with other events in the system, and it is typically performed by interrupt service routines. A single MMR write is required to trigger the next DMA sequence.

DMA Registers

Especially when applied to MDMA channels, such scenarios play an important role. Usually, the timing of MDMAs cannot be controlled (See “[Handshaked Memory DMA Operation](#)” on page 5-46). By halting descriptor chains or rings this way, the whole DMA transaction can be broken into pieces that are individually triggered by software.

 Source and destination channels of a MDMA may differ in descriptor structure. However, the total work count must match when the DMA stops. Whenever a MDMA is stopped, destination and source channels should both provide the same `FLOW = 0` mode after exactly the same number of words. Accordingly, both channels need to be started afterward.

Software-triggered descriptor fetches are illustrated in [Listing 5-7 on page 5-136](#). MDMA channels can be paused by software at any time by writing a 0 to the `DRQ` bit field in the `HMDMAX_CONTROL` register. This simply disables the self-generated DMA requests, regardless whether HMDMA is enabled or not.

DMA Registers

This section describes three categories of DMA registers:

- “[DMA Channel Registers](#)” on page 5-74
- “[Handshake MDMA \(HMDMA\) Registers](#)” on page 5-116
- “[DMA Traffic Control Registers](#)” on page 5-125

DMA Channel Registers

The processor features 24 peripheral DMA channels and four channel pairs for memory DMA. All channels have an identical set of registers summarized in [Table 5-5](#).

Table 5-5 lists the generic names of the DMA registers. For each register, the table also shows the MMR offset, a brief description of the register, and the register category.

Table 5-5. Generic Names of the DMA Memory-Mapped Registers

MMR Offset	MMR Name	MMR Description	Register Category	Name of Corresponding Descriptor Element in Memory
0x00	DMAx_NEXT_DESC_PTR MDMA_yy_NEXT_DESC_PTR on page 5-110	Link pointer to next descriptor	Parameter	NDPH (upper 16 bits), NDPL (lower 16 bits)
0x04	DMAx_START_ADDR MDMA_yy_START_ADDR on page 5-91	Start address of current buffer	Parameter	SAH (upper 16 bits), SAL (lower 16 bits)
0x08	DMAx_CONFIG MDMA_yy_CONFIG on page 5-82	DMA configuration register, including enable bit	Parameter	DMACFG
0x0C	Reserved	Reserved		
0x10	DMAx_X_COUNT MDMA_yy_X_COUNT on page 5-96	Inner loop count	Parameter	XCNT
0x14	DMAx_X_MODIFY MDMA_yy_X_MODIFY on page 5-101	Inner loop address increment, in bytes	Parameter	XMOD
0x18	DMAx_Y_COUNT MDMA_yy_Y_COUNT on page 5-103	Outer loop count (2D only)	Parameter	YCNT
0x1C	DMAx_Y_MODIFY MDMA_yy_Y_MODIFY on page 5-108	Outer loop address increment, in bytes	Parameter	YMOD

DMA Registers


Table 5-5. Generic Names of the DMA Memory-Mapped Registers (Cont'd)

MMR Offset	MMR Name	MMR Description	Register Category	Name of Corresponding Descriptor Element in Memory
0x20	DMA _x _CURR_DESC_PTR MDMA _{yy} _CURR_DESC_PTR on page 5-113	Current descriptor Pointer	Current	N/A
0x24	DMA _x _CURR_ADDR MDMA _{yy} _CURR_ADDR on page 5-93	Current DMA address	Current	N/A
0x28	DMA _x _IRQ_STATUS MDMA _{yy} _IRQ_STATUS on page 5-87	Interrupt status register: Contains completion and DMA error interrupt status and channel state (run/fetch/paused)	Control/Status	N/A
0x2C	DMA _x _PERIPHERAL_MAP MDMA _{yy} _PERIPHERAL_MAP on page 5-79	Peripheral to DMA channel mapping: Contains a 4-bit value specifying the peripheral to associate with this DMA channel (read-only for MDMA channels)	Control/Status	N/A
0x30	DMA _x _CURR_X_COUNT MDMA _{yy} _CURR_X_COUNT on page 5-98	Current count (1D) or intra-row X count (2D); counts down from X_COUNT	Current	N/A
0x34	Reserved	Reserved		

Table 5-5. Generic Names of the DMA Memory-Mapped Registers (Cont'd)

MMR Offset	MMR Name	MMR Description	Register Category	Name of Corresponding Descriptor Element in Memory
0x38	DMA _x _CURR_Y_COUNT MDMA _{yy} _CURR_Y_COUNT on page 5-105	Current row count (2D only); counts down from Y_COUNT	Current	N/A
0x3C	Reserved	Reserved		

Channel-specific register names are shown in [Table 5-5](#). For peripheral DMA channels, the prefix “DMA_x” is used where “x” stands for a channel number between 0 and 23. For memory DMA channels, the prefix is “MDMA_{yy}”, where “yy” stands for “D0”, “D1”, “D2”, “D3”, “S0”, “S1”, “S2” or “S3”, and indicates the destination and source channel registers of MDMA0 through MDMA3. For example, the configuration register of peripheral DMA channel 6 is called DMA6_CONFIG, and the register for MDMA1 source channel is called MDMA_S1_CONFIG.

 The generic MMR names shown in [Table 5-5](#) are not actually mapped to resources in the processor.

For convenience, discussions in this chapter use generic (non-peripheral specific) DMA and memory DMA register names.

DMA Registers

DMA channel registers fall into three categories:

- Parameter registers, such as `DMAx_CONFIG` and `DMAx_X_COUNT` that can be loaded directly from descriptor elements; descriptor elements are listed in [Table 5-5](#).
- Current registers, such as `DMAx_CURR_ADDR` and `DMAx_CURR_X_COUNT`
- Control/status registers, such as `DMAx_IRQ_STATUS` and `DMAx_PERIPHERAL_MAP`

All DMA registers can be accessed as 16-bit entities. The following registers, however, may also be accessed as 32-bit registers:

- `DMAx_NEXT_DESC_PTR`
- `DMAx_START_ADDR`
- `DMAx_CURR_DESC_PTR`
- `DMAx_CURR_ADDR`



When these four registers are accessed as 16-bit entities, only the lower 16 bits can be accessed.

Because confusion might arise between descriptor element names and generic DMA register names, this chapter uses different naming conventions for physical registers and their corresponding elements in descriptors that reside in memory. [Table 5-5](#) shows the relation.

Peripheral Map (DMAx_PERIPHERAL_MAP and MDMA_yy_PERIPHERAL_MAP) Registers

Each DMA channel’s peripheral map registers and addresses (DMAx_PERIPHERAL_MAP and MDMA_yy_PERIPHERAL_MAP, shown in [Figure 5-7](#) and [Table 5-6](#)) contain bits that:

- Map the channel to a specific peripheral
- Identify whether the channel is a peripheral DMA channel or a memory DMA channel

Follow these steps to swap the DMA channel priorities of two channels. Assume that channels 6 and 7 are involved.

1. Ensure that 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.

Peripheral Map Registers (DMAx_PERIPHERAL_MAP/MDMA_yy_PERIPHERAL_MAP)

R/W prior to enabling channel; RO after enabling channel

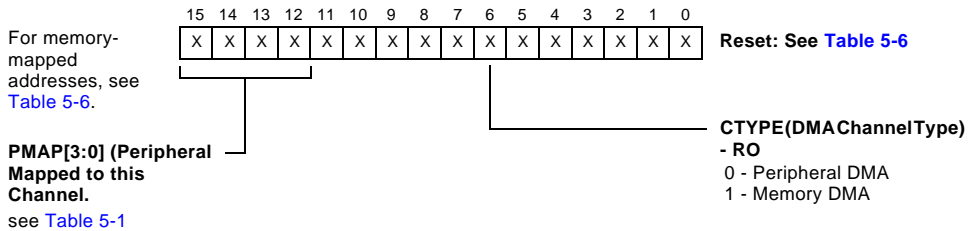


Figure 5-7. Peripheral Map Registers

DMA Registers

Table 5-6. Peripheral Map Register Addresses and Reset Values

Register Name	Memory-Mapped Address	Reset Value
DMA0_PERIPHERAL_MAP	0xFFC0 0C2C	0x0000 (SPORT0 Rx)
DMA1_PERIPHERAL_MAP	0xFFC0 0C6C	0x1000
DMA2_PERIPHERAL_MAP	0xFFC0 0CAC	0x2000
DMA3_PERIPHERAL_MAP	0xFFC0 0CEC	0x3000
DMA4_PERIPHERAL_MAP	0xFFC0 0D2C	0x4000
DMA5_PERIPHERAL_MAP	0xFFC0 0D6C	0x5000
DMA6_PERIPHERAL_MAP	0xFFC0 0DAC	0x6000
DMA7_PERIPHERAL_MAP	0xFFC0 0DEC	0x7000
DMA8_PERIPHERAL_MAP	0xFFC0 0E2C	0x8000
DMA9_PERIPHERAL_MAP	0xFFC0 0E6C	0x9000
DMA10_PERIPHERAL_MAP	0xFFC0 0EAC	0xA000
DMA11_PERIPHERAL_MAP	0xFFC0 0EEC	0xB000
DMA12_PERIPHERAL_MAP	0xFFC0 1C2C	0x0000
DMA13_PERIPHERAL_MAP	0xFFC0 1C6C	0x1000
DMA14_PERIPHERAL_MAP	0xFFC0 1CAC	0x2000
DMA15_PERIPHERAL_MAP	0xFFC0 1CEC	0x3000
DMA16_PERIPHERAL_MAP	0xFFC0 1D2C	0x4000
DMA17_PERIPHERAL_MAP	0xFFC0 1D6C	0x5000
DMA18_PERIPHERAL_MAP	0xFFC0 1DAC	0x6000
DMA19_PERIPHERAL_MAP	0xFFC0 1DEC	0x7000
DMA20_PERIPHERAL_MAP	0xFFC0 1E2C	0x8000
DMA21_PERIPHERAL_MAP	0xFFC0 1E6C	0x9000
DMA22_PERIPHERAL_MAP	0xFFC0 1EAC	0xA000
DMA23_PERIPHERAL_MAP	0xFFC0 1EEC	0xB000

Table 5-6. Peripheral Map Register Addresses and Reset Values (Cont'd)

Register Name	Memory-Mapped Address	Reset Value
MDMA_D0_PERIPHERAL_MAP	0xFFC0 0F2C	0x0040
MDMA_S0_PERIPHERAL_MAP	0xFFC0 0F6C	0x0040
MDMA_D1_PERIPHERAL_MAP	0xFFC0 0FAC	0x0040
MDMA_S1_PERIPHERAL_MAP	0xFFC0 0FEC	0x0040
MDMA_D2_PERIPHERAL_MAP	0xFFC0 1F2C	0x0040
MDMA_S2_PERIPHERAL_MAP	0xFFC0 1F6C	0x0040
MDMA_D3_PERIPHERAL_MAP	0xFFC0 1FAC	0x0040
MDMA_S3_PERIPHERAL_MAP	0xFFC0 1FEC	0x0040

[Table 5-1 on page 5-10](#) lists the peripheral map settings for each DMA-capable peripheral.

DMA Registers

DMA Configuration (DMAx_CONFIG and MDMA_yy_CONFIG) Registers

The DMA configuration registers and addresses (DMAx_CONFIG and MDMA_yy_CONFIG), shown in [Figure 5-8](#) and [Table 5-7](#), set up DMA parameters and operating modes. Note that writing the DMAx_CONFIG register while DMA is already running causes a DMA error unless writing with the DMAEN bit set to 0.

Configuration Registers (DMAx_CONFIG/MDMA_yy_CONFIG)

R/W prior to enabling channel; RO after enabling channel

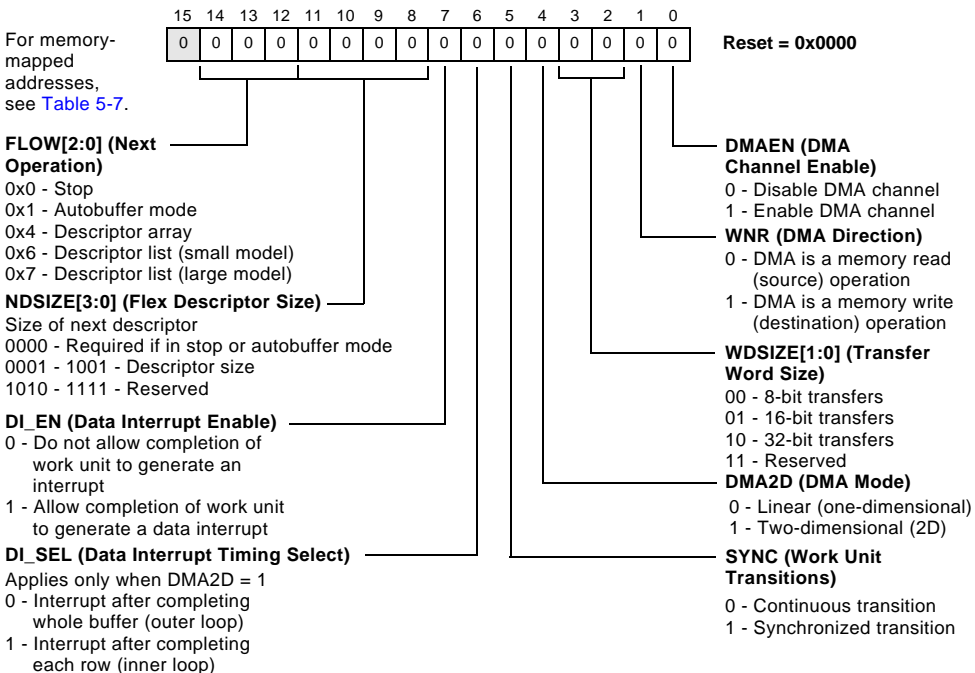


Figure 5-8. DMA Configuration Registers

Table 5-7. DMA Configuration Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_CONFIG	0xFFC0 0C08
DMA1_CONFIG	0xFFC0 0C48
DMA2_CONFIG	0xFFC0 0C88
DMA3_CONFIG	0xFFC0 0CC8
DMA4_CONFIG	0xFFC0 0D08
DMA5_CONFIG	0xFFC0 0D48
DMA6_CONFIG	0xFFC0 0D88
DMA7_CONFIG	0xFFC0 0DC8
DMA8_CONFIG	0xFFC0 0E08
DMA9_CONFIG	0xFFC0 0E48
DMA10_CONFIG	0xFFC0 0E88
DMA11_CONFIG	0xFFC0 0EC8
DMA12_CONFIG	0xFFC0 1C08
DMA13_CONFIG	0xFFC0 1C48
DMA14_CONFIG	0xFFC0 1C88
DMA15_CONFIG	0xFFC0 1CC8
DMA16_CONFIG	0xFFC0 1D08
DMA17_CONFIG	0xFFC0 1D48
DMA18_CONFIG	0xFFC0 1D88
DMA19_CONFIG	0xFFC0 1DC8
DMA20_CONFIG	0xFFC0 1E08
DMA21_CONFIG	0xFFC0 1E48
DMA22_CONFIG	0xFFC0 1E88
DMA23_CONFIG	0xFFC0 1EC8

DMA Registers

Table 5-7. DMA Configuration Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
MDMA_D0_CONFIG	0xFFC0 0F08
MDMA_S0_CONFIG	0xFFC0 0F48
MDMA_D1_CONFIG	0xFFC0 0F88
MDMA_S1_CONFIG	0xFFC0 0FC8
MDMA_D2_CONFIG	0xFFC0 1F08
MDMA_S2_CONFIG	0xFFC0 1F48
MDMA_D3_CONFIG	0xFFC0 1F88
MDMA_S3_CONFIG	0xFFC0 1FC8

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:

0x0 - stop. When the current work unit completes, the DMA channel stops automatically, after signaling an interrupt (if selected). The `DMA_RUN` status bit in the `DMAx_IRQ_STATUS` register changes from 1 to 0, while the `DMAEN` bit in the `DMAx_CONFIG` register 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 the `DMAx_CONFIG` register specifying the next work unit, in which the `DMAEN` bit 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. Upon 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 DMAEN bit in the DMA_{MAX}_CONFIG register.

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 the DMA_{MAX}_CURR_DESC_PTR register 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 DMA_{MAX}_NEXT_DESC_PTR register, thus confining all descriptors to a specific 64K page in memory.

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 descriptor elements in memory to load. 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.
- SYNC (work unit transitions). This bit specifies whether the DMA channel performs a continuous transition (SYNC = 0) or a synchronized transition (SYNC = 1) between work units. For more information, see [“Work Unit Transitions” on page 5-33](#).

DMA Registers

In DMA transmit (memory read) and MDMA source channels, the SYNC bit controls the interrupt timing at the end of the work unit and the handling of the DMA FIFO between the current and next work unit.



Work unit transitions for MDMA streams are controlled by SYNC bit of the MDMA source channel's DMA_x_CONFIG register. The SYNC bit of the MDMA destination channel is reserved and must be 0.

- DMA2D (DMA mode). This bit specifies whether DMA mode involves only DMA_x_X_COUNT and DMA_x_X_MODIFY (one-dimensional DMA) or also involves DMA_x_Y_COUNT and DMA_x_Y_MODIFY (two-dimensional DMA).
- 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 DMA access bus). The DMA address pointer registers' increment sizes (strides) must be a multiple of the transfer unit size—1 for 8-bit, 2 for 16-bit, 4 for 32-bit.
- WNR (DMA direction). This bit specifies DMA direction—memory read (0) or memory write (1).
- DMAEN (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.

Interrupt Status (DMAx_IRQ_STATUS and MDMA_yy_IRQ_STATUS) Registers

The interrupt status registers and addresses (DMAx_IRQ_STATUS and MDMA_yy_IRQ_STATUS), shown in [Figure 5-9](#) and [Table 5-8](#), contain 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 memory 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 undelivered transmit data in the FIFO, the DMA_RUN bit is 1.

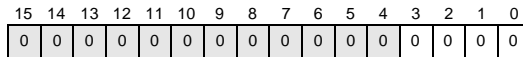
For a memory write DMA channel, the state of the DMA_RUN bit has no meaning after the last DMA_DONE event is signaled. It does not indicate the status of the DMA FIFO.

For MDMA transfers where it is not desired to use an interrupt to notify when the DMA operation has ended, software should poll the DMA_DONE bit, and not the DMA_RUN bit, to determine when the transaction has completed.

DMA Registers

Interrupt Status Registers (DMAx_IRQ_STATUS/MDMA_yy_IRQ_STATUS)

For memory-mapped addresses, see [Table 5-8](#).



Reset = 0x0000

DMA_RUN (DMA Channel Running) - RO

This bit is set to 1 automatically when the DMAx_CONFIG register is written.

- 0 - This DMA channel is disabled, or it is enabled but paused (FLOW mode 0)
- 1 - This DMA channel is enabled and operating, either transferring data or fetching a DMA descriptor

DFETCH (DMA Descriptor Fetch) - RO

This bit is set to 1 automatically when the DMAx_CONFIG register is written with FLOW modes 4–7.

- 0 - This DMA channel is disabled, or it is enabled but stopped (FLOW mode 0)
- 1 - This DMA channel is enabled and presently fetching a DMA descriptor

DMA_DONE (DMA Completion Interrupt Status) - W1C

- 0 - No interrupt is being asserted for this channel
- 1 - DMA work unit has completed, and this DMA channel's interrupt is being asserted

DMA_ERR (DMA Error Interrupt Status) - W1C

- 0 - No DMA error has occurred
- 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/status and parameter registers are unchanged.

Figure 5-9. Interrupt Status Registers

Table 5-8. Interrupt Status Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_IRQ_STATUS	0xFFC0 0C28
DMA1_IRQ_STATUS	0xFFC0 0C68
DMA2_IRQ_STATUS	0xFFC0 0CA8
DMA3_IRQ_STATUS	0xFFC0 0CE8
DMA4_IRQ_STATUS	0xFFC0 0D28
DMA5_IRQ_STATUS	0xFFC0 0D68
DMA6_IRQ_STATUS	0xFFC0 0DA8
DMA7_IRQ_STATUS	0xFFC0 0DE8
DMA8_IRQ_STATUS	0xFFC0 0E28
DMA9_IRQ_STATUS	0xFFC0 0E68
DMA10_IRQ_STATUS	0xFFC0 0EA8
DMA11_IRQ_STATUS	0xFFC0 0EE8
DMA12_IRQ_STATUS	0xFFC0 1C28
DMA13_IRQ_STATUS	0xFFC0 1C68
DMA14_IRQ_STATUS	0xFFC0 1CA8
DMA15_IRQ_STATUS	0xFFC0 1CE8
DMA16_IRQ_STATUS	0xFFC0 1D28
DMA17_IRQ_STATUS	0xFFC0 1D68
DMA18_IRQ_STATUS	0xFFC0 1DA8
DMA19_IRQ_STATUS	0xFFC0 1DE8
DMA20_IRQ_STATUS	0xFFC0 1E28
DMA21_IRQ_STATUS	0xFFC0 1E68
DMA22_IRQ_STATUS	0xFFC0 1EA8

DMA Registers

Table 5-8. Interrupt Status Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
DMA23_IRQ_STATUS	0xFFC0 1EE8
MDMA_D0_IRQ_STATUS	0xFFC0 0F28
MDMA_S0_IRQ_STATUS	0xFFC0 0F68
MDMA_D1_IRQ_STATUS	0xFFC0 0FA8
MDMA_S1_IRQ_STATUS	0xFFC0 0FE8
MDMA_D2_IRQ_STATUS	0xFFC0 1F28
MDMA_S2_IRQ_STATUS	0xFFC0 1F68
MDMA_D3_IRQ_STATUS	0xFFC0 1FA8
MDMA_S3_IRQ_STATUS	0xFFC0 1FE8

The processor supports a flexible interrupt control structure with three interrupt sources:

- Data driven interrupts (see [Table 5-9](#))
- Peripheral error interrupts
- DMA error interrupts (for example, bad descriptor or bus error)

Separate interrupt request (IRQ) levels are allocated for data and peripheral error interrupts, and DMA error interrupts.

Table 5-9. Data Driven Interrupts

Interrupt Name	Description
No interrupt	Interrupts can be disabled for a given work unit.
Peripheral interrupt	These are peripheral (non-DMA) interrupts.

Table 5-9. Data Driven Interrupts (Cont'd)

Interrupt Name	Description
Row completion	DMA Interrupts can occur on the completion of a row (CURR_X_COUNT expiration).
Buffer completion	DMA Interrupts can occur on the completion of an entire buffer (when CURR_X_COUNT and CURR_Y_COUNT expire).

The DMA error conditions for 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.



The `DMA_DONE` and `DMA_ERR` interrupt indicators are write-1-to-clear (W1C).



When switching a peripheral from DMA to non-DMA mode, the peripheral's interrupts should be disabled during the mode switch (through the appropriate peripheral registers or `SIC_IMASKx`) so that no unintended interrupt is generated on the shared DMA/interrupt request line.

Start Address (`DMAx_START_ADDR` and `MDMA_yy_START_ADDR`) Registers

The start address registers and addresses (`DMAx_START_ADDR` and `MDMA_yy_START_ADDR`), shown in [Figure 5-10](#) and [Table 5-10](#), contain the start address of the data buffer currently targeted for DMA.

DMA Registers

Start Address Registers (DMAx_START_ADDR/ MDMA_yy_START_ADDR)

R/W prior to enabling channel; RO after enabling channel

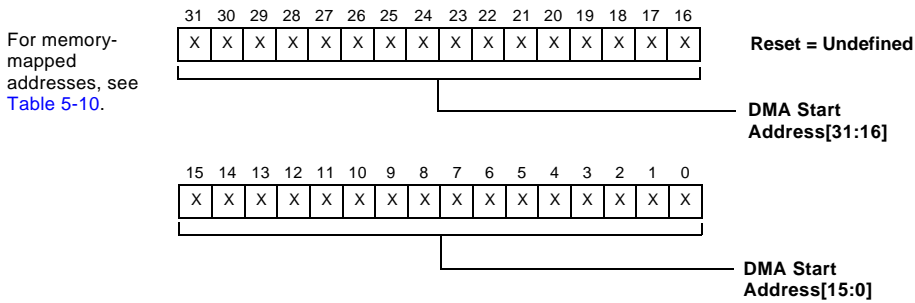


Figure 5-10. Start Address Registers

Table 5-10. Start Address Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_START_ADDR	0xFFC0 0C04
DMA1_START_ADDR	0xFFC0 0C44
DMA2_START_ADDR	0xFFC0 0C84
DMA3_START_ADDR	0xFFC0 0CC4
DMA4_START_ADDR	0xFFC0 0D04
DMA5_START_ADDR	0xFFC0 0D44
DMA6_START_ADDR	0xFFC0 0D84
DMA7_START_ADDR	0xFFC0 0DC4
DMA8_START_ADDR	0xFFC0 0E04
DMA9_START_ADDR	0xFFC0 0E44
DMA10_START_ADDR	0xFFC0 0E84
DMA11_START_ADDR	0xFFC0 0EC4
DMA12_START_ADDR	0xFFC0 1C04

Table 5-10. Start Address Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
DMA13_START_ADDR	0xFFC0 1C44
DMA14_START_ADDR	0xFFC0 1C84
DMA15_START_ADDR	0xFFC0 1CC4
DMA16_START_ADDR	0xFFC0 1D04
DMA17_START_ADDR	0xFFC0 1D44
DMA18_START_ADDR	0xFFC0 1D84
DMA19_START_ADDR	0xFFC0 1DC4
DMA20_START_ADDR	0xFFC0 1E04
DMA21_START_ADDR	0xFFC0 1E44
DMA22_START_ADDR	0xFFC0 1E84
DMA23_START_ADDR	0xFFC0 1EC4
MDMA_D0_START_ADDR	0xFFC0 0F04
MDMA_S0_START_ADDR	0xFFC0 0F44
MDMA_D1_START_ADDR	0xFFC0 0F84
MDMA_S1_START_ADDR	0xFFC0 0FC4
MDMA_D2_START_ADDR	0xFFC0 1F04
MDMA_S2_START_ADDR	0xFFC0 1F44
MDMA_D3_START_ADDR	0xFFC0 1F84
MDMA_S3_START_ADDR	0xFFC0 1FC4

Current Address (DMAx_CURR_ADDR and MDMA_yy_CURR_ADDR) Registers

The current address registers and addresses (DMAx_CURR_ADDR and MDMA_yy_CURR_ADDR), shown in [Figure 5-11](#) and [Table 5-11](#), contain the present DMA transfer address for a given DMA session. On the first mem-

DMA Registers

ory transfer of a DMA work unit, the `DMAx_CURR_ADDR` register is loaded from the `DMAx_START_ADDR` register, and it is incremented as each transfer occurs. The current address register contains 32 bits.

Current Address Registers (`DMAx_CURR_ADDR`/`MDMA_yy_CURR_ADDR`)

R/W prior to enabling channel; RO after enabling channel

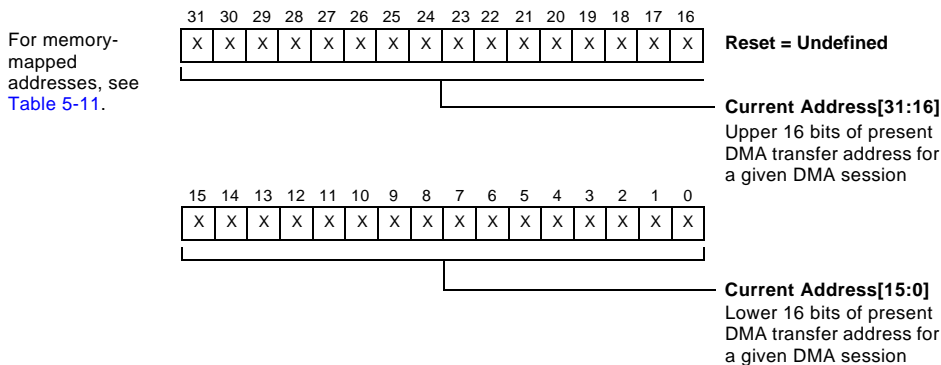


Figure 5-11. Current Address Registers

Table 5-11. Current Address Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_CURR_ADDR	0xFFC0 0C24
DMA1_CURR_ADDR	0xFFC0 0C64
DMA2_CURR_ADDR	0xFFC0 0CA4
DMA3_CURR_ADDR	0xFFC0 0CE4
DMA4_CURR_ADDR	0xFFC0 0D24
DMA5_CURR_ADDR	0xFFC0 0D64
DMA6_CURR_ADDR	0xFFC0 0DA4
DMA7_CURR_ADDR	0xFFC0 0DE4
DMA8_CURR_ADDR	0xFFC0 0E24
DMA9_CURR_ADDR	0xFFC0 0E64
DMA10_CURR_ADDR	0xFFC0 0EA4
DMA11_CURR_ADDR	0xFFC0 0EE4
DMA12_CURR_ADDR	0xFFC0 1C24
DMA13_CURR_ADDR	0xFFC0 1C64
DMA14_CURR_ADDR	0xFFC0 1CA4
DMA15_CURR_ADDR	0xFFC0 1CE4
DMA16_CURR_ADDR	0xFFC0 1D24
DMA17_CURR_ADDR	0xFFC0 1D64
DMA18_CURR_ADDR	0xFFC0 1DA4
DMA19_CURR_ADDR	0xFFC0 1DE4
DMA20_CURR_ADDR	0xFFC0 1E24
DMA21_CURR_ADDR	0xFFC0 1E64
DMA22_CURR_ADDR	0xFFC0 1EA4

DMA Registers

Table 5-11. Current Address Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA23_CURR_ADDR	0xFFC0 1EE4
MDMA_D0_CURR_ADDR	0xFFC0 0F24
MDMA_S0_CURR_ADDR	0xFFC0 0F64
MDMA_D1_CURR_ADDR	0xFFC0 0FA4
MDMA_S1_CURR_ADDR	0xFFC0 0FE4
MDMA_D2_CURR_ADDR	0xFFC0 1F24
MDMA_S2_CURR_ADDR	0xFFC0 1F64
MDMA_D3_CURR_ADDR	0xFFC0 1FA4
MDMA_S3_CURR_ADDR	0xFFC0 1FE4

Inner Loop Count (DMA_x_X_COUNT and MDMA_{yy}_X_COUNT) Registers

For 2D DMA, the inner loop count registers and addresses (DMA_x_X_COUNT and MDMA_{yy}_X_COUNT), shown in [Figure 5-12](#) and [Table 5-12](#), contain the inner loop count. For 1D DMA, it specifies the number of elements to read in. For details, see “[Two-Dimensional DMA Operation](#)” on [page 5-19](#). A value of 0 in DMA_x_X_COUNT corresponds to 65,536 elements.

Inner Loop Count Registers (DMA_x_X_COUNT/MDMA_{yy}_X_COUNT)

R/W prior to enabling channel; RO after enabling channel

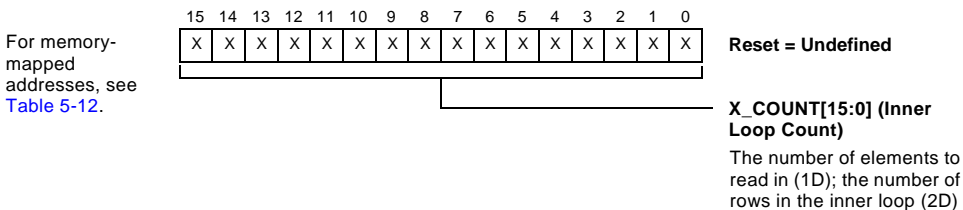


Figure 5-12. Inner Loop Count Registers

Table 5-12. Inner Loop Count Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_X_COUNT	0xFFC0 0C10
DMA1_X_COUNT	0xFFC0 0C50
DMA2_X_COUNT	0xFFC0 0C90
DMA3_X_COUNT	0xFFC0 0CD0
DMA4_X_COUNT	0xFFC0 0D10
DMA5_X_COUNT	0xFFC0 0D50
DMA6_X_COUNT	0xFFC0 0D90
DMA7_X_COUNT	0xFFC0 0DD0
DMA8_X_COUNT	0xFFC0 0E10
DMA9_X_COUNT	0xFFC0 0E50
DMA10_X_COUNT	0xFFC0 0E90
DMA11_X_COUNT	0xFFC0 0ED0
DMA12_X_COUNT	0xFFC0 1C10
DMA13_X_COUNT	0xFFC0 1C50
DMA14_X_COUNT	0xFFC0 1C90
DMA15_X_COUNT	0xFFC0 1CD0
DMA16_X_COUNT	0xFFC0 1D10
DMA17_X_COUNT	0xFFC0 1D50
DMA18_X_COUNT	0xFFC0 1D90
DMA19_X_COUNT	0xFFC0 1DD0
DMA20_X_COUNT	0xFFC0 1E10
DMA21_X_COUNT	0xFFC0 1E50
DMA22_X_COUNT	0xFFC0 1E90
DMA23_X_COUNT	0xFFC0 1ED0

DMA Registers

Table 5-12. Inner Loop Count Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
MDMA_D0_X_COUNT	0xFFC0 0F10
MDMA_S0_X_COUNT	0xFFC0 0F50
MDMA_D1_X_COUNT	0xFFC0 0F90
MDMA_S1_X_COUNT	0xFFC0 0FD0
MDMA_D2_X_COUNT	0xFFC0 1F10
MDMA_S2_X_COUNT	0xFFC0 1F50
MDMA_D3_X_COUNT	0xFFC0 1F90
MDMA_S3_X_COUNT	0xFFC0 1FD0

Current Inner Loop Count (DMAx_CURR_X_COUNT and MDMA_yy_CURR_X_COUNT) Registers

The current inner loop count registers and addresses (DMAx_CURR_X_COUNT and MDMA_yy_CURR_X_COUNT), shown in [Figure 5-13](#) and [Table 5-13](#), hold the number of transfers remaining in the current DMA row (inner loop).

On the first memory transfer of each DMA work unit, it is loaded with the value in the DMAx_X_COUNT register and then decremented. For 2D DMA, on the last memory transfer in each row except the last row, it is reloaded with the value in the DMAx_X_COUNT register; this occurs at the same time that the value in the DMAx_CURR_Y_COUNT register is decremented. 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, the `DMAx_CURR_X_COUNT` register value is 0 only when the entire transfer is complete. Between rows it is equal to the value of the `DMAx_X_COUNT` register.

Current Inner Loop Count Registers (`DMAx_CURR_X_COUNT/MDMA_yy_CURR_X_COUNT`)

R/W prior to enabling channel; RO after enabling channel

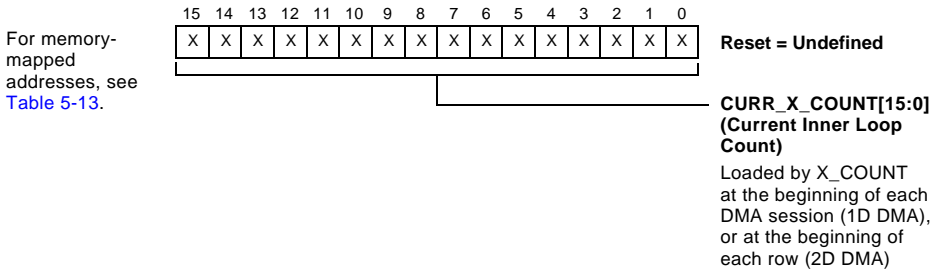


Figure 5-13. Current Inner Loop Count Registers

Table 5-13. Current Inner Loop Count Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_CURR_X_COUNT	0xFFC0 0C30
DMA1_CURR_X_COUNT	0xFFC0 0C70
DMA2_CURR_X_COUNT	0xFFC0 0CB0
DMA3_CURR_X_COUNT	0xFFC0 0CF0
DMA4_CURR_X_COUNT	0xFFC0 0D30
DMA5_CURR_X_COUNT	0xFFC0 0D70
DMA6_CURR_X_COUNT	0xFFC0 0DB0
DMA7_CURR_X_COUNT	0xFFC0 0DF0
DMA8_CURR_X_COUNT	0xFFC0 0E30
DMA9_CURR_X_COUNT	0xFFC0 0E70


DMA Registers

Table 5-13. Current Inner Loop Count Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
DMA10_CURR_X_COUNT	0xFFC0 0EB0
DMA11_CURR_X_COUNT	0xFFC0 0EF0
DMA12_CURR_X_COUNT	0xFFC0 1C30
DMA13_CURR_X_COUNT	0xFFC0 1C70
DMA14_CURR_X_COUNT	0xFFC0 1CB0
DMA15_CURR_X_COUNT	0xFFC0 1CF0
DMA16_CURR_X_COUNT	0xFFC0 1D30
DMA17_CURR_X_COUNT	0xFFC0 1D70
DMA18_CURR_X_COUNT	0xFFC0 1DB0
DMA19_CURR_X_COUNT	0xFFC0 1DF0
DMA20_CURR_X_COUNT	0xFFC0 1E30
DMA21_CURR_X_COUNT	0xFFC0 1E70
DMA22_CURR_X_COUNT	0xFFC0 1EB0
DMA23_CURR_X_COUNT	0xFFC0 1EF0
MDMA_D0_CURR_X_COUNT	0xFFC0 0F30
MDMA_S0_CURR_X_COUNT	0xFFC0 0F70
MDMA_D1_CURR_X_COUNT	0xFFC0 0FB0
MDMA_S1_CURR_X_COUNT	0xFFC0 0FF0
MDMA_D2_CURR_X_COUNT	0xFFC0 1F30
MDMA_S2_CURR_X_COUNT	0xFFC0 1F70
MDMA_D3_CURR_X_COUNT	0xFFC0 1FB0
MDMA_S3_CURR_X_COUNT	0xFFC0 1FF0

Inner Loop Address Increment (DMAx_X_MODIFY and MDMA_yy_X_MODIFY) Registers

The inner loop address increment registers and addresses (DMAx_X_MODIFY and MDMA_yy_X_MODIFY), shown in [Figure 5-14](#) and [Table 5-14](#), contain a signed, two’s-complement byte-address increment. In 1D DMA, this increment is the stride that is applied after transferring each element.

 DMAx_X_MODIFY is specified in bytes, regardless of the DMA transfer size.

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, the DMAx_Y_MODIFY register is applied instead, except on the very last transfer of each work unit. The DMAx_X_MODIFY register is always applied on the last transfer of a work unit.

The DMAx_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.

Inner Loop Address Increment Registers (DMAx_X_MODIFY/MDMA_yy_X_MODIFY)

R/W prior to enabling channel; RO after enabling channel

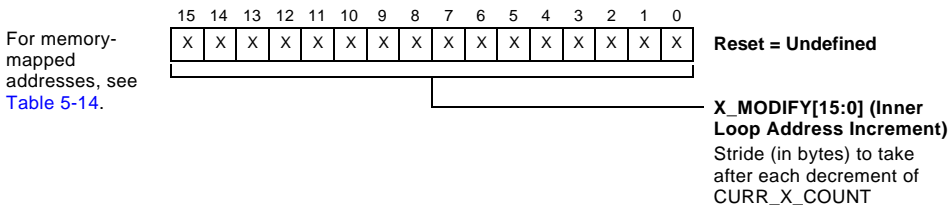


Figure 5-14. Inner Loop Address Increment Registers

DMA Registers

Table 5-14. Inner Loop Address Increment Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_X_MODIFY	0xFFC0 0C14
DMA1_X_MODIFY	0xFFC0 0C54
DMA2_X_MODIFY	0xFFC0 0C94
DMA3_X_MODIFY	0xFFC0 0CD4
DMA4_X_MODIFY	0xFFC0 0D14
DMA5_X_MODIFY	0xFFC0 0D54
DMA6_X_MODIFY	0xFFC0 0D94
DMA7_X_MODIFY	0xFFC0 0DD4
DMA8_X_MODIFY	0xFFC0 0E14
DMA9_X_MODIFY	0xFFC0 0E54
DMA10_X_MODIFY	0xFFC0 0E94
DMA11_X_MODIFY	0xFFC0 0ED4
DMA12_X_MODIFY	0xFFC0 1C14
DMA13_X_MODIFY	0xFFC0 1C54
DMA14_X_MODIFY	0xFFC0 1C94
DMA15_X_MODIFY	0xFFC0 1CD4
DMA16_X_MODIFY	0xFFC0 1D14
DMA17_X_MODIFY	0xFFC0 1D54
DMA18_X_MODIFY	0xFFC0 1D94
DMA19_X_MODIFY	0xFFC0 1DD4
DMA20_X_MODIFY	0xFFC0 1E14
DMA21_X_MODIFY	0xFFC0 1E54
DMA22_X_MODIFY	0xFFC0 1E94
DMA23_X_MODIFY	0xFFC0 1ED4

Table 5-14. Inner Loop Address Increment Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
MDMA_D0_X_MODIFY	0xFFC0 0F14
MDMA_S0_X_MODIFY	0xFFC0 0F54
MDMA_D1_X_MODIFY	0xFFC0 0F94
MDMA_S1_X_MODIFY	0xFFC0 0FD4
MDMA_D2_X_MODIFY	0xFFC0 1F14
MDMA_S2_X_MODIFY	0xFFC0 1F54
MDMA_D3_X_MODIFY	0xFFC0 1F94
MDMA_S3_X_MODIFY	0xFFC0 1FD4

Outer Loop Count (DMAx_Y_COUNT and MDMA_yy_Y_COUNT) Registers

For 2D DMA, the outer loop count registers and addresses (DMAx_Y_COUNT and MDMA_yy_Y_COUNT), shown in [Figure 5-15](#) and [Table 5-15](#), 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 Operation”](#) on page 5-19.

Outer Loop Count Registers (DMAx_Y_COUNT/MDMA_yy_Y_COUNT)

R/W prior to enabling channel; RO after enabling channel

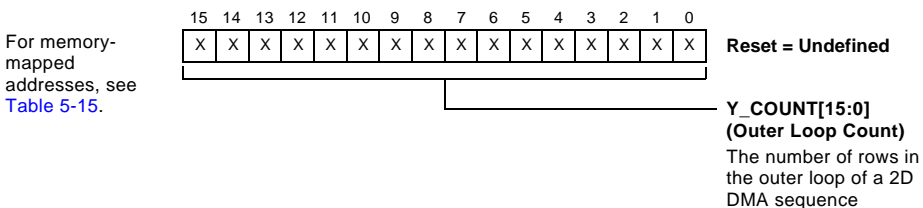


Figure 5-15. Outer Loop Count Registers

DMA Registers

Table 5-15. Outer Loop Count Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_Y_COUNT	0xFFC0 0C18
DMA1_Y_COUNT	0xFFC0 0C58
DMA2_Y_COUNT	0xFFC0 0C98
DMA3_Y_COUNT	0xFFC0 0CD8
DMA4_Y_COUNT	0xFFC0 0D18
DMA5_Y_COUNT	0xFFC0 0D58
DMA6_Y_COUNT	0xFFC0 0D98
DMA7_Y_COUNT	0xFFC0 0DD8
DMA8_Y_COUNT	0xFFC0 0E18
DMA9_Y_COUNT	0xFFC0 0E58
DMA10_Y_COUNT	0xFFC0 0E98
DMA11_Y_COUNT	0xFFC0 0ED8
DMA12_Y_COUNT	0xFFC0 1C18
DMA13_Y_COUNT	0xFFC0 1C58
DMA14_Y_COUNT	0xFFC0 1C98
DMA15_Y_COUNT	0xFFC0 1CD8
DMA16_Y_COUNT	0xFFC0 1D18
DMA17_Y_COUNT	0xFFC0 1D58
DMA18_Y_COUNT	0xFFC0 1D98
DMA19_Y_COUNT	0xFFC0 1DD8
DMA20_Y_COUNT	0xFFC0 1E18
DMA21_Y_COUNT	0xFFC0 1E58
DMA22_Y_COUNT	0xFFC0 1E98
DMA23_Y_COUNT	0xFFC0 1ED8

Table 5-15. Outer Loop Count Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
MDMA_D0_Y_COUNT	0xFFC0 0F18
MDMA_S0_Y_COUNT	0xFFC0 0F58
MDMA_D1_Y_COUNT	0xFFC0 0F98
MDMA_S1_Y_COUNT	0xFFC0 0FD8
MDMA_D2_Y_COUNT	0xFFC0 1F18
MDMA_S2_Y_COUNT	0xFFC0 1F58
MDMA_D3_Y_COUNT	0xFFC0 1F98
MDMA_S3_Y_COUNT	0xFFC0 1FD8

Current Outer Loop Count (DMAx_CURR_Y_COUNT and MDMA_yy_CURR_Y_COUNT) Registers

The current outer loop count registers and addresses (DMAx_CURR_Y_COUNT and MDMA_yy_CURR_Y_COUNT), shown in [Figure 5-16](#) and [Table 5-16](#), used only in 2D mode, hold the number of full or partial rows (outer loops) remaining in the current work unit.

On the first memory transfer of each DMA work unit, it is loaded with the value of the DMAx_Y_COUNT register. The register is decremented each time the DMAx_CURR_X_COUNT register expires during 2D DMA operation (1 to DMAx_X_COUNT or 1 to 0 transition), signifying completion of an entire row transfer. After a 2D DMA session is complete, DMAx_CURR_Y_COUNT = 1 and DMAx_CURR_X_COUNT = 0.

DMA Registers

Current Outer Loop Count Registers (DMAx_CURR_Y_COUNT/ MDMA_yy_CURR_Y_COUNT)

R/W prior to enabling channel; RO after enabling channel

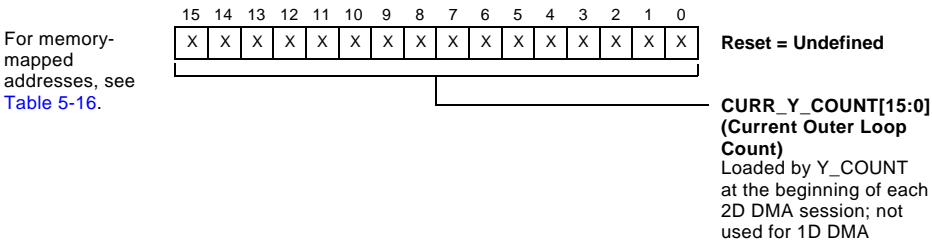


Figure 5-16. Current Outer Loop Count Registers

Table 5-16. Current Outer Loop Count Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_CURR_Y_COUNT	0xFFC0 0C38
DMA1_CURR_Y_COUNT	0xFFC0 0C78
DMA2_CURR_Y_COUNT	0xFFC0 0CB8
DMA3_CURR_Y_COUNT	0xFFC0 0CF8
DMA4_CURR_Y_COUNT	0xFFC0 0D38
DMA5_CURR_Y_COUNT	0xFFC0 0D78
DMA6_CURR_Y_COUNT	0xFFC0 0DB8
DMA7_CURR_Y_COUNT	0xFFC0 0DF8
DMA8_CURR_Y_COUNT	0xFFC0 0E38
DMA9_CURR_Y_COUNT	0xFFC0 0E78
DMA10_CURR_Y_COUNT	0xFFC0 0EB8
DMA11_CURR_Y_COUNT	0xFFC0 0EF8


Table 5-16. Current Outer Loop Count Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
DMA12_CURR_Y_COUNT	0xFFC0 1C38
DMA13_CURR_Y_COUNT	0xFFC0 1C78
DMA14_CURR_Y_COUNT	0xFFC0 1CB8
DMA15_CURR_Y_COUNT	0xFFC0 1CF8
DMA16_CURR_Y_COUNT	0xFFC0 1D38
DMA17_CURR_Y_COUNT	0xFFC0 1D78
DMA18_CURR_Y_COUNT	0xFFC0 1DB8
DMA19_CURR_Y_COUNT	0xFFC0 1DF8
DMA20_CURR_Y_COUNT	0xFFC0 1E38
DMA21_CURR_Y_COUNT	0xFFC0 1E78
DMA22_CURR_Y_COUNT	0xFFC0 1EB8
DMA23_CURR_Y_COUNT	0xFFC0 1EF8
MDMA_D0_CURR_Y_COUNT	0xFFC0 0F38
MDMA_S0_CURR_Y_COUNT	0xFFC0 0F78
MDMA_D1_CURR_Y_COUNT	0xFFC0 0FB8
MDMA_S1_CURR_Y_COUNT	0xFFC0 0FF8
MDMA_D2_CURR_Y_COUNT	0xFFC0 1F38
MDMA_S2_CURR_Y_COUNT	0xFFC0 1F78
MDMA_D3_CURR_Y_COUNT	0xFFC0 1FB8
MDMA_S3_CURR_Y_COUNT	0xFFC0 1FF8

DMA Registers

Outer Loop Address Increment (DMAx_Y_MODIFY and MDMA_yy_Y_MODIFY) Registers

The outer loop address increment registers and addresses (DMAx_Y_MODIFY and MDMA_yy_Y_MODIFY), shown in [Figure 5-17](#) and [Table 5-17](#), contain a signed, two’s-complement value. This byte-address increment is applied after each decrement of the DMAx_CURR_Y_COUNT register except for the last item in the 2D array where the DMAx_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 Operation” on page 5-19](#).

 DMAx_Y_MODIFY is specified in bytes, regardless of the DMA transfer size.

Outer Loop Address Increment Registers (DMAx_Y_MODIFY/ MDMA_yy_Y_MODIFY)

R/W prior to enabling channel; RO after enabling channel

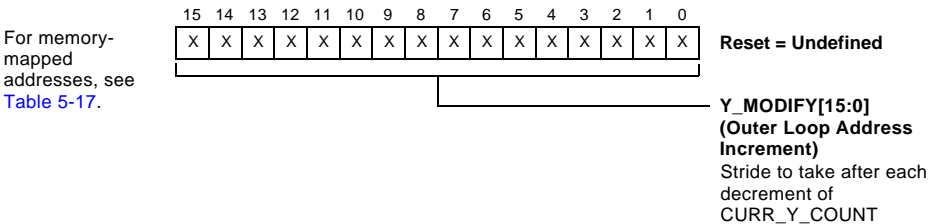


Figure 5-17. Outer Loop Address Increment Registers

Table 5-17. Outer Loop Address Increment Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_Y_MODIFY	0xFFC0 0C1C
DMA1_Y_MODIFY	0xFFC0 0C5C
DMA2_Y_MODIFY	0xFFC0 0C9C

Table 5-17. Outer Loop Address Increment Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
DMA3_Y_MODIFY	0xFFC0_0CDC
DMA4_Y_MODIFY	0xFFC0_0D1C
DMA5_Y_MODIFY	0xFFC0_0D5C
DMA6_Y_MODIFY	0xFFC0_0D9C
DMA7_Y_MODIFY	0xFFC0_0DDC
DMA8_Y_MODIFY	0xFFC0_0E1C
DMA9_Y_MODIFY	0xFFC0_0E5C
DMA10_Y_MODIFY	0xFFC0_0E9C
DMA11_Y_MODIFY	0xFFC0_0EDC
DMA12_Y_MODIFY	0xFFC0_1C1C
DMA13_Y_MODIFY	0xFFC0_1C5C
DMA14_Y_MODIFY	0xFFC0_1C9C
DMA15_Y_MODIFY	0xFFC0_1CDC
DMA16_Y_MODIFY	0xFFC0_1D1C
DMA17_Y_MODIFY	0xFFC0_1D5C
DMA18_Y_MODIFY	0xFFC0_1D9C
DMA19_Y_MODIFY	0xFFC0_1DDC
DMA20_Y_MODIFY	0xFFC0_1E1C
DMA21_Y_MODIFY	0xFFC0_1E5C
DMA22_Y_MODIFY	0xFFC0_1E9C
DMA23_Y_MODIFY	0xFFC0_1EDC
MDMA_D0_Y_MODIFY	0xFFC0_0F1C
MDMA_S0_Y_MODIFY	0xFFC0_0F5C
MDMA_D1_Y_MODIFY	0xFFC0_0F9C

DMA Registers

Table 5-17. Outer Loop Address Increment Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
MDMA_S1_Y_MODIFY	0xFFC0 0FDC
MDMA_D2_Y_MODIFY	0xFFC0 1F1C
MDMA_S2_Y_MODIFY	0xFFC0 1F5C
MDMA_D3_Y_MODIFY	0xFFC0 1F9C
MDMA_S3_Y_MODIFY	0xFFC0 1FDC

Next Descriptor Pointer (DMAx_NEXT_DESC_PTR and MDMA_yy_NEXT_DESC_PTR) Registers

The next descriptor pointer registers and addresses (DMAx_NEXT_DESC_PTR and MDMA_yy_NEXT_DESC_PTR), shown in [Figure 5-18](#) and [Table 5-18](#), specify 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 in small and large descriptor list modes. At the start of a descriptor fetch in either of these modes, the 32-bit DMAx_NEXT_DESC_PTR register is copied into the DMAx_CURR_DESC_PTR register. Then, during the descriptor fetch, the DMAx_CURR_DESC_PTR register increments after each element of the descriptor is read in.

 In small and large descriptor list modes, the DMAx_NEXT_DESC_PTR register, and not the DMAx_CURR_DESC_PTR register, must be programmed directly through MMR access before starting DMA operation.

In descriptor array mode, the next descriptor pointer register is disregarded, and fetching is controlled only by the DMAx_CURR_DESC_PTR register.

Next Descriptor Pointer Registers (DMAx_NEXT_DESC_PTR/MDMA_yy_NEXT_DESC_PTR)

R/W prior to enabling channel; RO after enabling channel

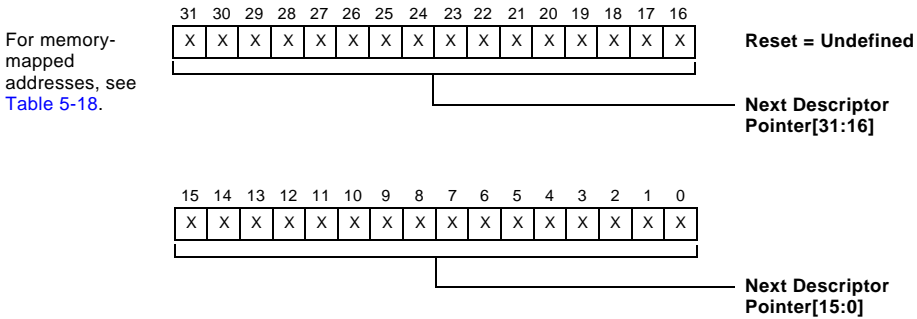


Figure 5-18. Next Descriptor Pointer Registers

Table 5-18. Next Descriptor Pointer Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_NEXT_DESC_PTR	0xFFC0 0C00
DMA1_NEXT_DESC_PTR	0xFFC0 0C40
DMA2_NEXT_DESC_PTR	0xFFC0 0C80
DMA3_NEXT_DESC_PTR	0xFFC0 0CC0
DMA4_NEXT_DESC_PTR	0xFFC0 0D00
DMA5_NEXT_DESC_PTR	0xFFC0 0D40
DMA6_NEXT_DESC_PTR	0xFFC0 0D80
DMA7_NEXT_DESC_PTR	0xFFC0 0DC0
DMA8_NEXT_DESC_PTR	0xFFC0 0E00

DMA Registers

Table 5-18. Next Descriptor Pointer Register Memory-Mapped Addresses
(Cont'd)

Register Name	Memory-Mapped Address
DMA9_NEXT_DESC_PTR	0xFFC0 0E40
DMA10_NEXT_DESC_PTR	0xFFC0 0E80
DMA11_NEXT_DESC_PTR	0xFFC0 0EC0
DMA12_NEXT_DESC_PTR	0xFFC0 1C00
DMA13_NEXT_DESC_PTR	0xFFC0 1C40
DMA14_NEXT_DESC_PTR	0xFFC0 1C80
DMA15_NEXT_DESC_PTR	0xFFC0 1CC0
DMA16_NEXT_DESC_PTR	0xFFC0 1D00
DMA17_NEXT_DESC_PTR	0xFFC0 1D40
DMA18_NEXT_DESC_PTR	0xFFC0 1D80
DMA19_NEXT_DESC_PTR	0xFFC0 1DC0
DMA20_NEXT_DESC_PTR	0xFFC0 1E00
DMA21_NEXT_DESC_PTR	0xFFC0 1E40
DMA22_NEXT_DESC_PTR	0xFFC0 1E80
DMA23_NEXT_DESC_PTR	0xFFC0 1EC0
MDMA_D0_NEXT_DESC_PTR	0xFFC0 0F00
MDMA_S0_NEXT_DESC_PTR	0xFFC0 0F40
MDMA_D1_NEXT_DESC_PTR	0xFFC0 0F80
MDMA_S1_NEXT_DESC_PTR	0xFFC0 0FC0
MDMA_D2_NEXT_DESC_PTR	0xFFC0 1F00
MDMA_S2_NEXT_DESC_PTR	0xFFC0 1F40
MDMA_D3_NEXT_DESC_PTR	0xFFC0 1F80
MDMA_S3_NEXT_DESC_PTR	0xFFC0 1FC0

Current Descriptor Pointer (DMAx_CURR_DESC_PTR and MDMA_yy_CURR_DESC_PTR) Registers

The current descriptor pointer registers and addresses (DMAx_CURR_DESC_PTR and MDMA_yy_CURR_DESC_PTR), shown in [Figure 5-19](#) and [Table 5-19](#), 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 the DMAx_NEXT_DESC_PTR register before loading each descriptor. Then, the address in the DMAx_CURR_DESC_PTR register increments as each descriptor element is read in.

When the entire descriptor is read, the DMAx_CURR_DESC_PTR register contains this value:

Descriptor Start Address + (2 x Descriptor Size) (# of elements)



For descriptor array mode (FLOW = 4), this register, and not the DMAx_NEXT_DESC_PTR register, must be programmed by MMR access before starting DMA operation.

DMA Registers

Current Descriptor Pointer Registers (DMAx_CURR_DESC_PTR/ MDMA_yy_CURR_DESC_PTR)

R/W prior to enabling channel; RO after enabling channel

For memory-mapped addresses, see [Table 5-19](#).

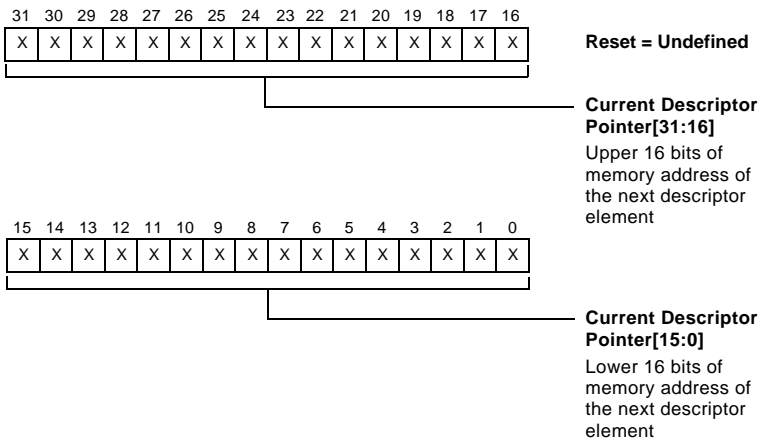


Figure 5-19. Current Descriptor Pointer Registers

Table 5-19. Current Descriptor Pointer Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
DMA0_CURR_DESC_PTR	0xFFC0 0C20
DMA1_CURR_DESC_PTR	0xFFC0 0C60
DMA2_CURR_DESC_PTR	0xFFC0 0CA0
DMA3_CURR_DESC_PTR	0xFFC0 0CE0
DMA4_CURR_DESC_PTR	0xFFC0 0D20
DMA5_CURR_DESC_PTR	0xFFC0 0D60
DMA6_CURR_DESC_PTR	0xFFC0 0DA0
DMA7_CURR_DESC_PTR	0xFFC0 0DE0
DMA8_CURR_DESC_PTR	0xFFC0 0E20
DMA9_CURR_DESC_PTR	0xFFC0 0E60

Table 5-19. Current Descriptor Pointer Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
DMA10_CURR_DESC_PTR	0xFFC0_0EA0
DMA11_CURR_DESC_PTR	0xFFC0_0EE0
DMA12_CURR_DESC_PTR	0xFFC0_1C20
DMA13_CURR_DESC_PTR	0xFFC0_1C60
DMA14_CURR_DESC_PTR	0xFFC0_1CA0
DMA15_CURR_DESC_PTR	0xFFC0_1CE0
DMA16_CURR_DESC_PTR	0xFFC0_1D20
DMA17_CURR_DESC_PTR	0xFFC0_1D60
DMA18_CURR_DESC_PTR	0xFFC0_1DA0
DMA19_CURR_DESC_PTR	0xFFC0_1DE0
DMA20_CURR_DESC_PTR	0xFFC0_1E20
DMA21_CURR_DESC_PTR	0xFFC0_1E60
DMA22_CURR_DESC_PTR	0xFFC0_1EA0
DMA23_CURR_DESC_PTR	0xFFC0_1EE0
MDMA_D0_CURR_DESC_PTR	0xFFC0_0F20
MDMA_S0_CURR_DESC_PTR	0xFFC0_0F60
MDMA_D1_CURR_DESC_PTR	0xFFC0_0FA0
MDMA_S1_CURR_DESC_PTR	0xFFC0_0FE0
MDMA_D2_CURR_DESC_PTR	0xFFC0_1F20
MDMA_S2_CURR_DESC_PTR	0xFFC0_1F60
MDMA_D3_CURR_DESC_PTR	0xFFC0_1FA0
MDMA_S3_CURR_DESC_PTR	0xFFC0_1FE0

Handshake MDMA (HMDMA) Registers

The Blackfin processor features two HMDMA blocks. HMDMA0 is associated with MDMA2, and HMDMA1 is associated with MDMA3.

[Table 5-20](#) lists the naming conventions for these registers.

Table 5-20. Naming Conventions for Handshake MDMA Registers

Handshake MDMA MMR Name (x = 0 or 1)	Memory-Mapped Address
HMDMA0_CONTROL (on page 5-117)	0xFFC0 4500
HMDMA0_ECINIT (on page 5-123)	0xFFC0 4504
HMDMA0_BCINIT (on page 5-120)	0xFFC0 4508
HMDMA0_ECURGENT (on page 5-124)	0xFFC0 450C
HMDMA0_ECOVERFLOW (on page 5-125)	0xFFC0 4510
HMDMA0_ECOUNT (on page 5-121)	0xFFC0 4514
HMDMA0_BCOUNT (on page 5-120)	0xFFC0 4518
HMDMA1_CONTROL (on page 5-117)	0xFFC0 4540
HMDMA1_ECINIT (on page 5-123)	0xFFC0 4544
HMDMA1_BCINIT (on page 5-120)	0xFFC0 4548
HMDMA1_ECURGENT (on page 5-124)	0xFFC0 454C
HMDMA1_ECOVERFLOW (on page 5-125)	0xFFC0 4550
HMDMA1_ECOUNT (on page 5-121)	0xFFC0 4554
HMDMA1_BCOUNT (on page 5-120)	0xFFC0 4558

Handshake MDMA Control (HMDMAx_CONTROL) Registers

The handshake MDMA control registers (HMDMAx_CONTROL), shown in [Figure 5-20](#), set up HMDMA parameters and operating modes.

The DRQ[1:0] field is used to control the priority of the MDMA channel when the HMDMA is disabled, that is, when handshake control is not being used (see [Table 5-21](#)).

Table 5-21. DRQ[1:0] Values

DRQ[1:0]	Priority	Description
b#00	Disabled	The MDMA request is disabled.
b#01	Enabled/S	Normal MDMA channel priority. The channel in this mode is limited to single memory transfers separated by one idle system clock. Request single transfer from MDMA channel.
b#10	Enabled/M	Normal MDMA channel functionality and priority. Request multiple transfers from MDMA channel (default).
b#11	Urgent	The MDMA channel priority is elevated to urgent. In this state, it has higher priority for memory access than non-urgent channels. If two channels are both urgent, the lower-numbered channel has priority.

The RBC bit forces the BCOUNT register to be reloaded with the BCINIT value while the module is already active. Do not set this bit in the same write that sets the HMDMAEN bit to active.

The HMDMA[10:11] bits are used to control the gating of Core, DMAC0, PIXC and MDMA during EPPI urgency conditions. For more information please refer to section "Elevating EPPI Urgent requests at the DDR Controller Interface" in *ADSP-BF54x Blackfin Hardware Reference Volume 2 of 2*.

DMA Registers

Table 5-22. EPPI_DMA_URGENT_ACCESS

HMDMA1[11]	HMDMA[10]	Action
0	0	Do not gate-off core, PIXC or DMAC0 on EPPI(0,1 2) urgent conditions
0	1	Gate-off core only
1	0	Gate-off PIXC, DMAC0 and USB
1	1	Gate-off ALL - core, pixc, DMAC0 and USB

Handshake MDMA Control Registers (HMDMAx_CONTROL)

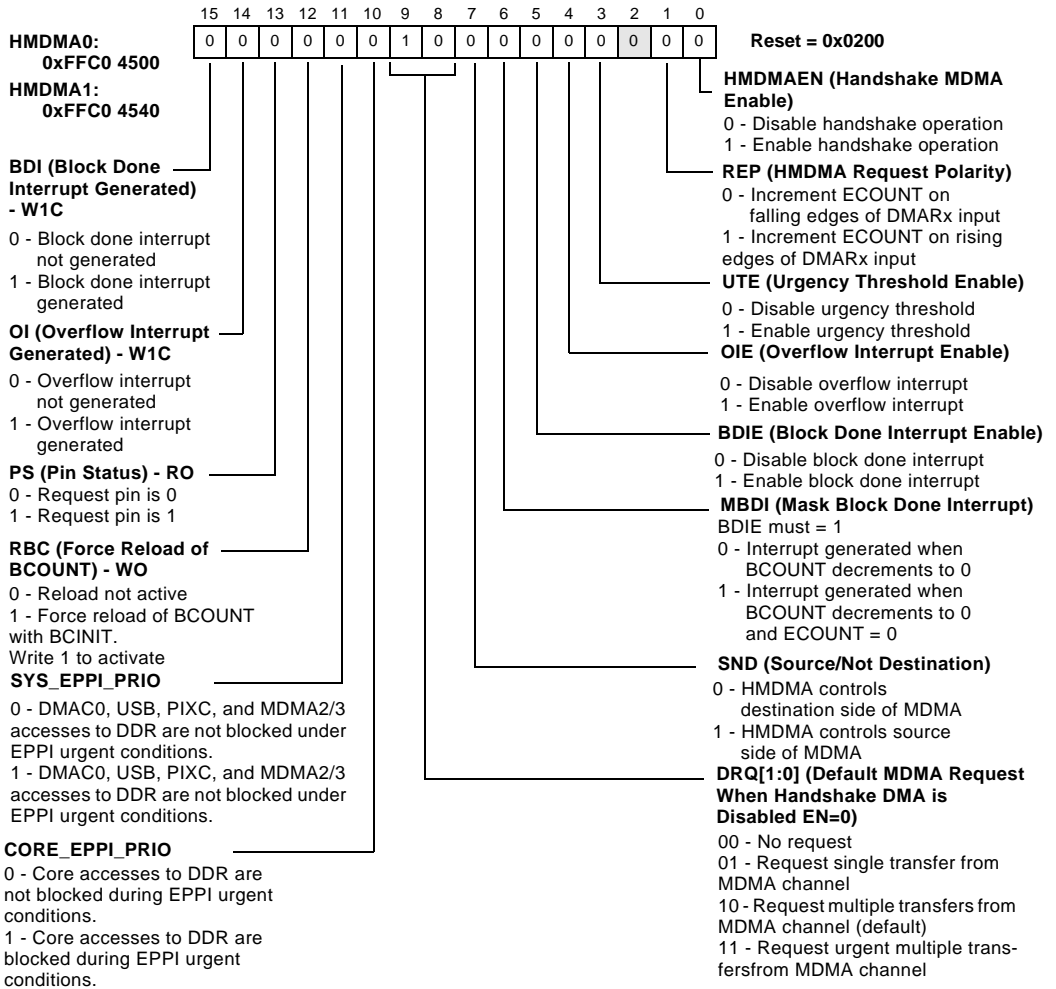


Figure 5-20. Handshake MDMA Control (HMDMAx_CONTROL) Registers

DMA Registers

Handshake MDMA Initial Block Count (HMDMAx_BCINIT) Registers

The handshake MDMA initial block count registers (HMDMAx_BCINIT), shown in [Figure 5-21](#), hold the number of transfers to complete per edge of the DMARx control signal.

Handshake MDMA Initial Block Count Registers (HMDMAx_BCINIT)

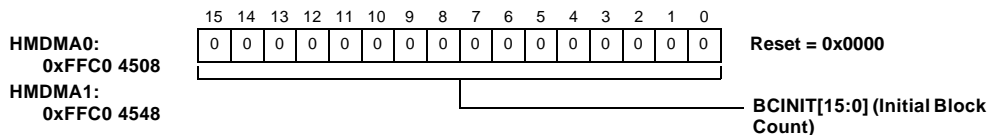


Figure 5-21. Handshake MDMA Initial Block Count (HMDMAx_BCINIT) Registers

Handshake MDMA Current Block Count (HMDMAx_BCOUNT) Registers

The handshake MDMA current block count registers (HMDMAx_BCOUNT), shown in [Figure 5-22](#), hold the number of transfers remaining for the current edge. MDMA requests are generated if this count is greater than 0.

Examples:

- 0x0000 = 0 transfers remaining
- 0xFFFF = 65535 transfers remaining

The BCOUNT field is loaded with BCINIT when ECOUNT is greater than 0 and BCOUNT is expired (0). Also, if the RBC bit in the HMDMAx_CONTROL register is written to a 1, BCOUNT is loaded with BCINIT. The BCOUNT field is decremented with each MDMA grant. It is cleared when HMDMA is disabled.

A block done interrupt is generated when `BCOUNT` decrements to 0. If the `MBDI` bit in the `HMDMAx_CONTROL` register is set, the interrupt is suppressed until `ECOUNT` is 0. Note if `BCINIT` is 0, no block done interrupt is generated, since no DMA requests were generated or grants received.

Handshake MDMA Current Block Count Register (HMDMAx_BCOUNT)

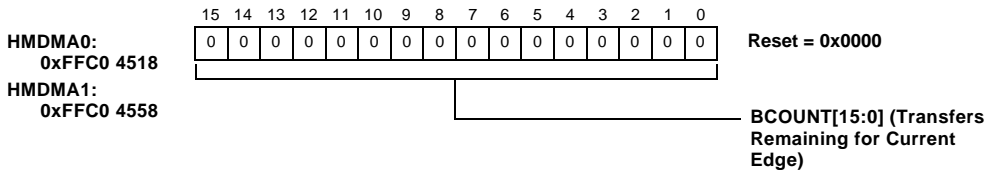


Figure 5-22. Handshake MDMA Current Block Count (HMDMAx_BCOUNT) Registers

Handshake MDMA Current Edge Count (HMDMAx_ECOUNTER) Registers

The handshake MDMA current edge count registers (`HMDMAx_ECOUNTER`), shown in [Figure 5-23](#), hold a signed number of edges remaining to be serviced. This number is in a signed, two's-complement representation. An edge is detected on the respective `DMARx` input. Requests occur if this count is greater than or equal to 0, and `BCOUNT` is greater than 0.

When the handshake mode is enabled, `ECOUNT` is loaded and the resulting number of requests is:

Number of edges + N,

where N is the number loaded from `ECINIT`. The number N is a positive or negative signed number.

DMA Registers

Examples:

- $0x7FFF = 32767$ edges remaining
- $0x0000 = 0$ edges remaining
- $0x8000 = -32768$: ignore the next 32768 edges

Each time that `BCOUNT` expires, `ECOUNT` is decremented and `BCOUNT` is reloaded from `BCINIT`. When a handshake request edge is detected, `ECOUNT` is incremented. The `ECOUNT` field is cleared when `HMDMA` is disabled.

Handshake MDMA Current Edge Count Register (HMDMAx_ECOUNTER)

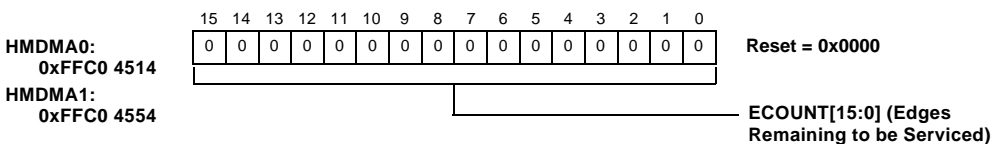


Figure 5-23. Handshake MDMA Current Edge Count (HMDMAx_ECOUNTER) Registers

Handshake MDMA Initial Edge Count (HMDMAx_ECINIT) Registers

The handshake MDMA initial edge count registers (HMDMAx_ECINIT), shown in [Figure 5-24](#), hold a signed number that is loaded into current edge count (HMDMAx_ECOUNTER) when the handshake DMA is enabled. This number is in a signed, two's-complement representation.

Handshake MDMA Initial Edge Count Registers (HMDMAx_ECINIT)

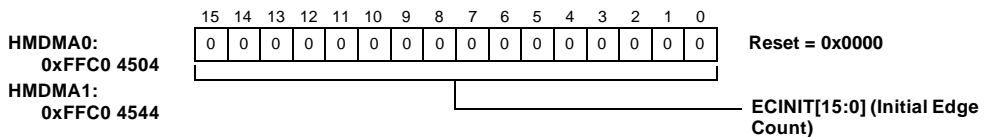


Figure 5-24. Handshake MDMA Initial Edge Count (HMDMAx_ECINIT) Registers

Handshake MDMA Edge Count Urgent (HMDMAx_ECURGENT) Registers

The handshake MDMA edge count urgent registers (HMDMAx_ECURGENT), shown in [Figure 5-25](#) and, hold the urgent threshold. If the ECOUNT field in the handshake MDMA edge count register is greater than this threshold, the MDMA request is urgent and might get higher priority.

Handshake MDMA Edge Count Urgent Registers (HMDMAx_ECURGENT)

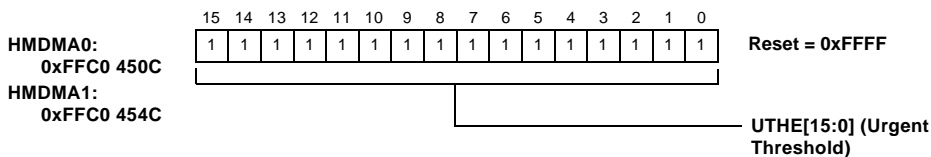


Figure 5-25. Handshake MDMA Edge Count Urgent (HMDMAx_ECURGENT) Registers

Handshake MDMA Edge Count Overflow Interrupt (HMDMAx_ECOVERFLOW) Registers

The handshake MDMA edge count overflow interrupt registers, (HMDMAx_ECOVERFLOW), shown in [Figure 5-26](#), hold the interrupt threshold. If the ECOUNT field in the handshake MDMA edge count register is greater than this threshold, an overflow interrupt is generated.

Handshake MDMA Edge Count Overflow Interrupt Registers (HMDMAx_ECOVERFLOW)

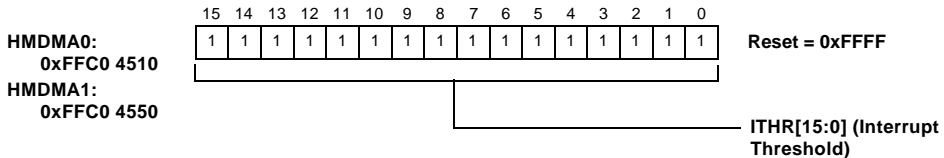


Figure 5-26. Handshake MDMA Edge Count Overflow Interrupt (HMDMAx_ECOVERFLOW) Registers

DMA Traffic Control Registers

The DMACx_TCPER registers and the DMACx_TCCNT registers work with other DMA registers to define traffic control.

i Traffic control works within one DMA controller (DMAC0 or DMAC1), not between DMA controllers.

Table 5-23. DMA Traffic Control Registers

Register Name	Refer to	Memory-Mapped Address
DMAC0_TCPER	Listing on page 5-126	0xFFC0 0B0C
DMAC0_TCCNT	Listing on page 5-127	0xFFC0 0B10
DMAC1_TCPER	Listing on page 5-126	0xFFC0 1B0C
DMAC1_TCCNT	Listing on page 5-127	0xFFC0 1B10

DMA Registers

This section also describes the `DMAC1_PERIMUX` register [on page 5-129](#).

DMA Traffic Control Counter Period (DMACx_TCPER) Registers

The DMA traffic control counter period registers (`DMACx_TCPER`) are shown in [Figure 5-27](#).

DMA Traffic Control Counter Period Register (DMACx_TCPER)

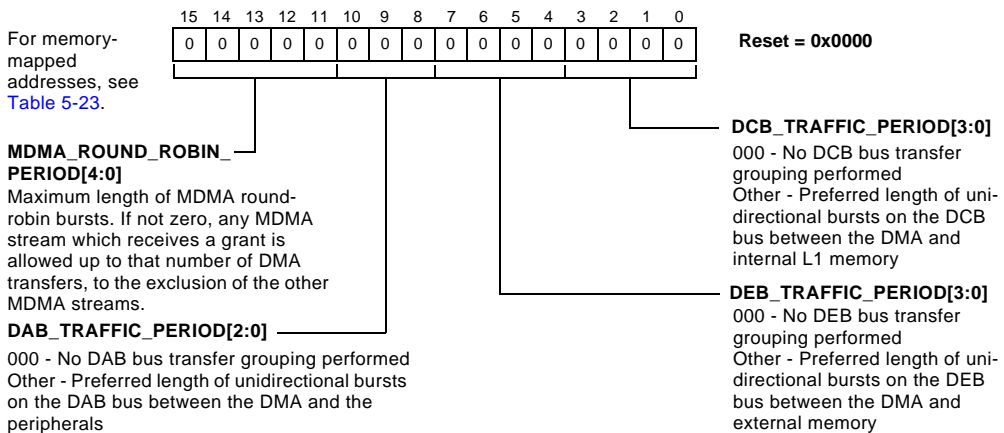


Figure 5-27. DMA Traffic Control Counter Period (DMACx_TCPER) Registers

DMA Traffic Control Counter (DMACx_TCCNT) Registers

The DMA traffic control counter registers (DMACx_TCCNT) are shown in Figure 5-28.

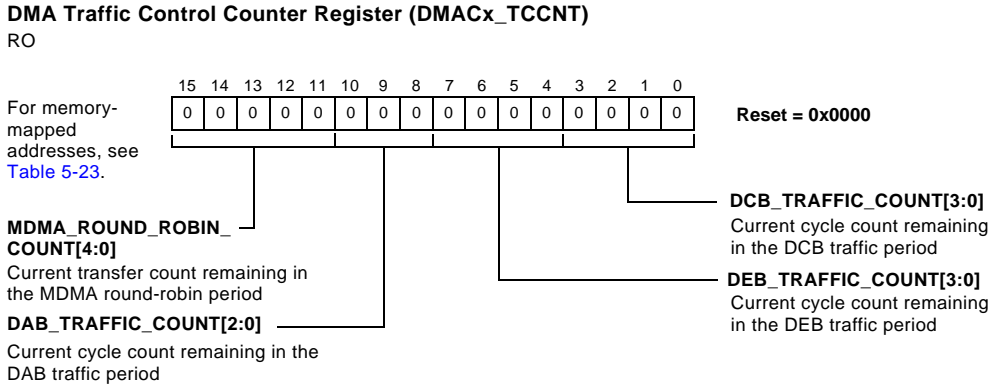


Figure 5-28. DMA Traffic Control Counter (DMACx_TCCNT) Registers

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_TCPER` is written, whenever a different MDMA stream is granted, or whenever every MDMA stream is idle. It then counts down to 0 with each MDMA transfer. When this count decrements from 1 to 0, the next available MDMA stream is selected.

The `DAB_TRAFFIC_COUNT` field shows the current cycle count remaining in the DAB traffic period. It initializes to `DAB_TRAFFIC_PERIOD` whenever `DMACx_TCPER` is written, or whenever the DAB bus changes direction or becomes idle. It then counts down from `DAB_TRAFFIC_PERIOD` to 0 on each system clock (except for DMA stalls). While this count is nonzero, same direction DAB accesses are treated preferentially. When this count decrements from 1 to 0, the opposite direction DAB access is treated

DMA Registers

preferentially, which may result in a direction change. When this count is 0 and a DAB bus access occurs, the count is reloaded from `DAB_TRAFFIC_PERIOD` to begin a new burst.

The `DEB_TRAFFIC_COUNT` field shows the current cycle count remaining in the DEB traffic period. It initializes to `DEB_TRAFFIC_PERIOD` whenever `DMACX_TCPER` is written, or whenever the DEB bus changes direction or becomes idle. It then counts down from `DEB_TRAFFIC_PERIOD` to 0 on each system clock (except for DMA stalls). While this count is nonzero, same direction DEB accesses are treated preferentially. When this count decrements from 1 to 0, the opposite direction DEB access is treated preferentially, which may result in a direction change. When this count is 0 and a DEB bus access occurs, the count is reloaded from `DEB_TRAFFIC_PERIOD` to begin a new burst.

The `DCB_TRAFFIC_COUNT` field shows the current cycle count remaining in the DCB traffic period. It initializes to `DCB_TRAFFIC_PERIOD` whenever `DMACX_TCPER` is written, or whenever the DCB bus changes direction or becomes idle. It then counts down from `DCB_TRAFFIC_PERIOD` to 0 on each system clock (except for DMA stalls). While this count is nonzero, same direction DCB accesses are treated preferentially. When this count decrements from 1 to 0, the opposite direction DCB access is treated preferentially, which may result in a direction change. When this count is 0 and a DCB bus access occurs, the count is reloaded from `DCB_TRAFFIC_PERIOD` to begin a new burst.

DMA Controller 1 Peripheral Multiplexer (DMAC1_PERIMUX) Register

The DMAC1_PERIMUX register is shown in [Figure 5-29](#).

DMA Controller 1 Peripheral Multiplexer Register (DMAC1_PERIMUX)

R/W

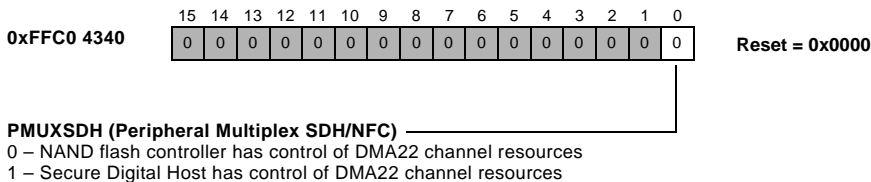


Figure 5-29. DMA Controller 1 Peripheral Multiplexer (DMAC1_PERIMUX) Register

The DMAC1_PERIMUX register controls the common sharing of a single DMA channel between the NAND flash controller (NFC) and the secure digital host (SDH) module. The sharing of this resource prevents the simultaneous use of the NFC and the SDH with DMA access to internal and external memory. DMAC1_PERIMUX controls the peripheral that gains access to DMA resources. DMAC1_PERIMUX is a 16-bit wide register and requires 16-bit access.

Programming Examples

The following examples illustrate memory DMA and handshaked memory DMA basics. Examples for peripheral DMAs can be found in the respective peripheral chapters in Volume 2.

Register-Based 2D Memory DMA

Listing 5-1 shows a register-based, two-dimensional MDMA. While the source channel processes linearly, the destination channel re-sorts elements of the two-dimensional data array. See Figure 5-30.

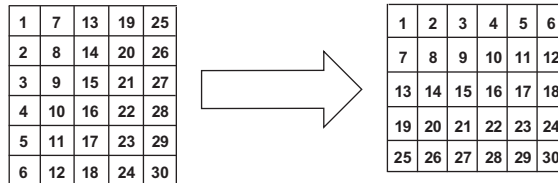


Figure 5-30. DMA Example, 2D Array

The two arrays reside in two different L1 data memory blocks. However, the arrays could reside in any internal or external memory, including L1 instruction memory, memory other than L1, and DDR SDRAM. For the case where the destination array resides in DDR SDRAM, it is a good idea to let the source channel re-sort elements and to let the destination buffer store linearly.

Listing 5-1. Register-Based 2-D Memory DMA

```
#include <defBF54x.h>
#define X 5
#define Y 6

.section L1_data_a;
.byte2 aSource[X*Y] =
    1,  7, 13, 19, 25,
    2,  8, 14, 20, 26,
    3,  9, 15, 21, 27,
    4, 10, 16, 22, 28,
    5, 11, 17, 23, 29,
```

```

        6, 12, 18, 24, 30;

.section L1_data_b;
.byte2 aDestination[X*Y];

.section L1_code;
.global _main;
_main:
    p0.l = lo(MDMA_SO_CONFIG);
    p0.h = hi(MDMA_SO_CONFIG);
    call memdma_setup;
    call memdma_wait;
_main.forever:
    jump _main.forever;
_main.end:

```

The setup routine shown in [Listing 5-2](#) initializes either MDMA0 or MDMA1 depending on whether the MMR address of `MDMA_SO_CONFIG` or `MDMA_S1_CONFIG` is passed in the `P0` register. Note that the source channel is enabled before the destination channel. Also, it is common to synchronize interrupts with the destination channel, because only those interrupts indicate completion of both DMA read and write operations.

Listing 5-2. 2-D Memory DMA Setup Example

```

memdma_setup:
    [--sp] = r7;
/* setup 1D source DMA for 16-bit transfers */
    r7.l = lo(aSource);
    r7.h = hi(aSource);
    [p0 + MDMA_SO_START_ADDR - MDMA_SO_CONFIG] = r7;
    r7.l = 2;
    w[p0 + MDMA_SO_X_MODIFY - MDMA_SO_CONFIG] = r7;
    r7.l = X * Y;
    w[p0 + MDMA_SO_X_COUNT - MDMA_SO_CONFIG] = r7;

```

Programming Examples

```
    r7.l = WDSIZE_16 | DMAEN;
    w[p0] = r7;
/* setup 2D destination DMA for 16-bit transfers */
    r7.l = lo(aDestination);
    r7.h = hi(aDestination);
    [p0 + MDMA_D0_START_ADDR - MDMA_S0_CONFIG] = r7;
    r7.l = 2*Y;
    w[p0 + MDMA_D0_X_MODIFY - MDMA_S0_CONFIG] = r7;
    r7.l = Y;
    w[p0 + MDMA_D0_Y_COUNT - MDMA_S0_CONFIG] = r7;
    r7.l = X;
    w[p0 + MDMA_D0_X_COUNT - MDMA_S0_CONFIG] = r7;
    r7.l = -2 * (Y * (X-1) - 1);
    w[p0 + MDMA_D0_Y_MODIFY - MDMA_S0_CONFIG] = r7;
    r7.l = DMA2D | DI_EN | WDSIZE_16 | WNR | DMAEN;
    w[p0 + MDMA_D0_CONFIG - MDMA_S0_CONFIG] = r7;
    r7 = [sp++];
    rts;
memdma_setup.end:
```

For simplicity, the example shown in [Listing 5-3](#) polls the DMA status rather than using interrupts, which is the normal case in a real application.

Listing 5-3. Polling DMA Status

```

memdma_wait:
    [--sp] = r7;
memdma_wait.test:
    r7 = w[p0 + MDMA_D0_IRQ_STATUS - MDMA_S0_CONFIG] (z);
    CC = bittst (r7, bitpos(DMA_DONE));
    if !CC jump memdma_wait.test;
    r7 = DMA_DONE (z);
    w[p0 + MDMA_D0_IRQ_STATUS - MDMA_S0_CONFIG] = r7;
    r7 = [sp++];
    rts;
memdma_wait.end:

```

Initializing Descriptors in Memory

Descriptor-based DMAs expect the descriptor data to be available in memory by the time the DMA is enabled. Often, the descriptors are programmed by software at run time. Many times, however, the descriptors—or at least large portions of them—can be static and therefore initialized at boot time. How to set up descriptors in global memory depends heavily on the programming language and the toolset used. The following examples show how this is best performed in the VisualDSP++ tools’ assembly language.

[Listing 5-4](#) uses multiple variables of either 16-bit or 32-bit size to describe DMA descriptors. This example has two descriptors in small list flow mode that point to each other mutually. At the end of the second work unit an interrupt is generated without discontinuing the DMA processing. The trailing “.end” label is required to let the linker know that a descriptor forms a logical unit. It prevents the linker from removing variables when optimizing.

Programming Examples

Listing 5-4. Two Descriptors in Small List Flow Mode

```
.section sdram;
.byte2 arrBlock1[0x400];
.byte2 arrBlock2[0x800];

.section L1_data_a;
.byte2 descBlock1 = 1o(descBlock2);
.var descBlock1.addr = arrBlock1;
.byte2 descBlock1.cfg = FLOW_SMALL|NDSIZE_5|WDSIZE_16|DMAEN;
.byte2 descBlock1.len = length(arrBlock1);
    descBlock1.end:

.byte2 descBlock2 = 1o(descBlock1);
.var descBlock2.addr = arrBlock2;
.byte2 descBlock2.cfg =
FLOW_SMALL|NDSIZE_5|DI_EN|WDSIZE_16|DMAEN;
.byte2 descBlock2.len = length(arrBlock2);
    descBlock2.end:
```

Another method featured by the VisualDSP++ tools takes advantage of C-style structures in global header files. The header file `descriptor.h` could look like [Listing 5-5](#).

Listing 5-5. Header File to Define Descriptor Structures

```
#ifndef __INCLUDE_DESCRIPTOR__
#define __INCLUDE_DESCRIPTOR__
#ifdef _LANGUAGE_C
typedef struct {
    void *pStart;
    short dConfig;
    short dxCount;
    short dxModify;
    short dyCount;
```

```

        short dYModify;
    } dma_desc_arr;

typedef struct {
    void *pNext;
    void *pStart;
    short dConfig;
    short dxCount;
    short dxModify;
    short dYCount;
    short dYModify;
} dma_desc_list;

#endif // _LANGUAGE_C
#endif // __INCLUDE_DESCRIPTOR__

```

Note that near pointers are not natively supported by the C language, pointers are always 32 bits wide. Therefore, the scheme above cannot be used directly for small list mode without giving up pointer syntax. The variable definition file is required to import the C-style header file and can finally take advantage of the structures. See [Listing 5-6](#).

Listing 5-6. Using Descriptor Structures

```

#include "descriptors.h"
.import "descriptors.h";

.section L1_data_a;
.align 4;
.var arrBlock3[N];
.var arrBlock4[N];

.struct dma_desc_list descBlock3 = {
    descBlock4, arrBlock3,
    FLOW_LARGE | NDSIZE_7 | WDSIZE_32 | DMAEN,

```

Programming Examples

```
length(arrBlock3), 4,  
0, 0 /* unused values */  
};  
  
.struct dma_desc_list descBlock4 = {  
    descBlock3, arrBlock4,  
    FLOW_LARGE | NDSIZE_7 | DI_EN | WDSIZE_32 | DMAEN,  
    length(arrBlock4), 4,  
    0, 0 /* unused values */  
};
```

Software-Triggered Descriptor Fetch Example

[Listing 5-7](#) demonstrates a large list of descriptors that provide flow stop mode configuration. Consequently, the DMA stops by itself as soon as the work unit has finished. Software triggers the next work unit by simply writing the proper value into the DMA configuration registers. Since these values instruct the DMA controller to fetch descriptors in large list mode, after being started, the DMA immediately fetches the descriptor and then overwrites the configuration value again with the new settings.

Note the requirement that source and destination channels stop after the same number of transfers. In between stops the two channels can have completely individual structure.

Listing 5-7. Software-Triggered Descriptor Fetch

```
#define N 4  
.section L1_data_a;  
.byte2 arrSource1[N] = { 0x1001, 0x1002, 0x1003, 0x1004 };  
.byte2 arrSource2[N] = { 0x2001, 0x2002, 0x2003, 0x2004 };  
.byte2 arrSource3[N] = { 0x3001, 0x3002, 0x3003, 0x3004 };  
.byte2 arrDest1[N];
```

```

.byte2 arrDest2[2*N];

.struct dma_desc_list descSource1 = {
    descSource2, arrSource1,
    WDSIZE_16 | DMAEN,
    length(arrSource1), 2,
    0, 0 /* unused values */
};

.struct dma_desc_list descSource2 = {
    descSource3, arrSource2,
    FLOW_LARGE | NDSIZE_7 | WDSIZE_16 | DMAEN,
    length(arrSource2), 2,
    0, 0 /* unused values */
};

.struct dma_desc_list descSource3 = {
    descSource1, arrSource3,
    WDSIZE_16 | DMAEN,
    length(arrSource3), 2,
    0, 0 /* unused values */
};

.struct dma_desc_list descDest1 = {
    descDest2, arrDest1,
    DI_EN | WDSIZE_16 | WNR | DMAEN,
    length(arrDest1), 2,
    0, 0 /* unused values */
};

.struct dma_desc_list descDest2 = {
    descDest1, arrDest2,
    DI_EN | WDSIZE_16 | WNR | DMAEN,
    length(arrDest2), 2,
    0, 0 /* unused values */
};

.section L1_code;

```

Programming Examples

```
_main:
/* write descriptor address to next descriptor pointer */
    p0.h = hi(MDMA_SO_CONFIG);
    p0.l = lo(MDMA_SO_CONFIG);
    r0.h = hi(descDest1);
    r0.l = lo(descDest1);
    [p0 + MDMA_DO_NEXT_DESC_PTR - MDMA_SO_CONFIG] = r0;
    r0.h = hi(descSource1);
    r0.l = lo(descSource1);
    [p0 + MDMA_SO_NEXT_DESC_PTR - MDMA_SO_CONFIG] = r0;

/* start first work unit */
    r6.l = FLOW_LARGE|NDSIZE_7|WDSIZE_16|DMAEN;
    w[p0 + MDMA_SO_CONFIG - MDMA_SO_CONFIG] = r6;
    r7.l = FLOW_LARGE|NDSIZE_7|WDSIZE_16|WNR|DMAEN;
    w[p0 + MDMA_DO_CONFIG - MDMA_SO_CONFIG] = r7;

/* wait until destination channel has finished and WIC latch */
_main.wait:
    r0 = w[p0 + MDMA_DO_IRQ_STATUS - MDMA_SO_CONFIG] (z);
    CC = bittst (r0, bitpos(DMA_DONE));
    if !CC jump _main.wait;
    r0.l = DMA_DONE;
    w[p0 + MDMA_DO_IRQ_STATUS - MDMA_SO_CONFIG] = r0;

/* wait for any software or hardware event here */

/* start next work unit */
    w[p0 + MDMA_SO_CONFIG - MDMA_SO_CONFIG] = r6;
    w[p0 + MDMA_DO_CONFIG - MDMA_SO_CONFIG] = r7;
    jump _main.wait;
_main.end:
```

Handshake Memory DMA Example

The functional block for the handshake MDMA operation can be seen completely separately from the MDMA channels themselves. Therefore the following HMDMA setup routine can be combined with any of the MDMA examples discussed above. Be sure that the HMDMA module is enabled before the MDMA channels.

[Listing 5-8](#) enables the HMDMA1 block which is controlled by the DMAR1 pin and is associated with the MDMA1 channel pair.

Listing 5-8. HMDMA1 Block Enable

```

/* optionally, enable all four bank select strobes */
    p1.l = lo(EBIU_AMGCTL);
    p1.h = hi(EBIU_AMGCTL);
    r0.l = 0x0009;
    w[p1] = r0;

/* function enable for DMAR1 */
    p1.l = lo(PORTH_FER);
    r0.l = PH6;
    w[p1] = r0;
    p1.l = lo(PORTH_MUX);
    r0.l = lo(MUX6_1);
    r0.h = hi(MUX6_1);
    [p1] = r0;

/* every single transfer requires one DMAR1 event */
    p1.l = lo(HMDMA1_BCINIT);
    r0.l = 1;
    w[p1] = r0;

/* start with balanced request counter */
    p1.l = lo(HMDMA1_ECINIT);

```

Programming Examples

```
r0.l = 0;
w[p1] = r0;

/* enable for rising edges */
p1.l = 1o(HMDMA1_CONTROL);
r2.l = REP | HMDMAEN;
w[p1] = r2;
```

If the HMDMA intent is to copy from internal memory to external devices, the above setup is appropriate. It controls the Memory DMA's destination channel. If the intent is to read data from external memory, set the **SND** bit in the `HMDMAx_CONTROL` register to control the source channel instead.

6 EXTERNAL BUS INTERFACE UNIT

The external bus interface unit (EBIU) provides a glueless interface to a variety of external memories. The EBIU supports both synchronous and asynchronous memories. The synchronous interface supports dual data rate (DDR) SDRAM memories. The asynchronous interface supports memories such as SRAM and flash memories including synchronous NOR flash.

The synchronous interface is controlled by a DDR controller. The asynchronous interface is controlled by the asynchronous memory controller (AMC). The asynchronous interface is further shared by an on-chip NAND flash controller and an ATAPI controller. The ATAPI and the NAND flash controllers are not part of EBIU; they just share the asynchronous interface pins. An asynchronous pin control module (APCM) controls and arbitrates the asynchronous interface between the ASYNC, NAND, and ATAPI controllers.

The chapter includes the following sections:

- [“General Overview” on page 6-2](#)
- [“DDR Arbitration” on page 6-12](#)
- [“DDR SDRAM Controller” on page 6-16](#)
- [“DDR SDRAM Memory Interface” on page 6-17](#)
- [“DDR Registers” on page 6-20](#)
- [“DDR Metrics Control Registers” on page 6-42](#)

General Overview

- [“Asynchronous Memory Interface”](#) on page 6-53
- [“ASYNC Interface Control Registers”](#) on page 6-56

General Overview

The EBIU services requests for external memory from the Blackfin core and from three on-chip DMA controllers (DMAC0, DMAC1, and USB DMA). An address decoder inside EBIU determines whether the request is serviced by the DDR memory controller or the asynchronous memory controller and routes the requests to the appropriate controller. Requests from different sources are prioritized based on a programmable priority scheme.

The EBIU is clocked by the system clock (SCLK), which runs at a maximum frequency that is specified in the ADSP-BF54x processor data sheet. All DDR SDRAM memories interfaced to the device operate at SCLK frequency.

The external memory space is shown in [Figure 6-1](#). Two of the memory regions are dedicated to DDR SDRAM. The DDR SDRAM interface timing and the size of each DDR SDRAM region are programmable. Each external DDR SDRAM bank can be populated up to 256M bytes. The start address of bank 0 is 0x0000 0000 and the start address of bank 1 follows contiguously from the previous bank. Depending upon the memory configuration, the area from the end of bank 1 to address 0x2000 0000 is reserved.

The next four regions are dedicated to support 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. Also, each of the asynchronous memory regions can be independently programmed to support burst mode or page mode flash memories.

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. When either of the DMAC0, DMAC1, or the USB DMA controllers address this region, the EBIU sends an error response on the internal buses to the controllers. The EBIU generates the hardware error (HWE) interrupt to the core when it is requested to access this reserved off-chip memory space.

General Overview

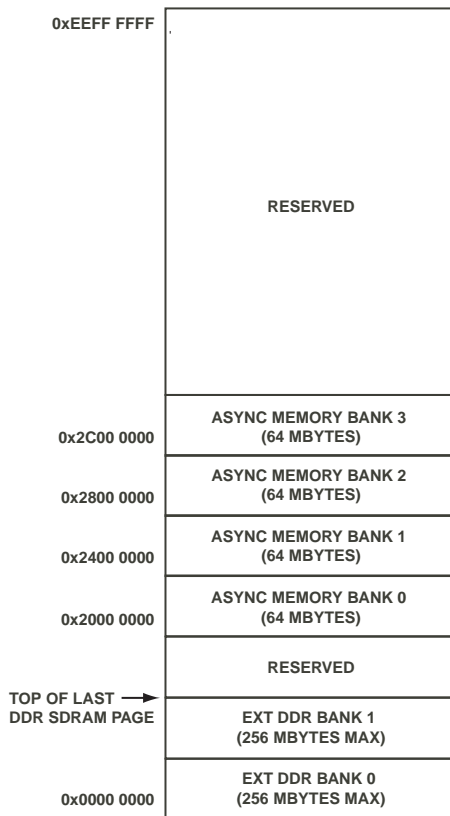


Figure 6-1. External Memory Map

Block Diagram

Figure 6-2 shows a conceptual block diagram of the EBIU. Note that the pins for the synchronous DDR memory interface are dedicated, whereas pins for the asynchronous memories are shared.

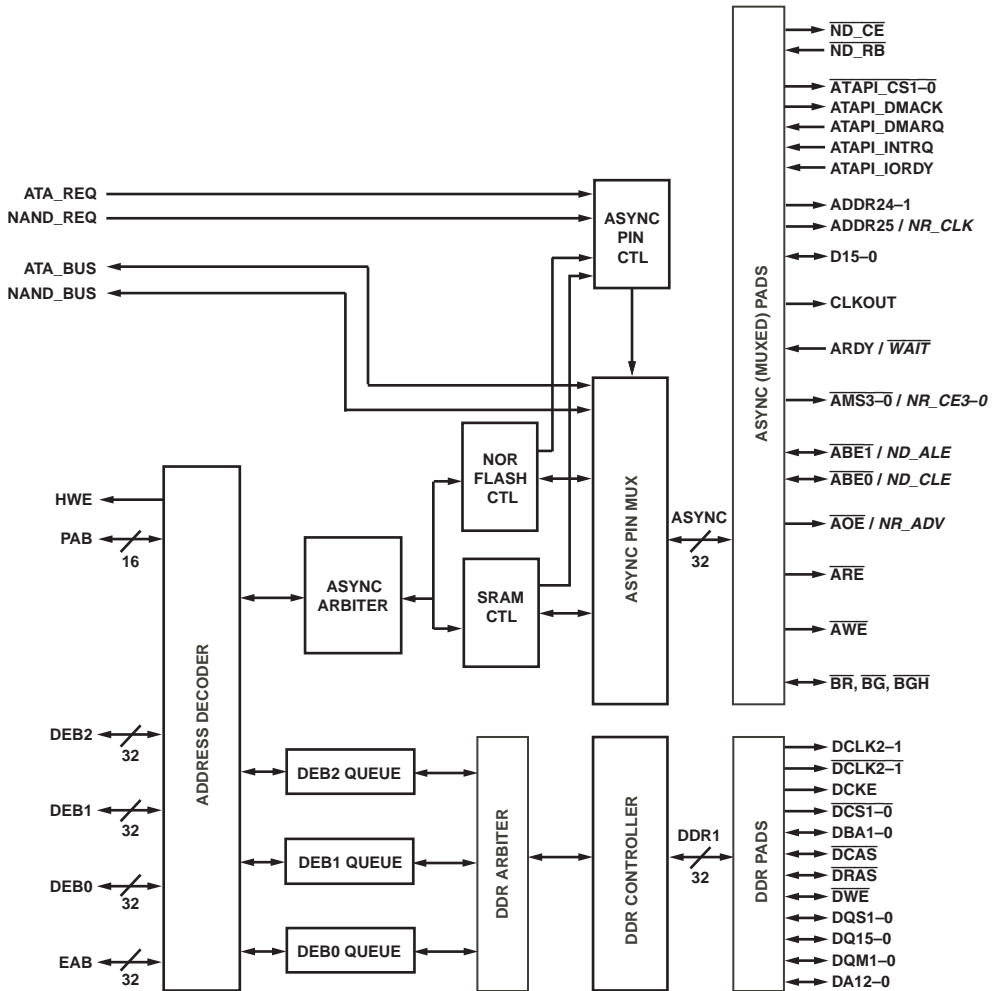


Figure 6-2. External Bus Interface Unit (EBIU) Diagram

General Overview

The EBIU allows the on-chip NAND flash controller and ATAPI controller to share its asynchronous interface pins. An asynchronous pin control module (APCM) in the EBIU automatically controls the accesses to the asynchronous memory interface pins, based on requests from the AMC, NAND, and ATAPI with a set priority. No extra configuration is needed. The multiplexing scheme of the shared pins is summarized in [Table 6-1](#). When reading [Table 6-1](#), note that an “x” indicates that the pin is used by the interface, a “–” indicates that the pin is not used by the interface, and an alternate pin name indicates that the pin is used for an alternate function by the interface.

Table 6-1. EBIU Pin List (With Multiplexing)

Pins	ASYNC	FLASH	NAND FLASH	ATAPI	DDR
ADDR24–1	x	x	–	x ¹	–
ADDR25	x	NR_CLK	–	-	–
D15–0	x	x	x	x	–
$\overline{\text{AMS3–0}}$	x	NR_CE3–0	–	–	–
$\overline{\text{ABE0}}$	x	–	ND_CLE	–	–
$\overline{\text{ABE1}}$	x	–	ND_ALE	–	–
$\overline{\text{AOE}}$	x	NR_ADV	–	–	–
$\overline{\text{ARE}}$	x	x	x	-	–
$\overline{\text{AWE}}$	x	x	x	-	–
ARDY	x	$\overline{\text{WAIT}}$	–	–	–
CLKOUT	x	–	–	–	–
$\overline{\text{ND_CE}}$	–	–	x	–	–
$\overline{\text{ND_RB}}$	–	–	x	–	–
$\overline{\text{ATAPI_CS1–0}}$	–	–	–	x	–
ATAPI_DMACK	–	–	–	x	–

Table 6-1. EBIU Pin List (With Multiplexing) (Cont'd)

Pins	ASYNC	FLASH	NAND FLASH	ATAPI	DDR
ATAPI_INTR	–	–	–	x	–
ATAPI_DMARQ	–	–	–	x	–
ATAPI_IORDY	–	–	–	x	–
$\overline{\text{BR}}$	x	–	–	–	–
$\overline{\text{BG}}$	x	–	–	–	–
$\overline{\text{BGH}}$	x	–	–	–	–
DCLK2–1	–	–	–	–	x
$\overline{\text{DCLK2-1}}$	–	–	–	–	x
DCKE	–	–	–	–	x
$\overline{\text{DCS1-0}}$	–	–	–	–	x
DBA1–0	–	–	–	–	x
$\overline{\text{DCAS}}$	–	–	–	–	x
$\overline{\text{DRAS}}$	–	–	–	–	x
$\overline{\text{DWE}}$	–	–	–	–	x
DQS1–0	–	–	–	–	x
DQ15–0	–	–	–	–	x
DQM1–0	–	–	–	–	x
DA12–0	–	–	–	–	x

- 1 Note that some of the pins listed in Table 6-1 are multiplexed with GPIO, especially the address lines ADDR4_ADDR25. So please set the general purpose port multiplexing before using them as asynchronous memory interface, NAND flash interface, or ATAPI interface. Please refer to Chapter 9, General Purpose Ports for more information.

On-Chip System Interfaces

The EBIU functions as a slave on five buses internal to the ADSP-BF54x processor, as follows:

- A 32-bit external access bus (EAB), mastered by the core, for external memory access
- A 16-bit DMA external bus (DEB0), mastered by DMA controller1, in response to external memory access requests from any DMAC0 (16-bit) channel
- A 32-bit DMA external bus (DEB1), mastered by DMA controller2, in response to external memory access request from any DMAC1 (32-bit) channel
- A 32-bit DMA external bus (DEB2), mastered by the DMA controller in the USB module
- A 16-bit PAB bus, mastered by the core, to access the system memory-mapped registers (SMMR) in the EBIU

These are synchronous interfaces, clocked by `SCLK`. The EAB, DEB0, DEB1, and DEB2 (USB) provide access to both synchronous DDR SDRAM and asynchronous external memories, including page mode and burst mode NOR flash memories.

Error Detection

The EBIU responds to any bus operation that addresses the range of `0x0000 0000 – 0xEEFF FFFF`, even if that bus operation addresses reserved or disabled memory. 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 the following error conditions:

- Any access to the reserved off-chip memory space
- Any access to disabled external memory bank
- Any access to an unpopulated area of a DDR SDRAM memory bank

If the core requested the faulting bus operation, the bus error response from the EBIU is routed to the HWE interrupt internal to the core. If the DMA master issues the request for the faulting bus operation, then the bus error is captured in that controller and can optionally generate an interrupt to the core. In both cases, the error address is latched in the corresponding EBIU error address register.

System Arbitration

As mentioned earlier, the EBIU implements two different memory interfaces that provide simultaneous accesses to DDR SDRAM and asynchronous memory in response to requests on any of the four internal data access buses. For example, while the DDR controller services a core request to DDR SDRAM memory, the AMSYNC could service a DMA request to asynchronous or flash memory. Although the synchronous and asynchronous memories run at different speeds, the EBIU ensures that data is returned to the requestor in the correct order.

To take advantage of the high performance DDR interface and the independent asynchronous memory interface, and to maintain correct order of data transfers on the internal buses, the EBIU implements some arbitration modules that augment the DDR controller and the AMSYNC memory.

Address Resolution

The EBIU address decoder block accepts the commands (read/writes) from the EAB and DMA buses (DEB0, DEB1, and DEB2). It then processes them and transfers them to the DDR queue manager (QM) block or the asynchronous memory controller block based on the address being accessed.

If the address happens to be in the reserved region (based on the memory configuration), it generates accordingly the required number of acknowledgements along with the error signal.

Reorder Unit

Because of simultaneous support of varying speed interfaces, there is a reorder engine in the EBIU for each of the system buses (DEB0, DEB1, DEB2, and EAB). The reorder engine handles out-of-order responses and makes sure that all responses from the interfaces (DDR SDRAM, asynchronous SRAM/flash) are still in the same order in which they were accepted and issued. For all read accesses, it keeps track of the states of all the requests that went to the EBIU controllers and makes sure that the responses are sent back to the original requestors in order. For write requests, each queue maintains the order in which the responses were transferred with the bus.

The following example shows out-of-order execution between the DDR interface and the asynchronous memory controller (ASYNC) interface.

The order in which the requests are accepted and issued to the controllers is as follows:

- Cycle 1: ASYNC Read Request-1
- Cycle 2: DDR Read Request-1
- Cycle 3: DDR Read Request-2

- Cycle 4: DDR Read Request-3
- Cycle 5: DDR Read Request-4

Since the DDR interface is much faster than the ASYNC interface, the DDR read data is be available from the DDR QM block much earlier than the ASYNC interface. So the reorder engine instructs the DDR QM to stop giving the read data and hold it until the ASYNC read data is available.

- Cycle 4: DDR Read Data-1 is available but is blocked and stored in DDR QM block
- Cycle 5: DDR Read Data-2 is available but is blocked and stored in DDR QM block
- Cycle 6: ASYNC Read Data-1 is available and DDR Read Data-3 is available
- Only ASYNC Read Data-1 is now passed on to the system bus
- Cycle 7: DDR Read Data-1 is passed on to the system bus
- Cycle 8: DDR Read Data-2 is passed on to the system bus
- Cycle 9: DDR Read Data-3 is passed on to the system bus

The first access request from the system bus to ASYNC is issued immediately (same cycle). Subsequent requests are issued only when the first access request is completed. Two consecutive requests to ASYNC block the next access (any, including DDR access from that bus that initiated the accesses). However, accesses to DDR from other buses are not blocked.

DDR Arbitration

DDR Queue Manager

To optimize for the high throughput of the DDR interface, the EBIU implements three identical queue modules for each of the DEB buses. The queue managers perform the following functions:

- Enable peripherals to utilize higher throughput provided by DDR SDRAM
- Optimize requests to the DDR controller to achieve maximum utilization of the DDR memory bus
- Handle data coherency between the DEB and core buses

DDR Arbitration

The DDR arbiter handles requests from all four system interface buses (DEB0, DEB1, DEB2, and EAB) and prefetches requests from all the DEB queue blocks. The arbiter has a fixed priority as shown in the following:

1. Core TESTSET instruction (highest)
2. Forced write access (by DEB queue manager)
3. Urgent DMA access
4. Core access
5. Normal DMA read access through DEB queue manager
6. Normal DMA write access through DEB queue manager
7. Prefetch buffer access (lowest)

Note, there is a further programmable priority scheme for the three DEB buses when DMA wins arbitration (urgent or normal access). The arbitration priority between the DEB buses are determined by bits [10:8] of the DDR queue configuration register (EBIU_DDRQUE) as follows:

000:DEB0>DEB1>DEB2 (*default*)

001:DEB1>DEB0>DEB2

010:DEB2>DEB0>DEB1

[Table 6-2](#) summarizes the arbitration scheme, in DDR SDRAM memory interface.

Table 6-2. DDR Arbiter Priority Scheme

DEB_ARB_PRIORITY: 000 (0>1>2)	DEB_ARB_PRIORITY: 001 (1>0>2)	DEB_ARB_PRIORITY: 010 (2>0>1)
Core TESTSET	Core TESTSET	Core TESTSET
Forced DEB Writes DEB0 WRITE DEB1 WRITE DEB2 WRITE	Forced DEB Writes DEB1 WRITE DEB0 WRITE DEB2 WRITE	Forced DEB Writes DEB2 WRITE DEB0 WRITE DEB1 WRITE
Urgent DMA DEB0 READ DEB1 READ DEB2 READ DEB0 WRITE DEB1 WRITE DEB2 WRITE	Urgent DMA DEB1 READ DEB0 READ DEB2 READ DEB1 WRITE DEB0 WRITE DEB2 WRITE	Urgent DMA DEB2 READ DEB0 READ DEB1 READ DEB2 WRITE DEB0 WRITE DEB1 WRITE
Core READ/WRITE	Core READ/WRITE	Core READ/WRITE
Normal DMA READ DEB0 READ DEB1 READ DEB2 READ	Normal DMA READ DEB1 READ DEB0 READ DEB2 READ	Normal DMA READ DEB2 READ DEB0 READ DEB1 READ

DDR Arbitration

Table 6-2. DDR Arbiter Priority Scheme (Cont'd)

DEB_ARB_PRIORITY: 000 (0>1>2)	DEB_ARB_PRIORITY: 001 (1>0>2)	DEB_ARB_PRIORITY: 010 (2>0>1)
Normal DMA WRITE DEB0 READ DEB1 READ DEB2 READ	Normal DMA WRITE DEB1 READ DEB0 READ DEB2 READ	Normal DMA WRITE DEB2 READ DEB0 READ DEB1 READ
Prefetch Access DEB0 READ DEB1 READ DEB2 READ	Prefetch Access DEB1 READ DEB0 READ DEB2 READ	Prefetch Access DEB2 READ DEB0 READ DEB1 READ

The EBIU adds further control to the DDR arbitration by allowing a normal DMA access to be elevated to urgent DMA access by setting bits [14:12] in the DDR queue configuration register (EBIU_DDRQUE) as follows:

Bit[12] = 1 : DEB0 Normal DMA treated as Urgent
 0 : DEB0 Normal DMA treated as Normal (Default)


Bit[13] = 1 : DEB1 Normal DMA treated as Urgent
 0 : DEB1 Normal DMA treated as Normal (Default)

Bit[14] = 1 : DEB2 Normal DMA treated as Urgent
 0 : DEB2 Normal DMA treated as Normal (Default)

[Table 6-3](#) summarizes the arbitration scheme for Asynchronous memory interface.

Table 6-3. ASYNC Arbiter Priority Scheme

DEB_ARB_PRIORITY: 000 (0>1>2)	DEB_ARB_PRIORITY: 001 (1>0>2)	DEB_ARB_PRIORITY: 010 (2>0>1)
Core TESTSET	Core TESTSET	Core TESTSET
Urgent DMA DEB0 READ/WRITE DEB1 READ/WRITE DEB2 READ/WRITE	Urgent DMA DEB1 READ/WRITE DEB0 READ/WRITE DEB2 READ/WRITE	Urgent DMA DEB2 READ/WRITE DEB0 READ/WRITE DEB1 READ/WRITE
Core READ/WRITE	Core READ/WRITE	Core READ/WRITE
Normal DMA DEB0 READ/WRITE DEB1 READ/WRITE DEB2 READ/WRITE	Normal DMA DEB1 READ/WRITE DEB0 READ/WRITE DEB2 READ/WRITE	Normal DMA DEB2 READ/WRITE DEB0 READ/WRITE DEB1 READ/WRITE

 The priority schemes described in [Table 6-2](#) (for DDR) and [Table 6-3](#) (for ASYNC) are from the arbiters' perspective. The priority schemes are followed by the arbiters only when they are ready to arbitrate, not when the EBIU receives requests on the DEB or processor buses. For example, a DEB bus may indicate urgent during a request, but if the urgent signal goes away before the arbiter arbitrates, the DEB request is treated as a regular request. Also note, that the DEB queue logic blocks optimize the DEB bus requests (for example, line hit, prefetch during reads, packing during writes, and others). Because of these optimizations, the DEB bus requests may not show up at the arbiters immediately and they may be in a different order.

DDR SDRAM Controller

The DDR SDRAM controller (SDC) enables a transfer of data to and from synchronous DDR SDRAM with a maximum data rate of 532M bytes per second at a clock frequency of 133 MHz using both the edges of the clock. It supports a glueless interface with two external banks, controlled by the memory chip select pins ($\overline{\text{DCS1-0}}$), of standard DDR SDRAMs of 64M bit to 512M bit with configurations x4, x8, x16 as shown in the following tables, up to a maximum total capacity of 256M bytes of SDRAM per chip select. The interface includes timing options to support additional buffers between DDR SDRAM and the EBIU to handle capacitive loads of large memory arrays.

Features

The features of the DDR SDRAM controller (SDC) are:

- Supports industry-standard, double-data rate (DDR SDRAM) from 64M bit to 512M bit device sizes with a configuration of x4, x8, or x16
- Provides 16-bit data interface to DDR SDRAM
- Supports up to 256M bytes of DDR SDRAM with one external bank
- Supports up to two external banks
- Provides page hit detection to support multiple column accesses within the same row
- Provides eight internal row address registers to keep track of eight open rows (two chip select with four internal banks each)

- Supports fixed SDRAM burst length of two
- Provides programmable SDRAM access timing parameters
- Provides automatic refresh generation with programmable refresh intervals
- Supports self-refresh mode to reduce system power consumption
- Provides support for connection to mobile DDR

DDR SDRAM Memory Interface

This is a DDR SDRAM compliant interface. None of the signals in this interface is multiplexed with any other signals on the chip. This interface has a 2.5 V power supply to support the DDR specification. This interface can also be connected to mobile DDR devices at 1.8 V.

Table 6-4. DDR SDRAM Memory Interface

Name	Type	Description
DCLK1 / $\overline{\text{DCLK1}}$	O	Output clock signals to DDR SDRAM chips. Use as differential clock signals to DDR SDRAM.
DCLK2 / $\overline{\text{DCLK2}}$	O	Output clock signals to DDR SDRAM chips. Use as differential clock signals to DDR SDRAM. Same as DDR_CLK1.
DCKE	O	Clock enable
$\overline{\text{DCS1-0}}$	O	Clock enable output to the DDR SDRAM
DBA1-0	O	Chip select: One chip select for each of the two external banks
$\overline{\text{DCAS}}$	O	Column address select
$\overline{\text{DRAS}}$	O	Row address select
$\overline{\text{DWE}}$	O	Write enable

DDR SDRAM Memory Interface

Table 6-4. DDR SDRAM Memory Interface (Cont'd)

Name	Type	Description
DQS1-0	IO	Data Strobe: output with write data, input with read data. DQS is edge aligned with read data, but centered with write data. It is generated by the DDR controller during write access.
DQ15-0	IO	DDR data input and output. DDR SDRAM has twice the data rate.
DQM1-0	IO	Data mask for writes. DM turns the out buffers off for writes. For Write, DM specifies the bytes to be written. It is also used to mask a single Write during an access cycle of burst length = 2.
DA12-0	O	Memory address bits: Indicates row and column address and signals auto-precharge. When 64M bit and 128M bit SDRAM are used, only DDR_ADDR[11:0] are used as addresses and BA [1:0] are used as bank select. When 256M bit and 512M bit DDR SDRAM are used, DDR_ADDR [12:0] are used as address and BA [1:0] are used as bank select.

DDR SDRAM Programming Model

This section describes the programming model of the EBIU. This model is based on system memory-mapped registers (SMMRs), used to program the EBIU. This set of control registers is accessed across the peripheral access bus (PAB) of the extended core.

The control and status registers in the DDR controller include:

- Memory control register 0 (EBIU_DDRCTL0) Address 0xFFC0 0A20
- Memory control register 1 (EBIU_DDRCTL1) Address 0xFFC0 0A24
- Memory control register 2 (EBIU_DDRCTL2) Address 0xFFC0 0A28
- Memory control register 3 (EBIU_DDRCTL3) Address 0xFFC0 0A2C

- DDR queue manager configuration register (EBIU_DDRQUE) Address 0xFFC0 0A30
- Error address register (EBIU_ERRADD) Address 0xFFC0 0A34
- Error master register (EBIU_ERRMST) Address 0xFFC0 0A38
- Reset control register (EBIU_RSTCTL) Address 0xFFC0 0A3C

Access to the DDR controller registers ONLY can be made after releasing the DDR controller soft reset bit in the reset control register by writing a 1 in bit[0] in the register.

The user may write to the DDR control registers as long as the controller is not accessing memory devices. Otherwise, the controller responds to any writes to its registers after it finishes any ongoing memory accesses.

The DDR control registers contain sensitive timing parameters and settings for the DDR SDRAM. Carefully program these registers with values that are in the operating range of the DDR used.

Values in the reserved fields in these registers must be maintained according to the specification. Writing to reserved fields or writing any reserved values in register bits cause the DDR to function erroneously.

The user must not change prefetch length fields of this register during an ongoing transfer on DEB buses; otherwise unpredictable behavior may happen.

DDR Registers

This section provides descriptions of the EBIU's memory-mapped registers (MMRs) for DDR programming.

This section describes the following registers:

- “Memory Control Register 0 (EBIU_DDRCTL0)”
- “Memory Control Register 1 (EBIU_DDRCTL1)”
- “Memory Control Register 2 (EBIU_DDRCTL2)”
- “Memory Control Register 3 (EBIU_DDRCTL3), Regular DDR Devices”
- “Memory Control Register 3 (EBIU_DDRCTL3) Mobile DDR Devices”
- “Error Master Register (EBIU_ERRMST)”
- “Error Address Register (EBIU_ERRADD)”
- “Reset Control Register (EBIU_RSTCTL)”

Memory Control Register 0 (EBIU_DDRCTL0)

Memory Control Register 0 (EBIU_DDRCTL0)

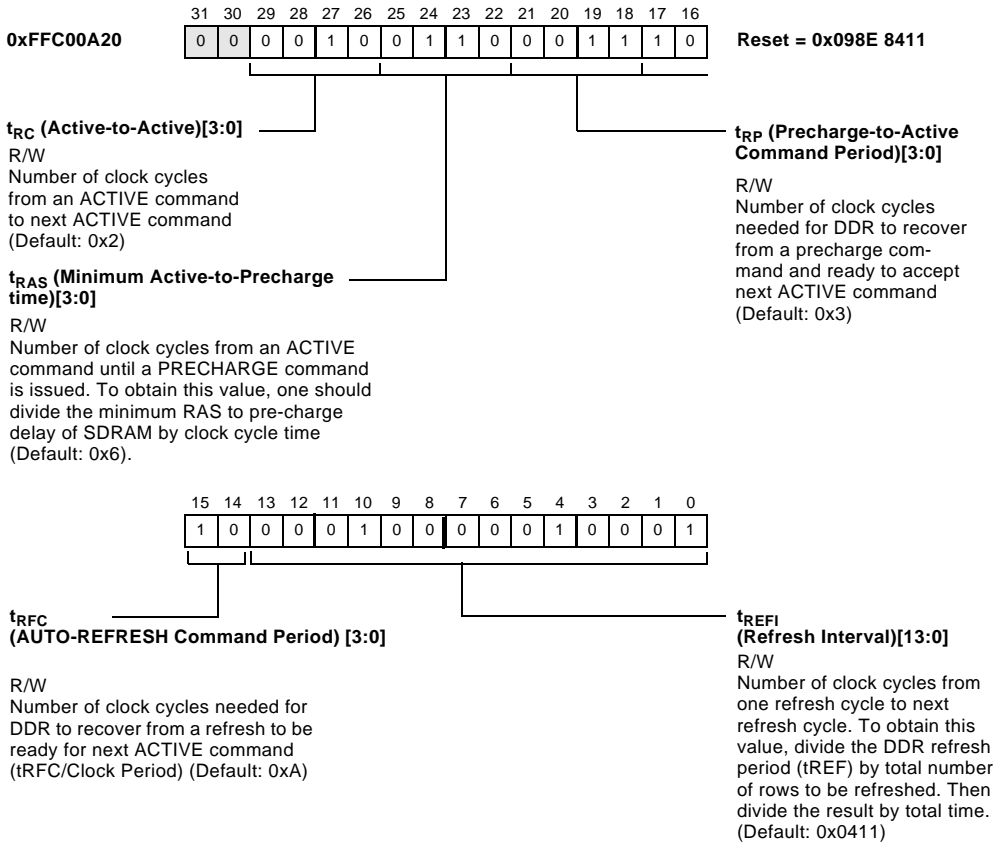


Figure 6-3. Memory Control Register 0 (EBIU_DDRCTL0)

DDR SDRAM Memory Interface

Memory Control Register 1 (EBIU_DDRCTL1)

Memory Control Register 1 (EBIU_DDRCTL1)

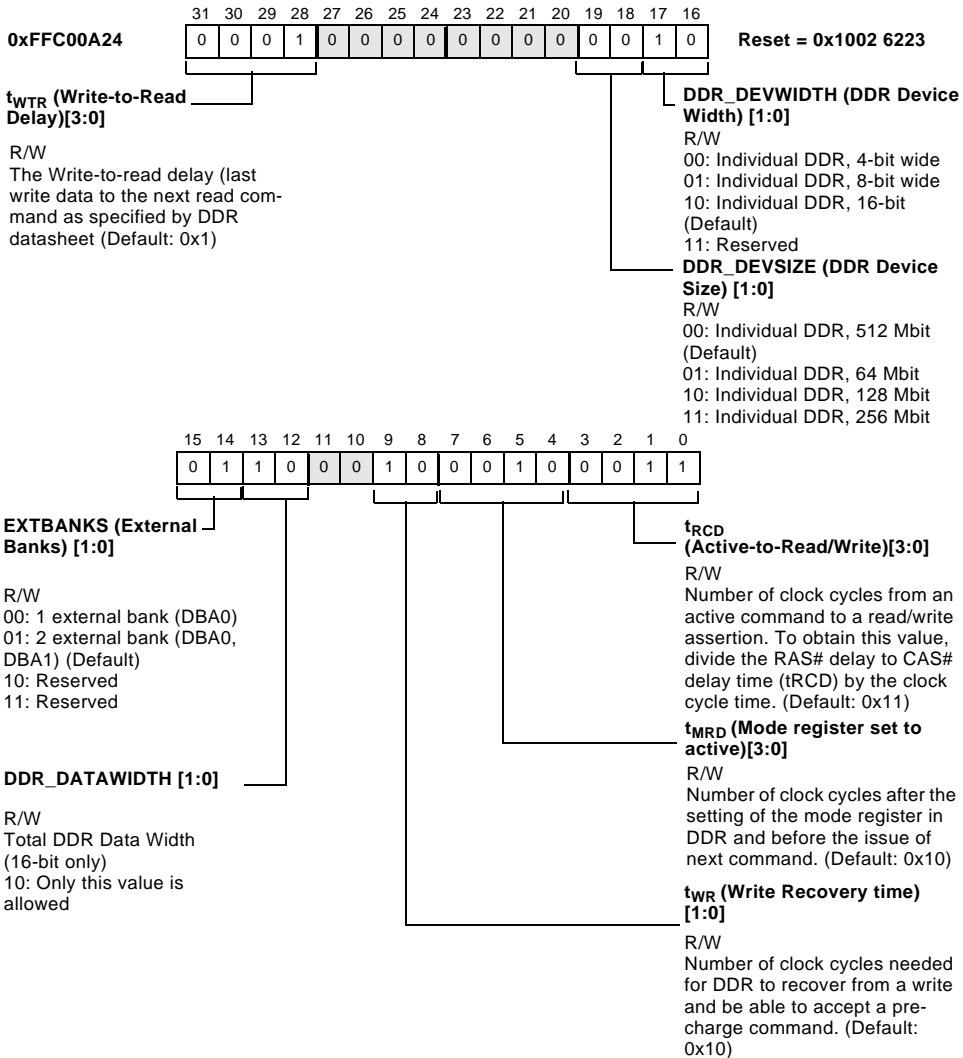


Figure 6-4. Memory Control Register 1 (EBIU_DDRCTL1)

Memory Control Register 2 (EBIU_DDRCTL2)

Memory Control Register 2 (EBIU_DDRCTL2)

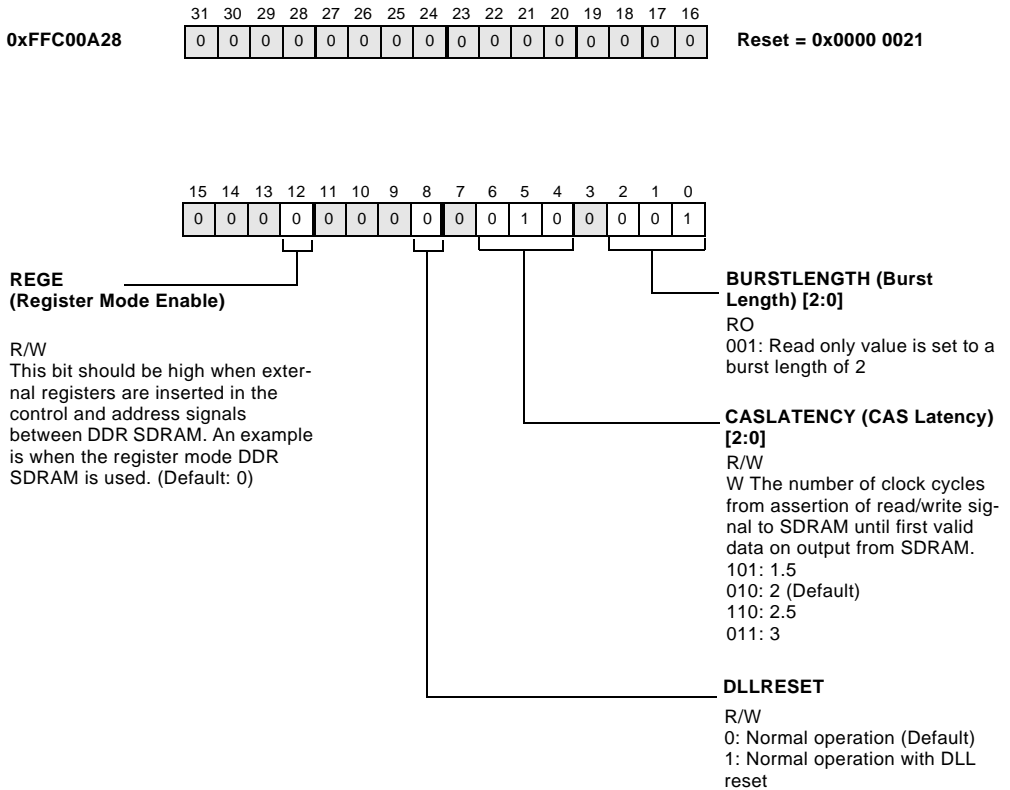


Figure 6-5. Memory Control Register 2 (EBIU_DDRCTL2)

DDR SDRAM Memory Interface

Memory Control Register 3 (EBIU_DDRCT3), Regular DDR Devices

Memory Control Register 3 (EBIU_DDRCTL3)

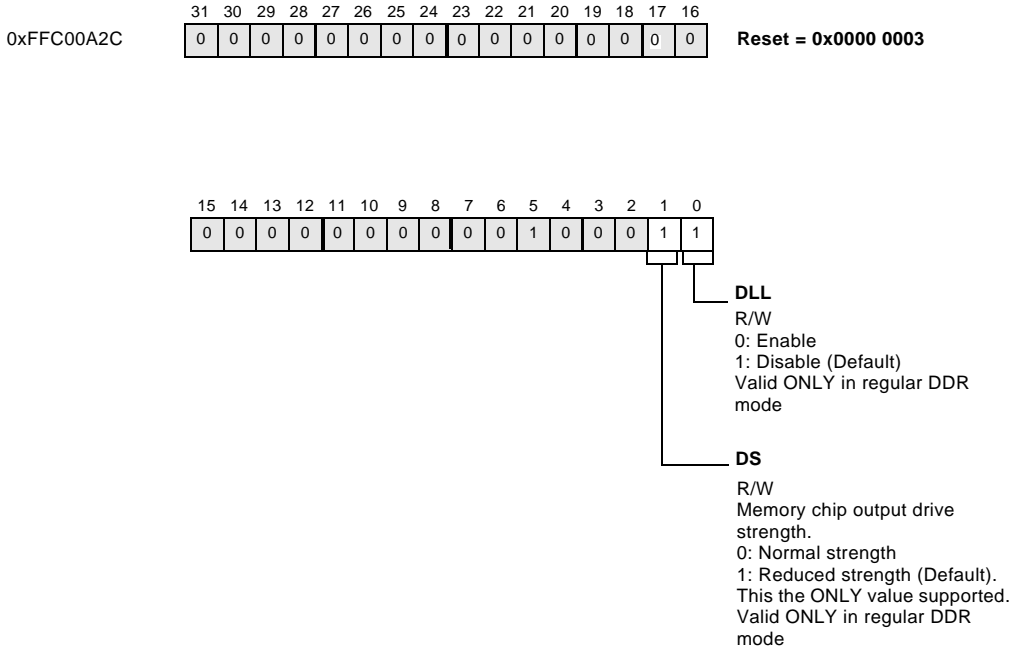


Figure 6-6. Memory Control Register 3 (EBIU_DDRCTL3), Regular DDR Devices

Memory Control Register 3 (EBIU_DDRCTL3) Mobile DDR Devices

Memory Control Register 3 (EBIU_DDRCTL3)

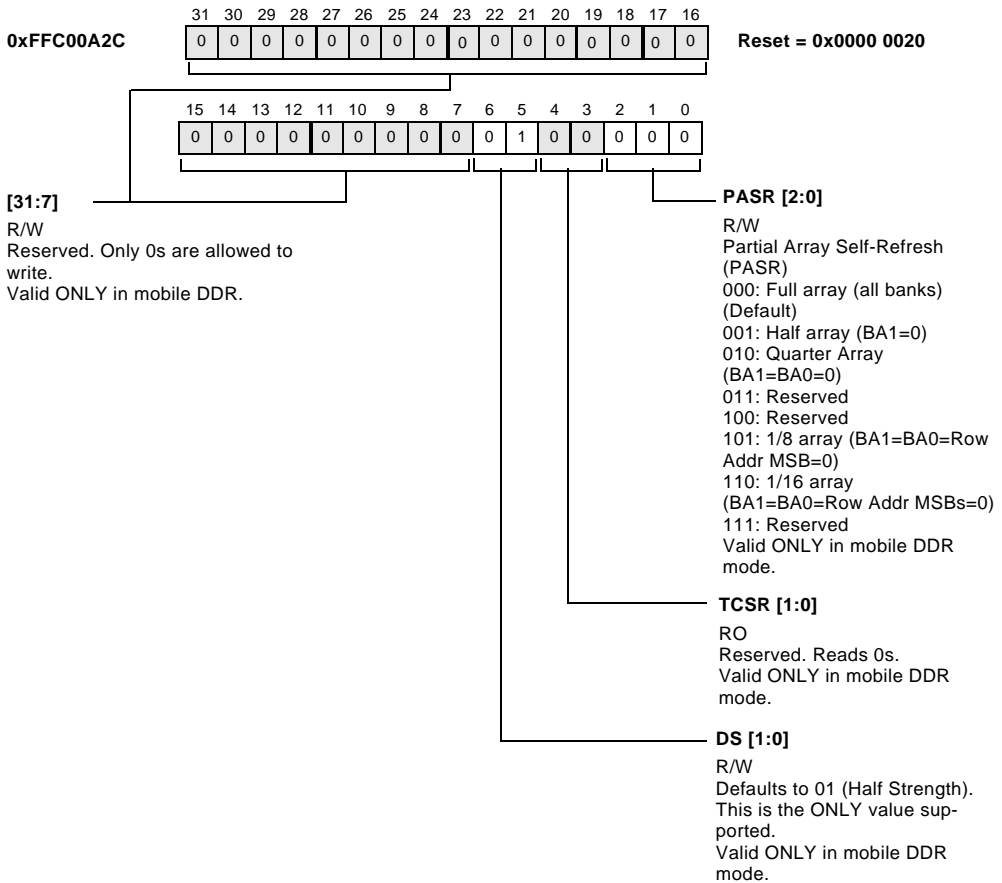


Figure 6-7. Memory Control Register 3 (EBIU_DDRCTL3) Mobile DDR Devices

DDR SDRAM Memory Interface

Queue Configuration Register (EBIU_DDRQUE)

Queue Configuration Register (EBIU_DDRQUE)

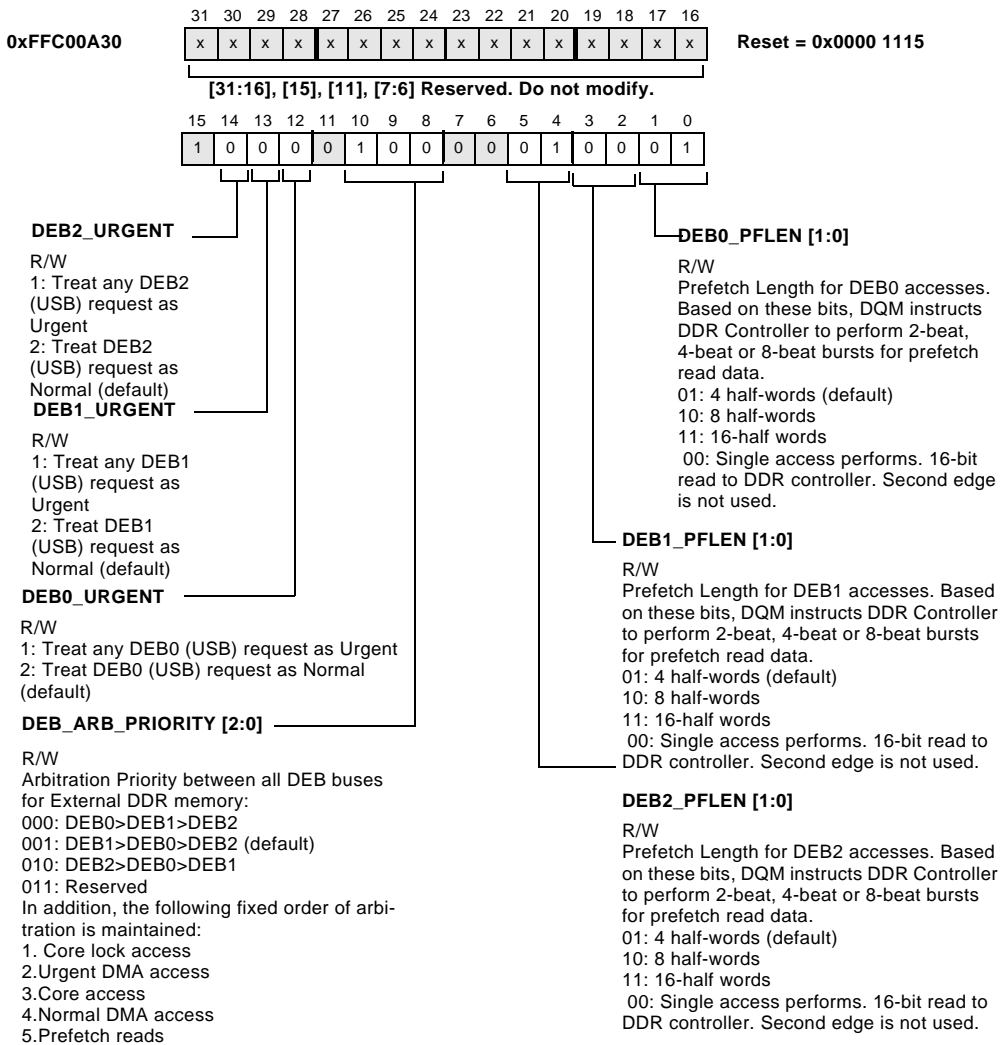
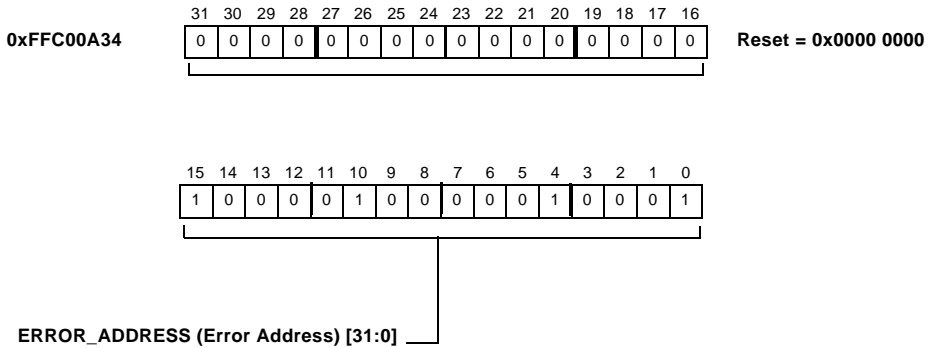


Figure 6-8. Queue Configuration Register (EBIU_DDRQUE)

Error Address Register (EBIU_ERRADD)

Error Address Register (EBIU_ERRADD)



RO

The error address to which any Bus Master (DEB0, DEB1, DEB2, Core) had accessed. This register captures the first error address by an individual bus. If two errors accesses happen by two buses, the address with the later bus will be captured.

Figure 6-9. Error Address Register (EBIU_ERRADD)

DDR SDRAM Memory Interface

Error Master Register (EBIU_ERRMST)

Error Master Register (EBIU_ERRMST)

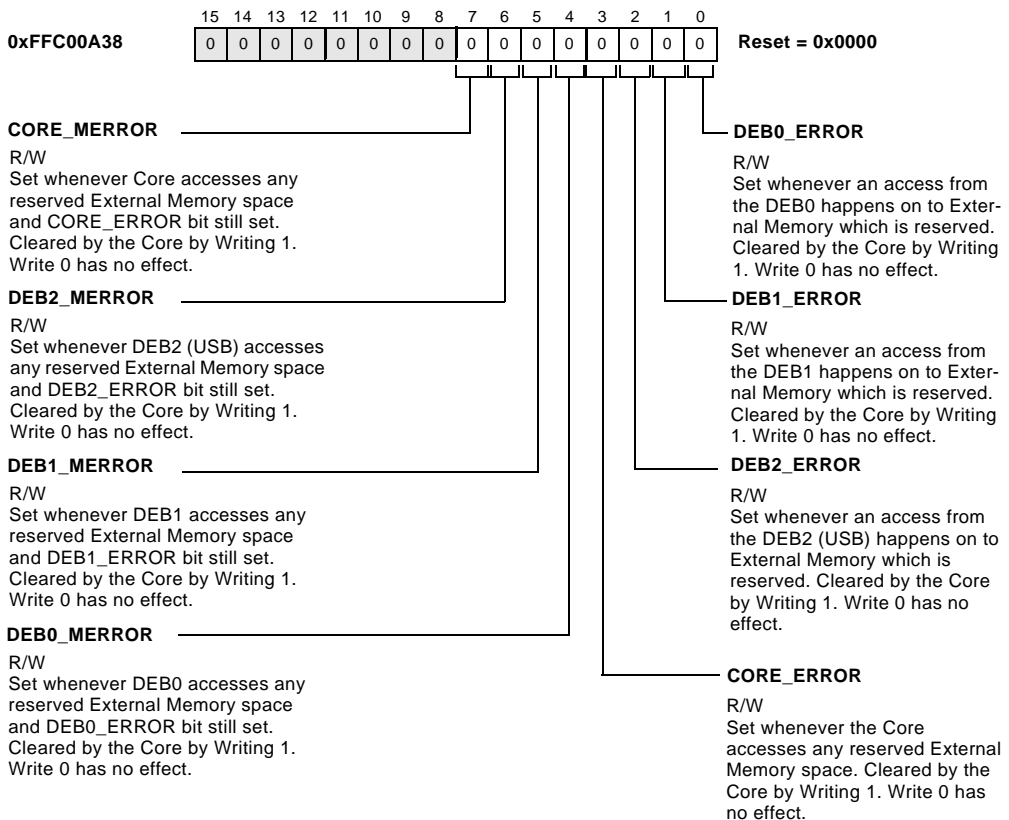


Figure 6-10. Error Master Register (EBIU_ERRMST)

Reset Control Register (EBIU_RSTCTL)

Reset Control Register (EBIU_RSTCTL)

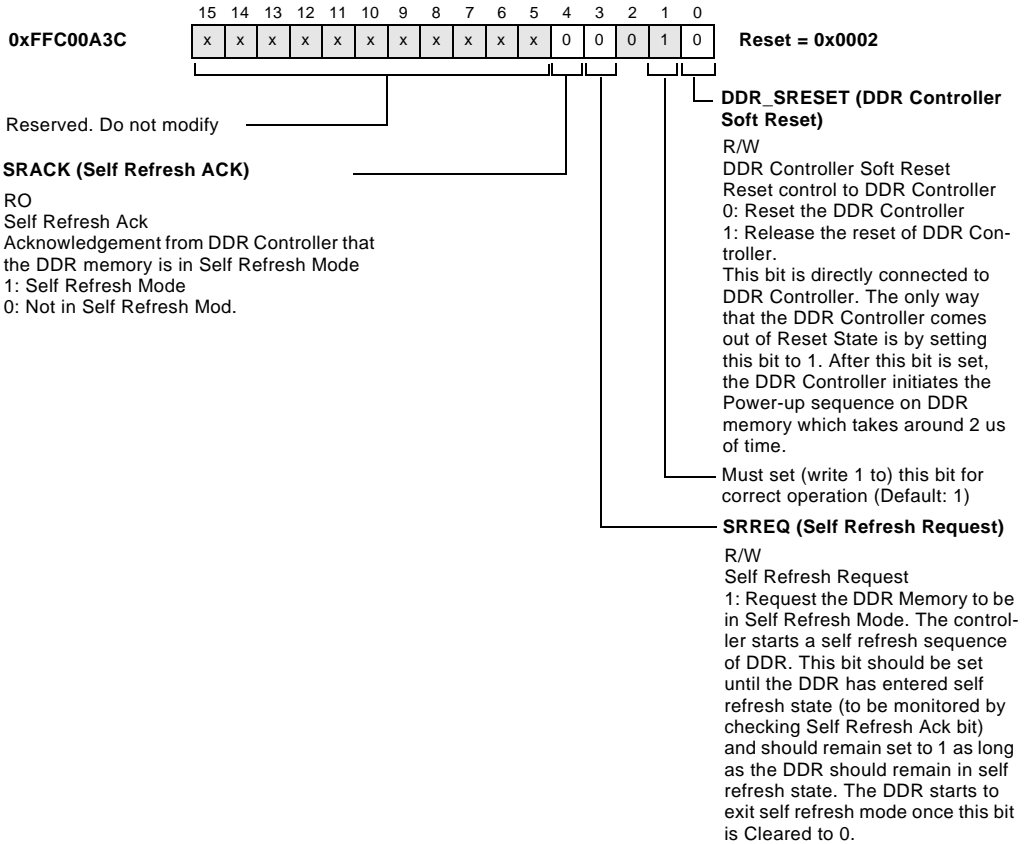


Figure 6-11. Reset Control Register 0 (EBIU_RSTCTL)

Mode of Operation - DDR

The DDR SDRAM controller performs the DDR SDRAM read and write accesses based on external SDRAM memory requests by the processor core

DDR SDRAM Memory Interface

EAB, DEB0, DEB1, and DEB2 buses.

The DDR SDRAM timing, such as row and column latency, precharge timing, and row access time are programmed to default values at system reset. They also can be programmed during run time if the application wishes to optimize the system performance. The internal counters in the DDR controller handle all the timing parameters.

Data between the DDR SDRAM controller and the DDR SDRAM device transfers at both the rising edge and falling edge of clock. The DDR SDRAM controller has the built-in data path to handle all data generation and sampling tasks.

Data Flow for 16-bit DDR SDRAMs

For read access, the DDR SDRAM drives 16 bits of data at both edges of the SDRAM clock. The DQS strobe is sampled by the DDR controller data path (synchronized with internal clock) and transferred to the DDR arbiter as a single 32-bit data. The DDR arbiter transfers the 32-bit data to the corresponding queues for which the read request command is accepted. The queue in turn transfers the same on to DMA buses or unpacks the 32-bit data word into two single half-words (16-bit) or 4 single bytes (8-bit), depending upon the DMA data width, before transferring them on to DMA buses. In the case of 32-bit wide DMA transfers, no unpacking is done.

In case of a core read request, the DDR arbiter transfers the 32-bit data to the core.

For write accesses, each DEB queue accepts byte/half-word/word requests from the corresponding DMA bus and packs into a 32-bit DDR SDRAM data word. A write request to the DDR arbiter is then made. The DDR arbiter then accepts a 32-bit write requests from DEB queues and the core bus, arbitrates based on arbitration priority and transfers one of the write request on to DDR controller. The DDR controller in turn writes as two 16-bit half-words on both edges of the clock (DQS strobe).

In case of write requests from the core, write commands are sent directly to the DDR arbiter without any packing.

The DDR SDRAM controller supports SDRAM devices of sizes of 64, 128, 256, 512 Mbits. For all device sizes it supports configurations of x4, x8 and x16 data width per SDRAM. The user can use multiple SDRAM devices to build a SDRAM data width of 16-bits. Both the device size and SDRAM data word size is programmable by user.

The DDR SDRAM controller supports an open page policy. Open page policy takes advantage of the fact that once a row is activated, multiple accesses can be made to the same row (page) without precharging the bank.

The pipeline feature of the DDR controller and the queuing feature of the queue manager block consecutive page hit write or consecutive page hit read to/from DDR without any idle cycles between accesses.

Definition of Standard DDR Terms

The following are definitions used in the rest of this chapter.

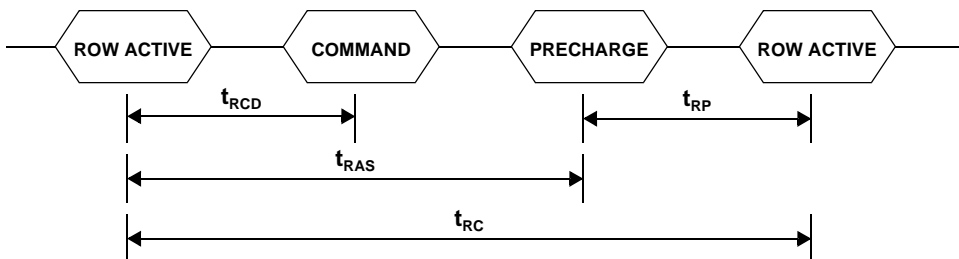


Figure 6-12. DDR Terms

Active command. The active command is used to open (or activate) a row in a particular bank for subsequent access. The value on the DBA1-0 pins selects DDR's internal bank, and address provided on DA12-0 pins selects

DDR SDRAM Memory Interface

the row. The open row is also referred to as the open page. This row (page) remains open for accesses until a precharge command is issued to that bank. In a particular bank, only one row can be open at any time. A precharge command must be issued before opening a different row in the same bank.

Precharge command. The precharge command is used to close (or deactivate) the open row in a particular bank or the open row in all internal banks. Once a bank is precharged, it is in an idle state and must be activated prior to any read or write commands being issued to the same bank.

Read command. Precharge not supported

Write command: Precharge not supported

Auto refresh command. The DDR data needs to be refreshed within a certain interval to prevent loss of data. The DDR controller automatically issues a auto refresh command to the DDR SDRAM device to refresh its memory. The DDR controller refreshes one row each time.

The refresh interval is programmable by the user in the control register. All control registers can be programmed during run time. The refresh interval field in the control register measures the interval for auto-refresh in terms of clock cycles. If the cycle time of the system clock is 10ns, the refresh interval value should be 780 to indicate 7.8 μ s refresh interval time.

The DDR SDRAM controller has an internal counter to count the refresh period. When the counter expires, the controller precharges all the banks and then issues an auto-refresh command to the SDRAM, if the SDRAM is in idle state. If the SDRAM is being accessed for read, write or other commands, the precharge and auto refresh commands are delayed until the current command is completed.

If there is a request for an access to DDR SDRAM while auto refresh is in progress, the same would be delay till auto refresh is completed.

Enter Self-Refresh Mode. The DDR SDRAM controller enters the self-refresh mode under user control to minimize power consumption. When the `SRREQ` bit is set (in the `EBIU_RSTCTL` register), it starts the self refresh sequence. This enables the SDRAM to continue to refresh its memory array while minimizing power consumption, resulting in no data loss. Once the `SRREQ` bit is set, it should not be cleared until DDR SDRAM enters a self refresh state, indicated by `SRACK=1` in the reset control register. The processor or DMA should not issue any further commands until the `SRACK` bit is set.

The DDR controller bring the DDR SDRAM to self-refresh mode by issuing self-refresh and de-asserts the `DCKE` signal. The `DCKE` signal is kept low until the DDR exits self-refresh mode.

Exit from Self-Refresh Mode. To exit from self-refresh mode, the `SRREQ` bit must be de-asserted by user. The controller asserts the `DCKE` signal and then issues an auto-refresh after waiting for 16 clock cycles. However, DDR SDRAM devices are required to wait for 200 clock cycles before process any read/write request. The DDR SDRAM Controller keeps the `SRACK` bit asserted high for 200 cycle after the `DCKE` is asserted. After the `SRACK` is cleared, SDRAM is operational again and user can issue normal SDRAM requests. So the processor should check for `SRACK` being cleared and then issue any commands.

Mode Register Set. The mode register is the DDR's internal configuration register containing user-defined parameters. The mode register set command is issued by the DDR controller automatically during power on initialization and when the user writes to `EBIU_DDRCTL2`.

DDR SDRAM Memory Interface

Extended Mode Register Set. The extended mode register set command is issued by the DDR controller automatically during power on initialization and when the user writes to EBIU_DDRCTL3. Extended mode register set and mode register set differ by encoding of the DBA1–0 signals.

Table 6-5. DDR SDRAM Commands

CS#	RAS#	CAS#	WE#	BA[1:0]	Commands
L	L	L	L	00	Mode register set
L	L	L	L	01	Extended mode register set
H	X	X	X	X	Command inhibit (NOP)
L	L	H	H	X	Active
L	H	L	H	X	Read
L	H	L	L	X	Write
L	L	H	L	X	Precharge
L	L	L	H	X	Refresh
L	L	L	L	X	Mode register set/extended mode register set
L	H	H	L	X	Burst terminate
-	-	-	-	L	Write enable/output enable
-	-	-	-	H	Write inhibit/output high -Z

Burst Length. The burst length determines the number of words the DDR stores or delivers after detecting a single write or read command, respectively. The burst length is programmed in the SDRAM mode register during the power-up sequence. The DDR controller, for the ADSP-BF54x processor, only supports burst length=2 mode.

Burst Stop Command. The burst stop command is one of several ways to terminate a burst read or write operation. Since the SDRAM burst length is always programmed to be 2, the DDR controller does not do any burst stop command.

Burst Type. The burst type determines the access order in which the DDR delivers burst data after detecting a read command or stores burst data after detecting a write command. The burst type is programmed in the DDR mode register during the power-up sequence. Burst type can be sequential or interleaved. Since the DDR controller only supports burst length of 2, the burst type does not matter. The ADSP-BF54x processor's DDR controller always sets the burst type to sequential-accesses-only during the SDRAM power-up sequence.

CAS Latency (also t_{AA} , t_{CAC} , t_{CL}). The column address strobe (\overline{DCAS}) 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 \overline{DCAS} latency is programmed in the SDRAM mode register during the power-up sequence. The speed grade of the device and the application's clock frequency determine the value of the \overline{DCAS} latency. The DDR controller supports \overline{DCAS} latency of 1.5, 2, 2.5, and 3 clocks.

CBR (CAS before RAS) Auto-Refresh. When the DDR controller refresh counter times out, it precharges 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 DDR internal banks are precharged.

DQM Data I/O Mask Function. The $DQM1-0$ pins provide a byte masking capability on 8-bit writes to DDR. The $DQM1-0$ pins are not used to mask data on read cycles.

Internal Bank. In a DDR, there are several internal memory banks. These banks are selected by the bank address ($DBA1-0$) pins.

Page Hit Detection. The DDR controller stores the row address in the row address register every time it activates a bank. Internally the DDR controller has four row address registers, one for each bank. Once a bank is activated for read or write, the bank remains active. When a new access request arrives to the DDR SDRAM controller, it automatically checks

DDR SDRAM Memory Interface

the internal row address register. If the new access is for the same row (page hit), the DDR SDRAM controller skips the active command and directly issues the read/write command to access the DDR.

Maximum Bank Active Time. Each DDR bank can remain in an active state up to hundreds of microseconds, but it must be precharged again before the maximum active to precharge time is exceeded. The DDR SDRAM controller assures that each bank does not exceed the maximum active-to-precharge time by use of refresh interval. Since the refresh period is smaller than maximum active-to-precharge time in SDRAMs and all banks must be idle before a refresh can be issued, no bank will remain in active state for more than the active-to-pre-charge time. For each refresh issued, the DDR SDRAM controller checks that all banks are idle. If any bank is active, the controller issues an all bank precharge command to DDR before the refresh command.

The user must make sure that the refresh cycle that is programmed in `EBIU_DDRCTL0` is smaller than the active to precharge time.

Page Miss Access. When a DDR SDRAM access generates a page miss that the bank is precharged (deactivated), the DDR controller starts the access with the ACTIVE command. If the bank is active but the row address is a mismatch, the DDR controller first issues a precharge command. After the precharge-to-active delay, the DDR controller issues the active command and then read or write command to access the memory. If the bank is already precharged, the precharge command is skipped.

Register Mode DDR Support. The DDR SDRAM controller supports DDR SDRAM systems with and without external registers for address and control signals. Buffered mode is functionally identical to using single discrete SDRAM devices. The control and address signals are buffered on the board to reduce loading to the SDRAM controller. The `REGE` bit must be set to 0 to support discrete and buffered mode.

Register mode is designed for systems that have external registers for each control and address signal between the DDR controller and the DDR SDRAM. The `REGE` bit is set to 1 to enable the register mode. When register mode is enabled, the latency of all accesses is increased by one system clock cycle.

DDR SDRAM System Organization

DDR devices are available with 4-, 8-, and 16-bit data width. To build a memory system with 16-bit data, multiple x4 or x8 DDR SDRAM devices can be connected in parallel to provide total data bits. Different data word sizes do not affect the address bit used to access the SDRAM. The word size on the interface between the DDR SDRAM controller and the DDR queue block is always double the width of the data path to the DDR because the DDR transfers two bits of data per pin per clock cycle.

All the address and control signals, with the exception of `DQM1-0`, are common to all SDRAM chips. The `DQM1-0` signal must match with the data bits with which they are associated.

DDR SDRAM Memory Interface

Figure 6-13 shows a DDR system of 16-bit data word made by using 512M bit (64M bytes) SDRAM devices with x8 configuration, producing 128M bytes per external memory bank.

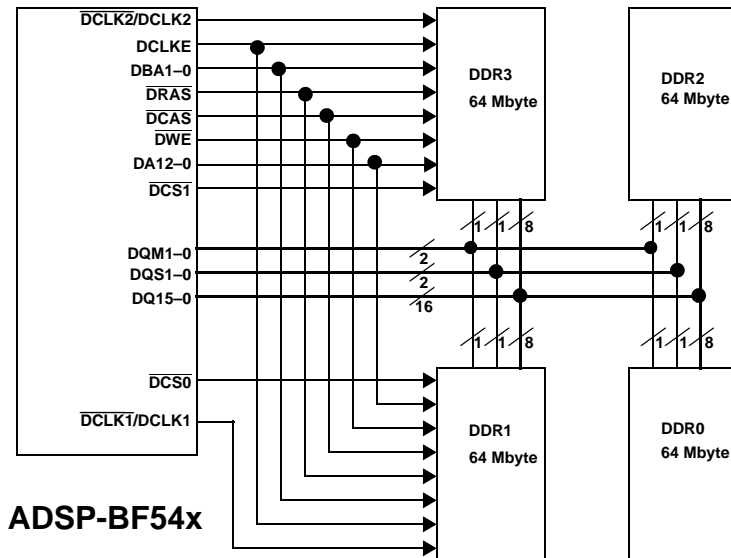


Figure 6-13. 16-bit Data Bus DDR System

DDR SDRAM Configurations Supported

The ADSP-BF54x DDR SDRAM controller supports different sizes of SDRAM chips from 64 Mbit to 512 Mbit. The following tables lists all the supported sizes.

Table 6-6. Using 64 Mbit (8M bytes) SDRAM Chips

SDRAM Data Width	Individual SDRAM Width	Total SDRAM Needed	Total Size per External Bank	Row Address Bits	Column Address Bits	Bank Address Bits	External Chip Selects
16	x4	4	32M bytes	22:11	10:1	24:23	26:25
16	x8	2	16M bytes	21:10	9:1	23:22	25:24
16	x16	1	8M bytes	20:9	8:1	22:21	24:23

Table 6-7. Using 128 Mbit (16M bytes) SDRAM Chips

SDRAM Data Width	Individual SDRAM Width	Total SDRAM Needed	Total Size per External Bank	Row Address Bits	Column Address Bits	Bank Address Bits	External Chip Selects
16	x4	4	64M bytes	23:12	11 :1	25:24	27:26
16	x8	2	32M bytes	22:11	10:1	24:23	26:25
16	x16	1	16M bytes	21:10	9:1	23:22	25:24

DDR SDRAM Memory Interface

Table 6-8. Using 256 Mbit (32M bytes) SDRAM Chips

SDRAM Data Width	Individual SDRAM Width	Total SDRAM Needed	Total Size per External bank	Row Address Bits	Column Address Bits	Bank Address Bits	External Chip Selects
16	x4	4	128M bytes	24:12	11:1	26:25	28:27
16	x8	2	64 M bytes	23:11	10:1	25:24	27:26
16	x16	1	32 M bytes	22:10	9:1	24:23	26:25

Table 6-9. Using 512 Mbit (64 M bytes) SDRAM Chips

SDRAM Data Width	Individual SDRAM Width	Total SDRAM Needed	Total Size per External bank	Row Address Bits	Column Address Bits	Bank Address Bits	External Chip Selects
16	x4	4	256 M bytes	25:13	12:1	27:26	29:28
16	x8	2	128 M bytes	24:12	11:1	26:25	28:27
16	x16	1	64 M bytes	23:11	10:1	25:24	27:26

DDR Timing Parameter Definitions

tRAS: ACTIVE-to-PRECHARGE Command Delay. Required delay between issuing an ACTIVE command and issuing a PRECHARGE command and between the SELF-REFRESH command and the exit from SELF-REFRESH.

tRP: PRECHARGE Command Period. Required delay between issuing a PRECHARGE command and issuing an Activate command, between a Precharge command and issuing an Auto-Refresh command, and between a Precharge command and issuing a Self-Refresh command.

tRC: ACTIVE_A-to-ACTIVE_A Delay. Required delay between issuing successive Activate commands to the same SDRAM internal bank. User must ensure that $t_{RAS} + t_{RP} \geq t_{RC}$.

tRCD: ACTIVE-to-READ/WRITE Delay. Required delay between an ACTIVE command and the start of the first READ or WRITE command

tRFC: AUTO-REFRESH Command Period. Required delay between issuing an AUTO-REFRESH command and an ACTIVE command.

tREFI: Average Refresh Interval. Number of cycles from one REFRESH command to next REFRESH command.

tWTR: WRITE-to-READ Delay. Number of cycles between last WRITE data and next READ command.

tWR: Write Recovery Time. The clock cycles needed for the SDRAM to recover from a WRITE command and be able to accept a PRECHARGE command.

tMRD: LOAD MODE REGISTER Command Cycle Time. The number of clock cycles after setting of the Mode Register in the DDR and before issue of next command.

DDR Metrics Control Registers

The EBIU provides a set of registers and counters to monitor performance and activities on the DDR SDRAM interface and in the DDR arbiter for accesses to DDR SDRAM memory. There are 23 counters and two control registers, each 32-bits wide, for this purpose.

The following sections describe the DDR metrics control registers:

- “DDR Metrics Counter Enable (EBIU_DDRMCEN) Register” on page 6-42
- “DDR Metrics Counter Clear (EBIU_DDRMCCL) Register” on page 6-45
- “DDR READ Access Count (EBIU_DDRBRCx) Registers” on page 6-48
- “DDR WRITE Access Count (EBIU_DDRBWCx) Registers” on page 6-49
- “DDR Page ACTIVATE Count (EBIU_DDRACCT) Register” on page 6-49
- “DDR TURN AROUND Count (EBIU_DDRTACT) Register” on page 6-50
- “DDR AUTO-REFRESH Count (EBIU_DDRARCT) Register” on page 6-50
- “DDR Grant Count (EBIU_DDRGCx) Registers” on page 6-50

DDR Metrics Counter Enable (EBIU_DDRMCEN) Register

This 32-bit SMMR independently controls different DDR metrics counters. Each bit in this register (except bit[25:24]) enables and disables the corresponding counter. When a bit is set to 1, the corresponding counter starts. When a bit is 0, the corresponding counter stops counting

but is not cleared. The corresponding bit in the DDR metrics counter clear (EBIU_DDRMCL) register must be set to clear the counter. Refer to [Table 6-10](#) and [Table 6-11](#).

Table 6-10. DDR Metrics Counter Enable Register

Address	Register Name	Size	Reset Value
0xFFC00AC0	DDR Metrics Counter Enable Register	32	0x00000000

Table 6-11. DDR Metrics Counter Enable Register Bits

Name	Offset	Access	Description
Reserved	31:30	RO	Reads 0
GCCONTROL DDR Grant Count Control	25:24	R/W	Specifies 4 different schemes for DDR Grants Count (see Table 6-14 on page 6-52): 00: Core, DEB0, DEB1, DEB2,(Default) 01: Core, DEB0WR, DEB0RD, DEB0PF 10: Core, DEB1WR, DEB1RD, DEB1PF 11: Core, DEB2WR, DEB2RD, DEB2PF
GC3ENABLE DDR Grant Count Register 3 Enable	23	R/W	1: Enable Grant Count Register 3 0: Disable Grant Count Register 3(Default)
GC2ENABLE DDR Grant Count Register 2 Enable	22	R/W	1: Enable Grant Count Register 2 0: Disable Grant Count Register2(Default)
GC1ENABLE DDR Grant Count Register 1 Enable	21	R/W	1: Enable Grant Count Register 1 0: Disable Grant Count Register1(Default)
GC0ENABLE DDR Grant Count Register 0 Enable	20	R/W	1: Enable Grant Count Register 0 0: Disable Grant Count Register01(Default)
Reserved	19	RO	Reads 0

DDR SDRAM Memory Interface

Table 6-11. DDR Metrics Counter Enable Register Bits (Cont'd)

ARCENABLE Total DDR Auto-Refresh Count Enable	18	R/W	1: Enable Auto-Refresh Count 0: Disable Auto-Refresh Count (Default)
RWTCENABLE Total DDR R/W Turn Around Count Enable	17	R/W	1: Enable Turn Around Count 0: Disable Turn Around Count (Default)
ROWACTCENABLE Total DDR Row ACTIVATE Count Enable	16	R/W	1: Enable Row Activate Count 0: Disable Row Activate Count (Default)
B7RCENABLE Bank7 Read Count Enable	15	R/W	1: Enable Read Count to Bank7 0: Disable Read Count to Bank7(Default)
B6RCENABLE Bank6 Read Count Enable	14	R/W	1: Enable Read Count to Bank6 0: Disable Read Count to Bank6(Default)
B5RCENABLE Bank5 Read Count Enable	13	R/W	1: Enable Read Count to Bank5 0: Disable Read Count to Bank5(Default)
B4RCENABLE Bank4 Read Count Enable	12	R/W	1: Enable Read Count to Bank4 0: Disable Read Count to Bank4(Default)
B3RCENABLE Bank3 Read Count Enable	11	R/W	1: Enable Read Count to Bank3 0: Disable Read Count to Bank3(Default)
B2RCENABLE Bank2 Read Count Enable	10	R/W	1: Enable Read Count to Bank2 0: Disable Read Count to Bank2(Default)
B1RCENABLE Bank1 Read Count Enable	9	R/W	1: Enable Read Count to Bank1 0: Disable Read Count to Bank1(Default)
B0RCENABLE Bank0 Read Count Enable	8	R/W	1: Enable Read Count to Bank0 0: Disable Read Count to Bank0(Default)
B7WCENABLE Bank7 WRite Count Enable	7	R/W	1: Enable Write Count to Bank7 0: Disable Write Count to Bank7(Default)
B6WCENABLE Bank6 WRite Count Enable	6	R/W	1: Enable Write Count to Bank6 0: Disable Write Count to Bank6(Default)
B5WCENABLE Bank5 WRite Count Enable	5	R/W	1: Enable Write Count to Bank5 0: Disable Write Count to Bank5(Default)

Table 6-11. DDR Metrics Counter Enable Register Bits (Cont'd)

B4WCENABLE Bank4 WRite Count Enable	4	R/W	1: Enable Write Count to Bank4 0: Disable Write Count to Bank4(Default)
B3WCENABLE Bank3 WRite Count Enable	3	R/W	1: Enable Write Count to Bank3 0: Disable Write Count to Bank3(Default)
B2WCENABLE Bank2 WRite Count Enable	2	R/W	1: Enable Write Count to Bank2 0: Disable Write Count to Bank2(Default)
B1WCENABLE Bank1 WRite Count Enable	1	R/W	1: Enable Write Count to Bank1 0: Disable Write Count to Bank1(Default)
B0WCENABLE Bank0 WRite Count Enable	0	R/W	1: Enable Write Count to Bank0 0: Disable Write Count to Bank0(Default)

DDR Metrics Counter Clear (EBIU_DDRMCCL) Register

This 32-bit SMMR controls independent clearing of DDR metrics counters. Each bit in this register, when set to 1, clears the corresponding counter. Writing 0 in a bit position has no affect on the corresponding counter. This register is used to clear the corresponding counter(s) before starting them. Refer to [Table 6-12](#) and [Table 6-13](#).

Table 6-12. DDR Metrics Counter Clear Register

Address	Register Name	Size	Reset Value
0xFFC00AC4	DDR Metrics Counter Clear Register	32	0x00000000

Table 6-13. DDR Metrics Counter Clear Register Bits

Name	Offset	Access	Description
Reserved	31:24	RO	Reads 0s

DDR SDRAM Memory Interface

Table 6-13. DDR Metrics Counter Clear Register Bits (Cont'd)

CG3COUNT Clear DDR Grant Count Register 3	23	R/W	1: Clear Grant Count Register 3 0: Do not Clear (Default)
CG2COUNT Clear DDR Grant Count Register 2	22	R/W	1: Clear Grant Count Register 2 0: Do not Clear (Default)
CG1COUNT Clear DDR Grant Count Register 1	21	R/W	1: Clear Grant Count Register 1 0: Do not Clear (Default)
CG0COUNT Clear DDR Grant Count Register 0	20	R/W	1: Clear Grant Count Register 0 0: Do not Clear (Default)
Reserved	19	RO	Reads 0
CARCOUNT Clear Total DDR Auto-Refresh Count	18	R/W	1: Clear Auto-Refresh Count 0: Do not Clear (Default)
CRWTACOUNT Clear Total DDR R/W Turn Around Count	17	R/W	1: Clear Turn Around Count 0: (Default)
CRACOUNT Clear Total DDR Row ACTIVATE Count	16	R/W	1: Clear Row Activate Count 0: Do not Clear (Default)
CB7RCOUNT Clear Bank7 Read Count	15	R/W	1: Clear Read Count to Bank7 0: Do not Clear (Default)
CB6RCOUNT Clear Bank6 Read Count	14	R/W	1: Clear Read Count to Bank6 0: Do not Clear (Default)
CB5RCOUNT Clear Bank5 Read Count	13	R/W	1: Clear Read Count to Bank5 0: Do not Clear (Default)
CB4RCOUNT Clear Bank4 Read Count	12	R/W	1: Clear Read Count to Bank4 0: Do not Clear (Default)
CB3RCOUNT Clear Bank3 Read Count	11	R/W	1: Clear Read Count to Bank3 0: Do not Clear (Default)

Table 6-13. DDR Metrics Counter Clear Register Bits (Cont'd)

CB2RCOUNT Clear Bank2 Read Count	10	R/W	1: Clear Read Count to Bank2 0: Do not Clear (Default)
CB1RCOUNT Clear Bank1 Read Count	9	R/W	1: Clear Read Count to Bank1 0: Do not Clear (Default)
CB0RCOUNT Clear Bank0 Read Count	8	R/W	1: Clear Read Count to Bank0 0: Do not Clear (Default)
CB7WCOUNT Clear Bank7 Write Count	7	R/W	1: Clear Write Count to Bank7 0: Do not Clear (Default)
CB6WCOUNT Clear Bank6 Write Count	6	R/W	1: Clear Write Count to Bank6 0: Do not Clear (Default)
CB5WCOUNT Clear Bank5 Write Count	5	R/W	1: Clear Write Count to Bank5 0: Do not Clear (Default)
CB4WCOUNT Clear Bank4 Write Count	4	R/W	1: Clear Write Count to Bank4 0: Do not Clear (Default)
CB3WCOUNT Clear Bank3 Write Count	3	R/W	1: Clear Write Count to Bank3 0: Do not Clear (Default)
CB2WCOUNT Clear Bank2 Write Count	2	R/W	1: Clear Write Count to Bank2 0: (Default)
CB1WCOUNT Clear Bank1 Write Count	1	R/W	1: Clear Write Count to Bank1 0: Do not Clear (Default)
CB0WCOUNT Clear Bank0 Write Count	0	R/W	1: Clear Write Count to Bank0 0: Do not Clear (Default)

DDR SDRAM Memory Interface

DDR READ Access Count (EBIU_DDRBRCx) Registers

Each of the following eight registers counts read accesses to the corresponding DDR SDRAM bank, when enabled. Bank4 through Bank7 imply banks in the second external memory bank.

- DDR Bank0 Read Count (EBIU_DDRBRC0) Register
(Address: 0xFFC0_0A60)
- DDR Bank1 Read Count (EBIU_DDRBRC1) Register
(Address: 0xFFC0_0A64)
- DDR Bank2 Read Count (EBIU_DDRBRC2) Register
(Address: 0xFFC0_0A68)
- DDR Bank3 Read Count (EBIU_DDRBRC3) Register
(Address: 0xFFC0_0A6C)
- DDR Bank4 Read Count (EBIU_DDRBRC4) Register
(Address: 0xFFC0_0A70)
- DDR Bank5 Read Count (EBIU_DDRBRC5) Register
(Address: 0xFFC0_0A74)
- DDR Bank6 Read Count (EBIU_DDRBRC6) Register
(Address: 0xFFC0_0A78)
- DDR Bank7 Read Count (EBIU_DDRBRC7) Register
(Address: 0xFFC0_0A7C)

DDR WRITE Access Count (EBIU_DDRBWCx) Registers

Each of the following eight registers counts write accesses to the corresponding DDR SDRAM bank, when enabled. Bank4 through Bank7 imply banks in the second external memory bank.

- DDR Bank0 Write Count Register (EBIU_DDRBWC0)
(Address: 0xFFC0_0A80)
- DDR Bank1 Write Count Register (EBIU_DDRBWC1)
(Address: 0xFFC0_0A84)
- DDR Bank2 Write Count Register (EBIU_DDRBWC2)
(Address: 0xFFC0_0A88)
- DDR Bank3 Write Count Register (EBIU_DDRBWC3)
(Address: 0xFFC0_0A8C)
- DDR Bank4 Write Count Register (EBIU_DDRBWC4)
(Address: 0xFFC0_0A90)
- DDR Bank5 Write Count Register (EBIU_DDRBWC5)
(Address: 0xFFC0_0A94)
- DDR Bank6 Write Count Register (EBIU_DDRBWC6)
(Address: 0xFFC0_0A98)
- DDR Bank7 Write Count Register (EBIU_DDRBWC7)
(Address: 0xFFC0_0A9C)

DDR Page ACTIVATE Count (EBIU_DDRACT) Register

(Address: 0xFFC0_0AA0) This register counts total number of times page ACTIVATE command was issued to the DDR SDRAM, for all banks.

DDR SDRAM Memory Interface

DDR TURN AROUND Count (EBIU_DDRTACT) Register

(Address: 0xFFC0_0AA8) This register counts total number of times there was a turn around between READ and WRITE or between WRITE and READ commands, for all banks.

DDR AUTO-REFRESH Count (EBIU_DDRARCT) Register

(Address: 0xFFC0_0AAC) This register counts total number of times AUTO-REFRESH command was issued, for all banks.

DDR Grant Count (EBIU_DDRGCx) Registers

There are four DDR grant count registers. These counters may be used to monitor how the four requesters (for example, EAB, DEB0, DEB1, DEB2) are granted access to the DDR memory:

- **DDR Grant Count Register 0 (EBIU_DDRGC0)**
(Address: 0xFFC0_0AB0) This register counts, when enabled, total number of times the EAB was granted access to DDR SDRAM, if the DDR grant control field of the DDR metrics counter enable register (bit[25:24]) is set to 0.
- **DDR Grant Count Register 1 (EBIU_DDRGC1)**
(Address: 0xFFC0_0AB4) This register, when enabled, counts total number of times the DEB0 was granted access to DDR SDRAM, if the DDR grant control field of the DDR metrics counter enable register (bit[25:24]) is set to 0.

- **DDR Grant Count Register 2** (`EBIU_DDRGC2`)
(Address: `0xFFC0_0AB8`) This register counts, when enabled, total number of times the DEB1 was granted access to DDR SDRAM, if the DDR grant control field of the DDR metrics counter enable register (bit[25:24]) is set to 0.
- **DDR Grant Count Register 3** (`EBIU_DDRGC3`)
(Address: `0xFFC0_0ABC`) This register counts, when enabled, total number of times the DEB2 was granted access to DDR SDRAM, if the DDR grant control field of the DDR metrics counter enable register (bit[25:24]) is set to 0.

More Grant Counter Options

The grant count registers can be configured to record grants in different ways, depending upon the DDR grant control field of the DDR metrics counter enable register (bit[25:24]). [Table 6-14](#) enumerates different ways user can configure these counters.

DDR Grant Count Control

The DDR grant count control field in the `EBIU_DDRMCEN` register (bit[25:24]) helps monitor arbitration activities inside the EBIU's arbiter.

- When this field is set to 0, the DDR grant count register 0, 1, 2, and 3 count the number of grants given to EAB, DEB0, DEB1, and DEB2 buses respectively for access requests to DDR SDRAM.
- When this field is set to 1, DDR grant count register 1, 2, and 3 count the total number of grants given to DEB0, number of grants given to DEB0 write requests, number of grants given to DEB0 read requests and number of grants given to DEB0 prefetch read requests, respectively. (grant count register 0 counts the number of grants given to EAB).

DDR SDRAM Memory Interface

- When this field is set to 2, DDR grant count register 1, 2, and 3 count the total number of grants given to DEB1, number of grants given to DEB1 write requests, number of grants given to DEB0 read requests and number of grants given to DEB1 prefetch read requests, respectively. (grant count register 0 counts the number of grants given to EAB).
- When this field is set to 3, DDR grant count register 1, 2, and 3 count the total number of grants given to DEB2, number of grants given to DEB2 write requests, number of grants given to DEB2 read requests and number of grants given to DEB2 prefetch read requests, respectively. (grant count register 0 counts the number of grants given to EAB).

Table 6-14. DDR Grant Control Scheme

DDR Grant Control Field[1:0]	DDR Grant Count Register 1	DDR Grant Count Register 2	DDR Grant Count Register 3	DDR Grant Count Register 4
00	Total EAB Grants	Total DEB0 Grants	Total DEB1 Grants	Total DEB2 Grants
01	Total EAB Grants	DEB0 WR Grants	DEB0 RD Grants	DEB0 Prefetch Grants
10	Total EAB Grants	DEB1 WR Grants	DEB1 RD Grants	DEB1 Prefetch Grants
11	Total EAB Grants	DEB2 WR Grants	DEB2 RD Grants	DEB2 Prefetch Grants

Asynchronous Memory Interface

The EBIU interface allows a view into a variety of memory and peripheral devices, including SRAM, ROM, EPROM, NOR flash memory, and FPGA/ASIC devices. Four asynchronous memory regions (banks) are supported. Each has a unique memory select associated with it, shown in [Table 6-15](#).

Table 6-15. Asynchronous Memory Bank Address Range

Memory Bank Select	Address Start	Address End
AMS[3]	2C00 0000	2FFF FFFF
AMS[2]	2800 0000	2BFF FFFF
AMS[1]	2400 0000	27FF FFFF
AMS[0]	2000 0000	23FF FFFF

Although it is called asynchronous memory interface, each bank in the asynchronous memory region supports synchronous memory device like NOR flash memory. Each bank may be individually configured for one of three operating modes:

- Asynchronous read/write
- Asynchronous page mode read
- Synchronous burst read

Asynchronous Memory Address Decode

The address range allocated to each asynchronous memory bank is fixed at 64M bytes. Many code and data structures may fit within the confines of a single memory bank and not all of an enabled memory bank need be populated.

Asynchronous Memory Interface

Accesses to unpopulated memory of partially populated ASYNC banks do not result in a bus error and will alias to valid ASYNC addresses.

The asynchronous memory signals are defined in [Table 6-16](#). The timing of these pins is programmable to allow a flexible interface to devices of different speeds. Certain pins switch between asynchronous and flash functions depending on the access mode selected for the memory bank being accessed. For example interfaces, see Chapter 18, “System Design”.

Table 6-16. Asynchronous Memory Interface Pins

Pin Name	Type	Asynchronous Function	FLASH Function	Changes with Mode?
ADDR25	O	Address Bus	Clock Output (CLK)	Yes
ADDR24–1	O	Address Bus	Address Bus	No
DATA15–0	I/O	Data Bus	Data Bus	No
$\overline{\text{AMS}}_x$	O	Memory Select	Chip Enable (CE#)	No
$\overline{\text{ABE}}_0$	O	Lower Byte Enable	--	Yes
$\overline{\text{ABE}}_1$	O	Upper Byte Enable	--	Yes
$\overline{\text{AOE}}$	O	Output Enable	Address Valid (ADV#)	Yes
$\overline{\text{ARE}}$	O	Read Enable	Output Enable (OE#)	No
$\overline{\text{AWE}}$	O	Write Enable	Write Enable (WE#)	No
ARDY	O	Ready	Wait (WAIT#)	No

Asynchronous Memory Arbitration

The asynchronous memory arbiter accepts requests from the address resolution block for each of the DEB0, DEB1, DEB2, and the external access bus. The arbiter in the asynchronous memory controller follows a similar, to DDR, but simplified arbitration scheme, where DMA reads and writes have same priority, which, in turn, eliminates the need for “forced write”. Also, there is no prefetch access in ASYNC.

Table 6-17 summarizes the arbitration scheme for asynchronous memory interface. The DEB_ARB_PRIORITY bits of the EBIU_DDRQUE register control the arbitration.

Table 6-17. ASYNC Arbiter Priority Scheme

DEB_ARB_PRIORITY: 000 (0>1>2)	DEB_ARB_PRIORITY: 001 (1>0>2)	DEB_ARB_PRIORITY: 010 (2>0>1)
Core TESTSET	Core TESTSET	Core TESTSET
Urgent DMA DEB0 READ/WRITE DEB1 READ/WRITE DEB2 READ/WRITE	Urgent DMA DEB1 READ/WRITE DEB0 READ/WRITE DEB2 READ/WRITE	Urgent DMA DEB2 READ/WRITE DEB0 READ/WRITE DEB1 READ/WRITE
Core READ/WRITE	Core READ/WRITE	Core READ/WRITE
Normal DMA DEB0 READ/WRITE DEB1 READ/WRITE DEB2 READ/WRITE	Normal DMA DEB1 READ/WRITE DEB0 READ/WRITE DEB2 (USB) READ/WRITE	Normal DMA DEB2 (USB) READ/WRITE DEB0 READ/WRITE DEB1 READ/WRITE

The priority schemes described above are from the arbiter's perspective. The priority schemes are followed by the arbiter only when they are ready to arbitrate, not when EBIU receives requests on the DEB or EAB buses. For example, a DEB bus may indicate Urgent during a request, but if the urgent signal goes away before the arbiter arbitrates, the DEB request is treated as regular request. Burst requests, from core, are arbitrated only in the first beat of a burst; for example, once processor core access is granted, it is granted for the whole burst.

ASYNC Interface Control Registers

The EBIU contains memory-mapped registers that control the access characteristics for each asynchronous memory bank. In addition, a status register is provided to reflect the arbiter status.

Table 6-18. EBIU Memory-Mapped Registers

Name	Address	Description
EBIU_AMGCTL	0xFFC0 0A00	Asynchronous memory global control register (on page 6-57)
EBIU_AMBCTL0	0xFFC0 0A04	Asynchronous memory bank control 0 register (on page 6-58)
EBIU_AMBCTL1	0xFFC0 0A08	Asynchronous memory bank control 1 register (on page 6-58)
EBIU_AMBSCTL	0xFFC0 0A0C	Memory bank select control register (on page 6-63)
EBIU_ARBSTAT	0xFFC0 0A10	Arbiter status register (on page 6-69)
EBIU_MODE	0xFFC0 0A14	Memory mode control register (on page 6-66)
EBIU_FCTL	0xFFC0 0A18	Flash control register (on page 6-67)
Reserved	0xFFC0 0A1C	Reserved

Asynchronous Memory Global Control Register (EBIU_AMGCTL)

The EBIU_AMGCTL register configures global aspects of the controller. It contains bank enables and other information as described in this section. Do not program this register while the ASYNC is in use (for example, when code is being executed from this memory space).

The EBIU_AMGCTL register should be the last control register written to when configuring the processor to access asynchronous memory-mapped asynchronous devices.

Asynchronous Memory Global Control Register (EBIU_AMGCTL)

Address = 0xFFC00A00

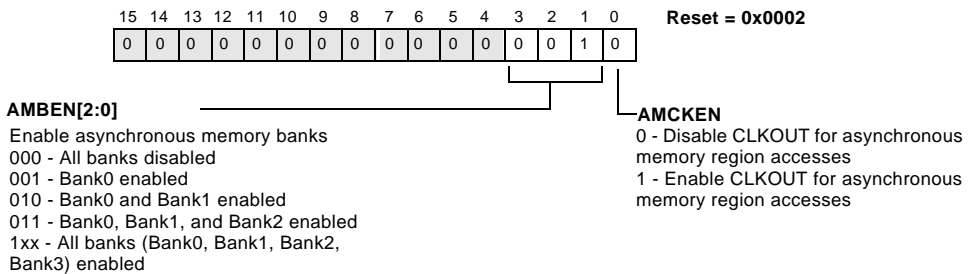


Figure 6-14. Asynchronous Memory Global Control (EBIU_AMGCTL) Register

If a bus operation accesses a disabled asynchronous memory bank, the EBIU responds by acknowledging the transfer and asserting the error signal on the requesting bus. The error signal propagates back to the requesting bus master. This generates a hardware exception to the core, if it is the requester. For DMA-mastered requests, the error is captured in the respective status register.

Asynchronous Memory Interface

If a bank is not fully populated with memory, then the memory likely aliases into multiple address regions within the bank. This aliasing condition is not detected by the EBIU, and no error response is asserted.

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.

Asynchronous Memory Bank Control Registers (`EBIU_AMBCTL0`, `EBIU_AMBCTL1`)

The EBIU asynchronous memory controller has two memory bank control registers (`EBIU_AMBCTL0` and `EBIU_AMBCTL1`). 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`. The configuration in these registers applies in all three operating modes. These registers should not be programmed while the `ASYNC` is in use.

The timing characteristics of the `ASYNC` can be programmed using these four parameters:

- **Setup:** the time between the beginning of a memory cycle (\overline{AMSx} low) and the read-enable assertion (\overline{ARE} low) or write-enable assertion (\overline{AWE} low).
- **Read Access:** the time between read-enable assertion (\overline{ARE} low) and deassertion (\overline{ARE} high).
- **Write Access:** the time between write-enable assertion (\overline{AWE} low) and deassertion (\overline{AWE} high).
- **Hold:** the time between read-enable deassertion (\overline{ARE} high) or write-enable deassertion (\overline{AWE} high) and the end of the memory cycle (\overline{AMSx} high).

Each of these parameters can be programmed in terms of EBIU clock cycles. In addition, there are minimum values for these parameters:

Setup ≥ 1 cycle

Read Access ≥ 1 cycle

Write Access ≥ 1 cycle

Hold ≥ 0 cycle

Asynchronous Memory Interface

Asynchronous Memory Bank Control 0 Register (EBIU_AMBCTL0)

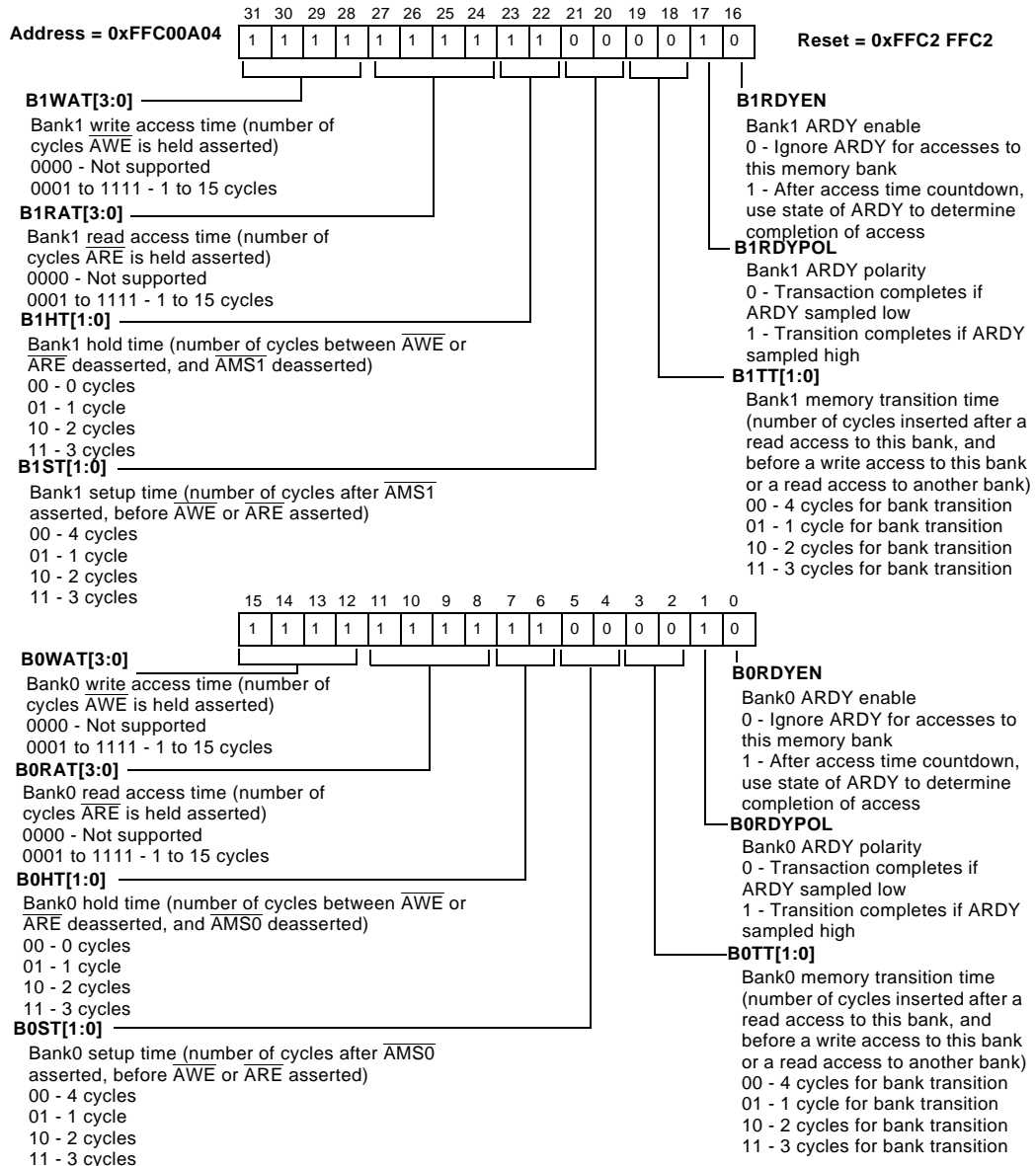


Figure 6-15. Asynchronous Memory Bank Control 0 (EBIU_AMBCTL0) Register

Asynchronous Memory Bank Control 1 Register (EBIU_AMBCTL1)

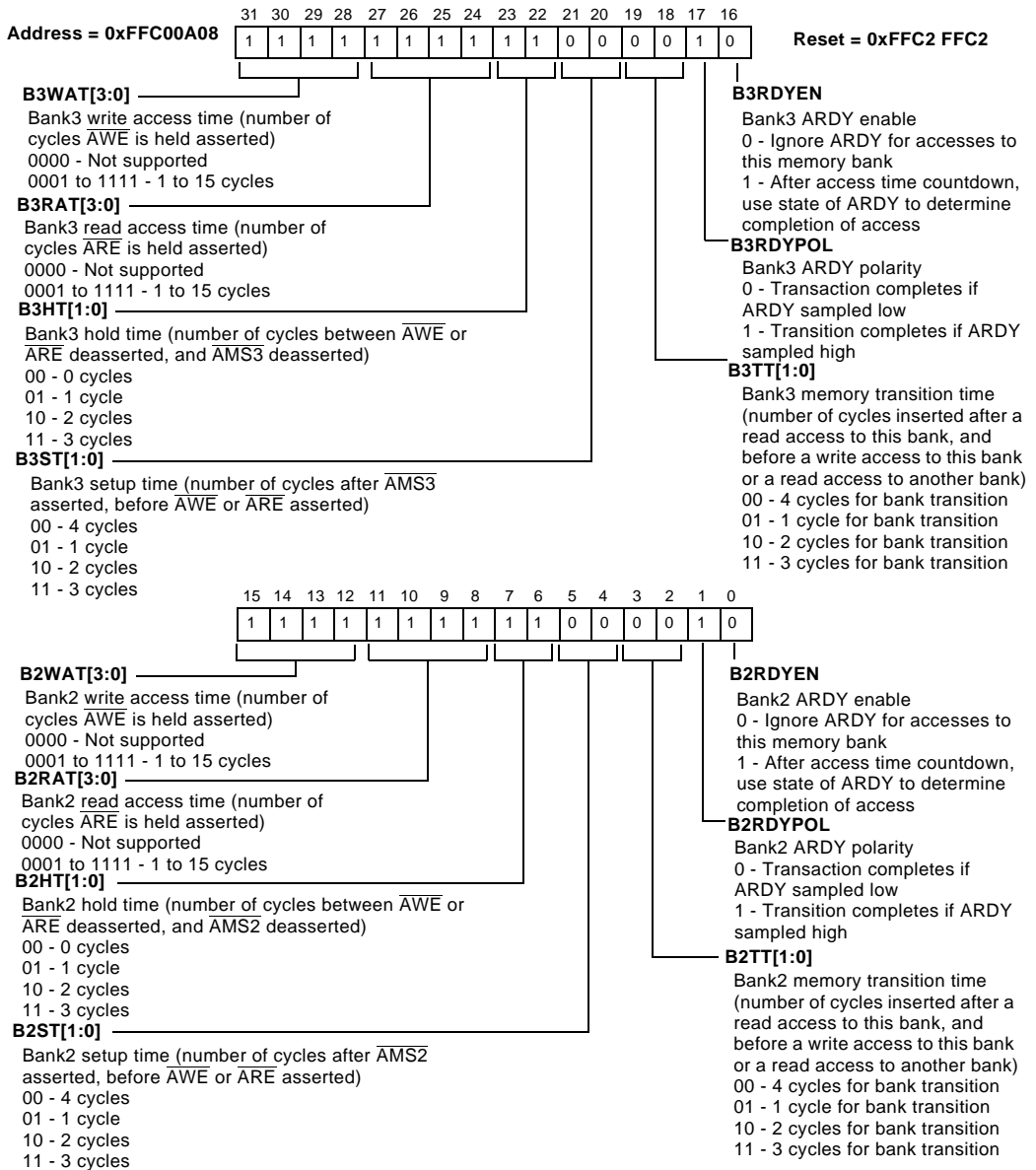


Figure 6-16. Asynchronous Memory Bank Control 1 (EBIU_AMBCTL1) Register

Asynchronous Memory Interface

Avoiding Bus Contention

Because the three-stateable 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.

The `ARDY` input is treated as an asynchronous input, however it must reach the desired value (either asserted or deasserted) more than one `SCLK` cycle before the scheduled rising edge of \overline{AWE} or \overline{ARE} . This determines whether the access is extended or not. Once the transaction is extended by the assertion of `ARDY`, the transaction completes in the cycle after `ARDY` is sampled asserted.

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 6-79.

When using flash memory, the $\overline{\text{WAIT}}$ input should be connected to ARDY.

Memory Bank Select Control Register (EBIU_MBSCTL)

External FIFO devices often do not have a separate chip select pin. As a result, for a read, the FIFO’s output enable ($\overline{\text{OE}}$) pin must be connected the OR (negative AND) of the $\overline{\text{AMS}}$ and the $\overline{\text{ARE}}$. Similarly, the write case requires an OR between $\overline{\text{AMS}}$ and $\overline{\text{AWE}}$. The Blackfin processor provides this function in the EBIU so that an external OR gate is not required. The appropriate $\overline{\text{AMS}}$ function can be selected for each memory bank region in the EBIU_MBSCTL register.

Memory Bank Select Control Register (EBIU_MBSCTL)

Address 0xFFC0 0A0C

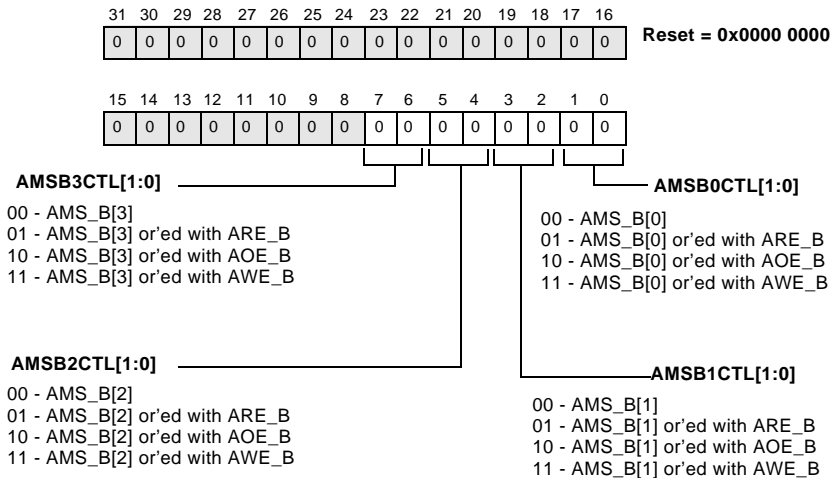


Figure 6-17. Memory Bank Select Control Register (EBIU_MBSCTL)

Asynchronous Memory Interface

Flash Memory Bank Control Registers (EBIU_FCTL, EBIU_MODE)

The asynchronous memory controller (ASYNC) also has two flash memory bank control registers.

- “Flash Memory Bank Control (EBIU_FCTL) Register” on page 6-67
- “Memory Mode Control (EBIU_MODE) Register” on page 6-66

They contain bits for mode selection, page access configuration, and synchronous access configuration. These registers should not be programmed while the ASYNC is in use.

Booting From Page Mode or Synchronous Flash

The EBIU resets to asynchronous mode access. This allows slow asynchronous access to any device during booting without configuration of the EBIU control registers. Synchronous burst mode and asynchronous page mode flash devices power up in asynchronous access mode and thus support an initial access of this type. Once configuration information is read from the external device, the boot code may select a higher performance operating mode.

Access Mode Selection

The EBIU may be configured for standard asynchronous mode access, asynchronous flash mode, asynchronous page mode access, or synchronous burst access. Asynchronous mode access should be used for most devices other than flash. Burst mode and page mode should only be used for read accesses. Flash mode ($MODE=01$) must be used for all writes to flash devices. The burst mode and page mode controls have no effect unless the corresponding access mode is selected.

Pin functionality and supported device width change with mode, as described in [Table 6-19](#).

Table 6-19. EBIU Pin Configuration by Mode

Mode	$\overline{\text{AOE}}$	ADDR[25]	Device Width
Asynchronous	$\overline{\text{AOE}}$	ADDR[25]	8 or 16 bit
Asynchronous flash	$\overline{\text{ADV}}$	ADDR[25]	16 bit
Asynchronous page	$\overline{\text{ADV}}$	ADDR[25]	16 bit
Synchronous burst	$\overline{\text{ADV}}$	CLK	16 bit

Asynchronous Memory Interface

Memory Mode Control (EBIU_MODE) Register

Memory Mode Control Register (EBIU_MODE)

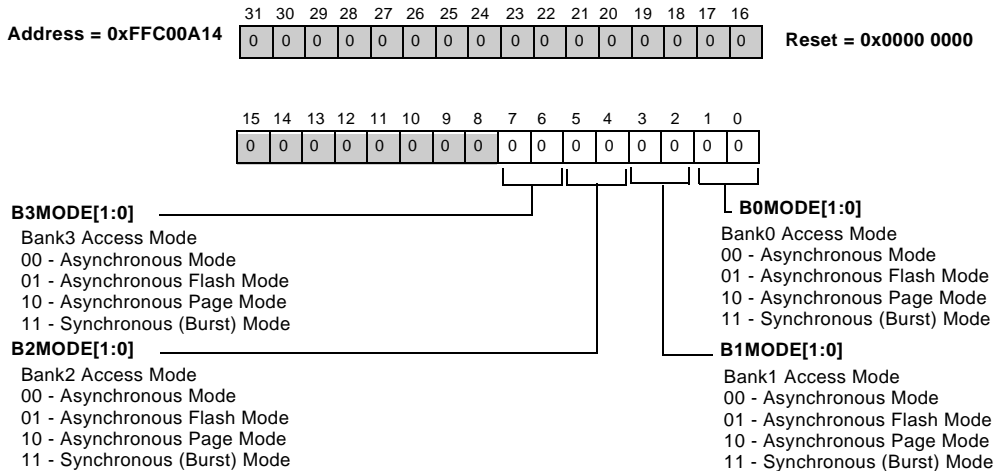


Figure 6-18. Memory Mode Control (EBIU_MODE) Register

Asynchronous Flash Mode

When the access selected mode is asynchronous flash (MODE=01), external bank accesses operate exactly the same as in standard asynchronous mode, except for the pin configuration. This mode should be used when accessing burst devices in non-read array modes.

Flash Memory Bank Control (EBIU_FCTL) Register

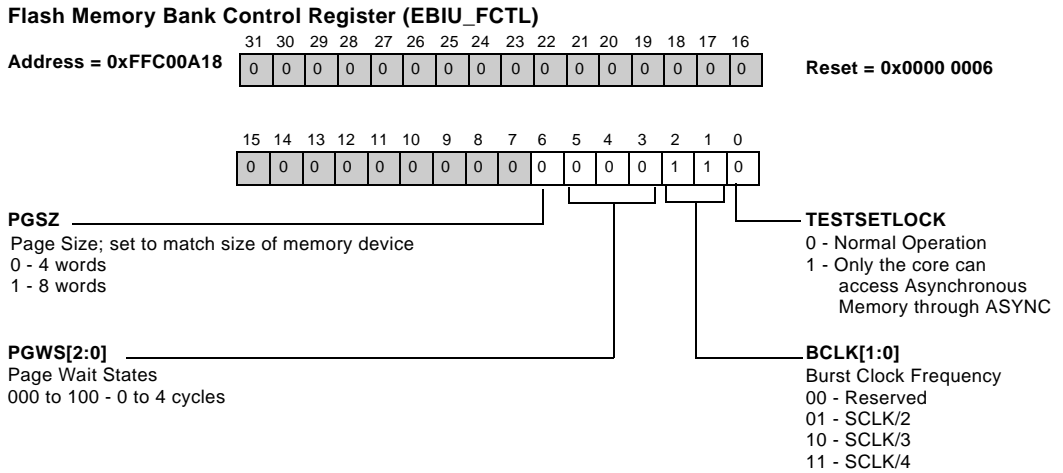


Figure 6-19. Flash Memory Bank Control (EBIU_FCTL) Register

Asynchronous Page Mode

When asynchronous page mode access is selected ($MODE=10$), asynchronous page reads are enabled. Page sizes of 4 or 8 words are supported. When performing a page mode read, the first access in the page proceeds according to the read access time configured in $EBIU_AMBCTLx$. This opens the page. Subsequent reads in that page extend the strobe time by one $SCLK$ plus the number of page wait states. Page mode access is only supported for back-to-back accesses, such as cache line fills (16 words), 64-bit instruction reads (4 words) and 32-bit DMA reads (2 words).


Synchronous Burst Mode

When synchronous mode access is selected ($MODE=11$), synchronous reads are enabled. The burst clock frequency can be configured for $SCLK/2$, $SCLK/3$ or $SCLK/4$.

Asynchronous Memory Interface

This is the frequency of the clock output and determines the frequency of latching data for subsequent beats of a burst. It does not affect any of the other timing parameters (which are still determined by `EBIU_AMBCTLx`).

During the setup time of an access, \overline{ADV} is asserted and the burst clock begins running. The flash device must be configured to latch the address on the rising edge of the clock. \overline{ADV} is asserted for the entire setup time. The first rising edge of CLK occurs one SCLK cycle before setup ends.

 The setup time must be configured appropriately with respect to the SCLK-to-CLK (burst clock) ratio, as follows:

- SCLK : CLK : 4 : 1 => Setup time = 3 SCLK cycles
- SCLK : CLK : 3 : 1 => Setup time = 2 SCLK cycles
- A minimum of 2 SCLK cycles must be programmed regardless of SCLK to CLK ratio.

Once the address is latched, the initial burst access occurs based on the read access timing for that bank. The strobe time is then extended by a burst clock duration for each subsequent beat of the burst. Any access in the burst may be extended by connecting the flash \overline{WAIT} to `ARDY`. The flash device must be configured to deassert `ARDY` at the same time that data is valid. Depending on the flash behavior, it may be necessary to disable the `ARDY` input before asynchronous read or write accesses.

The synchronous read may be burst or single mode, depending on the type of transfer requested. Burst access is only supported for back-to-back reads, such as cache line fills (16 words), 64-bit instruction reads (4 words), and 32-bit DMA reads (2 words). Burst access is not supported for 8-bit accesses. To support any of these burst types, the flash device must be configured for 16-word wrapping burst mode.

When programming the `ASYNC`, before setting the `ASYNC` to synchronous burst mode (`MODE=11`), it is necessary to do `SSYNC` instruction and then wait for $(BxST + BxWAT + BxHT)$ SCLK cycles, where x is the bank

being accessed and the terms are the configuration values from `EBIU_AMBCTL0` or `EBIU_AMBCTL1`. This is to prevent the potential contention of previous FLASH device operation and the upcoming mode change.

EBIU Arbitration Status Register (`EBIU_ARBSTAT`)

When the external flash device is put in non-read array mode for programming, erasing, or checking status, accesses to memory locations in the flash do not return the stored data. As a result, an arbitration locking mechanism is provided to allow the core to prevent DMA access during these operations.

Specifically, the EBIU may be configured to only allow DSP core accesses to the asynchronous memory banks, by setting the `TESTSETLOCK` bit in `EBIU_FCTL`. Once this bit is set, only the core can win arbitration for future accesses. Depending on the speed of any outstanding accesses, it may take many cycles before the arbitration lock takes effect. The `EBIU_ARBSTAT` register contains a status bit to indicate when the arbiter is locked. Once the arbiter is locked, any DMA access to the asynchronous memory banks is stalled until the `TESTSETLOCK` bit is cleared.

Asynchronous Memory Interface

It is recommended that software manage flash memory programming and DMA activities to prevent stalling of the DMA with arbiter locked status.

Arbiter Status Register (EBIU_ARBSTAT)

Address 0xFFC0 0A10

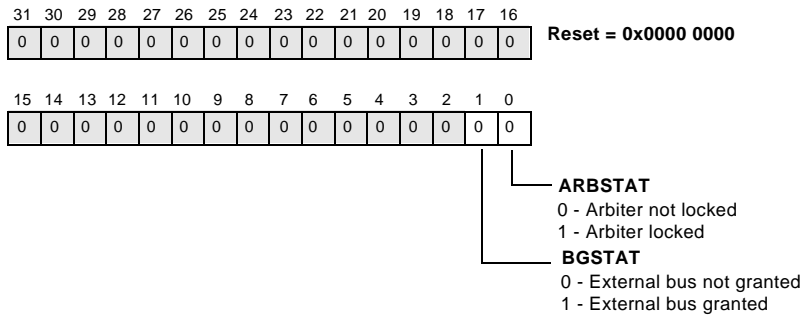


Figure 6-20. Arbiter Status Register (EBIU_ARBSTAT)

Programmable Timing Characteristics

This section describes the programmable timing characteristics for the EBIU. Timing relationships depend on the programming of the ASYNC, whether initiation is from the core or from DMA, and the sequence of transactions (read followed by read, read followed by write, and others).

Asynchronous Accesses by Core Instructions

Some asynchronous memory accesses are caused by core instructions of the type:

```
R0.L = W[P0++] ; /* Read from Asynchronous Memory, where P0
points to a location in Asynchronous Memory */
```

or:

```
W[P0++] = R0.L ; /* Write to Asynchronous Memory */
```

Asynchronous Reads

Figure 6-21 shows two core-initiated asynchronous read bus cycles to the same bank, with timing programmed with setup = 1 cycle, read access = 3 cycles, hold = 2 cycles, and transition time = 1 cycle.

Asynchronous read bus cycles proceed as:

- At the start of the setup period, $\overline{\text{AMSx}}$, the address bus, and $\overline{\text{ABE1-0}}$ become valid, and $\overline{\text{AOE}}$ asserts.
- At the beginning of the read access period and after the setup cycle, $\overline{\text{ARE}}$ asserts.
- At the beginning of the hold period, read data is sampled on the rising edge of CLKOUT . The $\overline{\text{ARE}}$ pin deasserts after this rising edge.
- At the end of the hold period, $\overline{\text{AOE}}$ and $\overline{\text{AMSx}}$ deassert.

Asynchronous Memory Interface

Unless another read of the same memory bank is queued internally, the ASYNC appends the programmed number of memory transition time cycles.

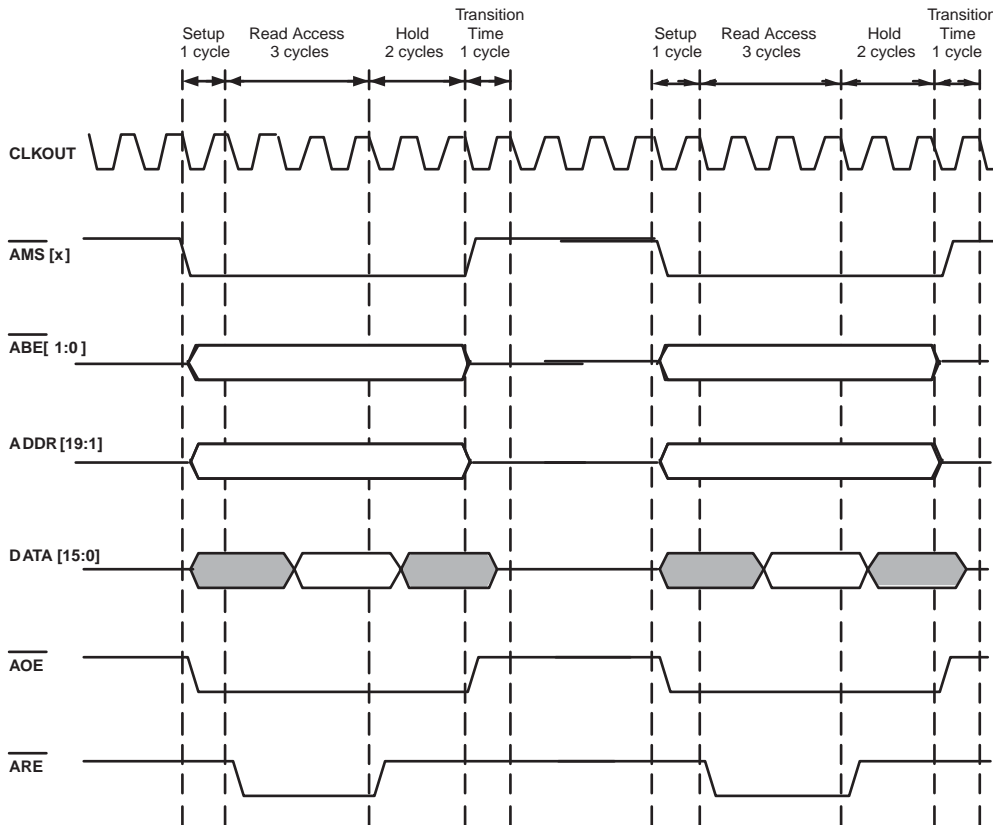


Figure 6-21. Core-Initiated Asynchronous Read Bus Cycles

Asynchronous Writes

Figure 6-22 shows two core-initiated asynchronous write bus cycles to the same bank, with timing programmed with setup = 1 cycle, write access = 2 cycles, hold = 2 cycles, and transition time = 1 cycle.

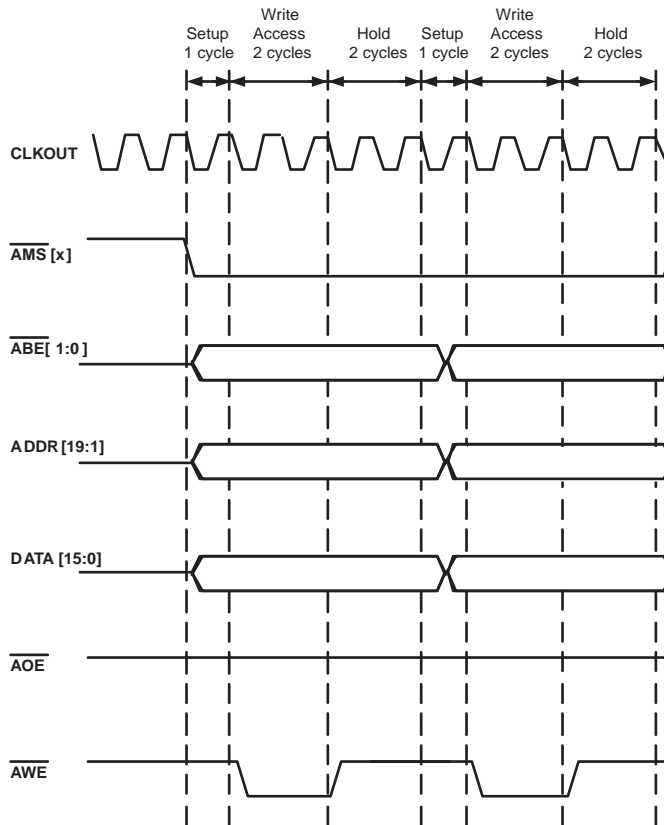


Figure 6-22. Core-Initiated Asynchronous Write Bus Cycles

Asynchronous Memory Interface

The first asynchronous write bus cycle proceeds as:

- At the start of the setup period, $\overline{\text{AMSx}}$, the address bus, data buses, and $\overline{\text{ABE1-0}}$ become valid.
- At the beginning of the write access period, $\overline{\text{AWE}}$ asserts.
- At the beginning of the hold period, AWE deasserts.
- After the hold period, $\overline{\text{AMSx}}$ remains low for the next setup period of the next access.

The second asynchronous write bus cycle proceeds as:

- At the start of the setup period, $\overline{\text{AMSx}}$ is still asserted. The address and data buses and $\overline{\text{ABE1-0}}$ become valid.
- At the beginning of the write access period, $\overline{\text{AWE}}$ asserts.
- At the beginning of the hold period, $\overline{\text{AWE}}$ deasserts.
- After the hold period, $\overline{\text{AMSx}}$ deasserts.

Figure 6-23 shows two higher-speed asynchronous write bus cycles to the same bank, with timing programmed with setup = 1 cycle, write access = 2 cycles, hold = 0 cycles, and transition time = 1 cycle.

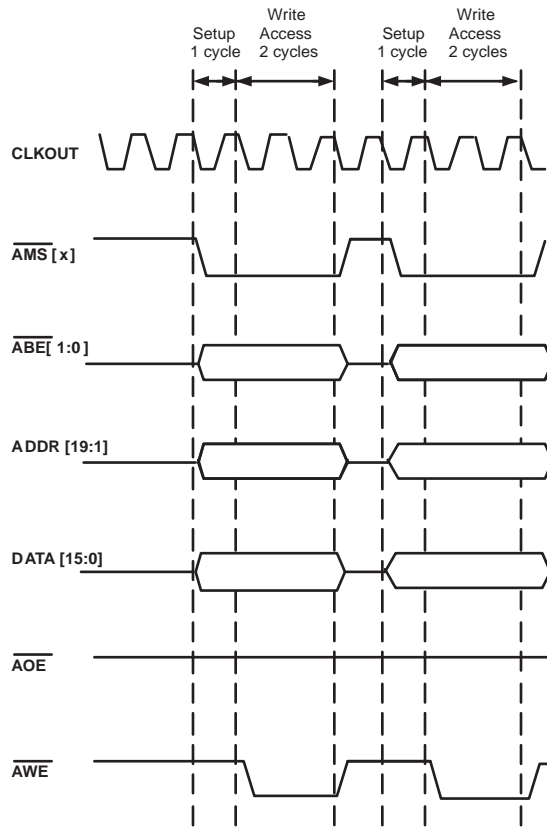


Figure 6-23. High Speed Core-Initiated Asynchronous Write Bus Cycles

Asynchronous Memory Interface

The first asynchronous write bus cycle proceeds as:

- At the start of the setup period, $\overline{\text{AMSx}}$, the address bus, data buses, and $\overline{\text{ABE1-0}}$ become valid.
- At the beginning of the write access period, $\overline{\text{AWE}}$ asserts.
- At the beginning of the hold period, $\overline{\text{AWE}}$ deasserts.
- After the hold period, $\overline{\text{AMSx}}$ deasserts.

The second asynchronous write bus cycle proceeds as:

- At the start of the setup period, $\overline{\text{AMSx}}$, the address bus, data buses, and $\overline{\text{ABE1-0}}$ become valid.
- At the beginning of the write access period, $\overline{\text{AWE}}$ asserts.
- At the beginning of the hold period, $\overline{\text{AWE}}$ deasserts.
- After the hold period, $\overline{\text{AMSx}}$ deasserts.

Asynchronous Writes Followed by Reads

Figure 6-24 shows an asynchronous write bus cycle followed by two asynchronous read cycles to the same bank, with timing programmed with setup = 1 cycle, write access = 2 cycles, read access = 2 cycles, hold = 2 cycles, and transition time = 1 cycle.

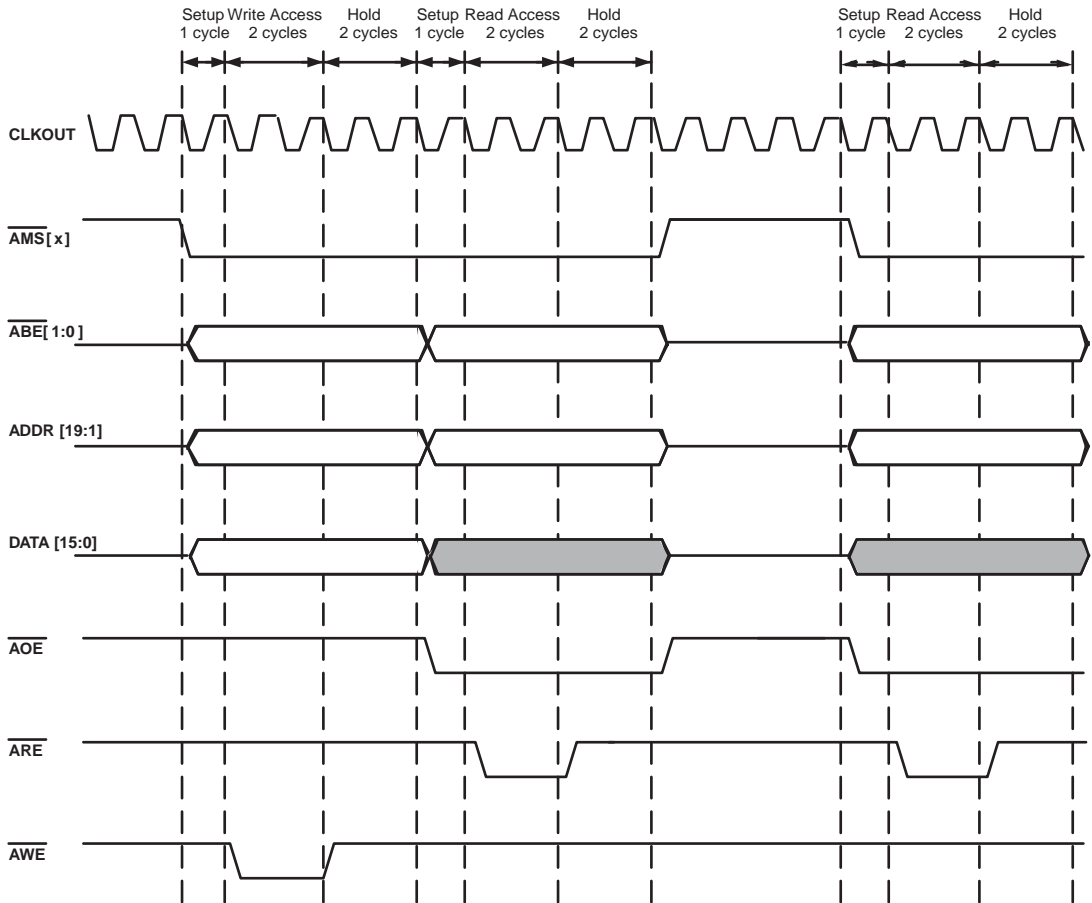


Figure 6-24. Core-Initiated Write and Read Bus Cycles

Asynchronous Memory Interface

The asynchronous write bus cycles proceed as:

- At the start of the setup period, $\overline{\text{AMSx}}$, the address bus, data buses, and $\overline{\text{ABE1-0}}$ become valid.
- At the beginning of the write access period, AWE asserts.
- At the beginning of the hold period, $\overline{\text{AWE}}$ deasserts and $\overline{\text{AMSx}}$ remains low for the setup period of the next access.

The first asynchronous read bus cycle proceeds as:

- At the start of the setup period, $\overline{\text{AMSx}}$ is still asserted. The address bus, and $\text{ABE}[1:0]$ become valid, and AOE asserts.
- At the beginning of the read access period, ARE asserts.
- At the beginning of the hold period, read data is sampled on the rising edge of the EBIU clock. The ARE pin deasserts after this rising edge.
- At the end of the hold period, AOE and $\overline{\text{AMSx}}$ deassert.

The second asynchronous read bus cycle proceeds as:

- At the start of the setup period, $\overline{\text{AMSx}}$, the address bus, and $\overline{\text{ABE1-0}}$ become valid, and AOE asserts.
- At the beginning of the read access period, ARE asserts again.
- At the beginning of the hold period, read data is sampled on the rising edge of the EBIU clock. The ARE pin deasserts after this rising edge.
- At the end of the hold period, $\overline{\text{AOE}}$ and $\overline{\text{AMSx}}$ deassert.

Unless another read of the same memory bank is queued internally, the ASYNC appends the programmed number of memory transition time cycles.

Adding Additional Wait States

The ARDY pin is used to insert extra wait states. An example of this behavior is shown in [Figure 6-25](#), 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.

Asynchronous Memory Interface

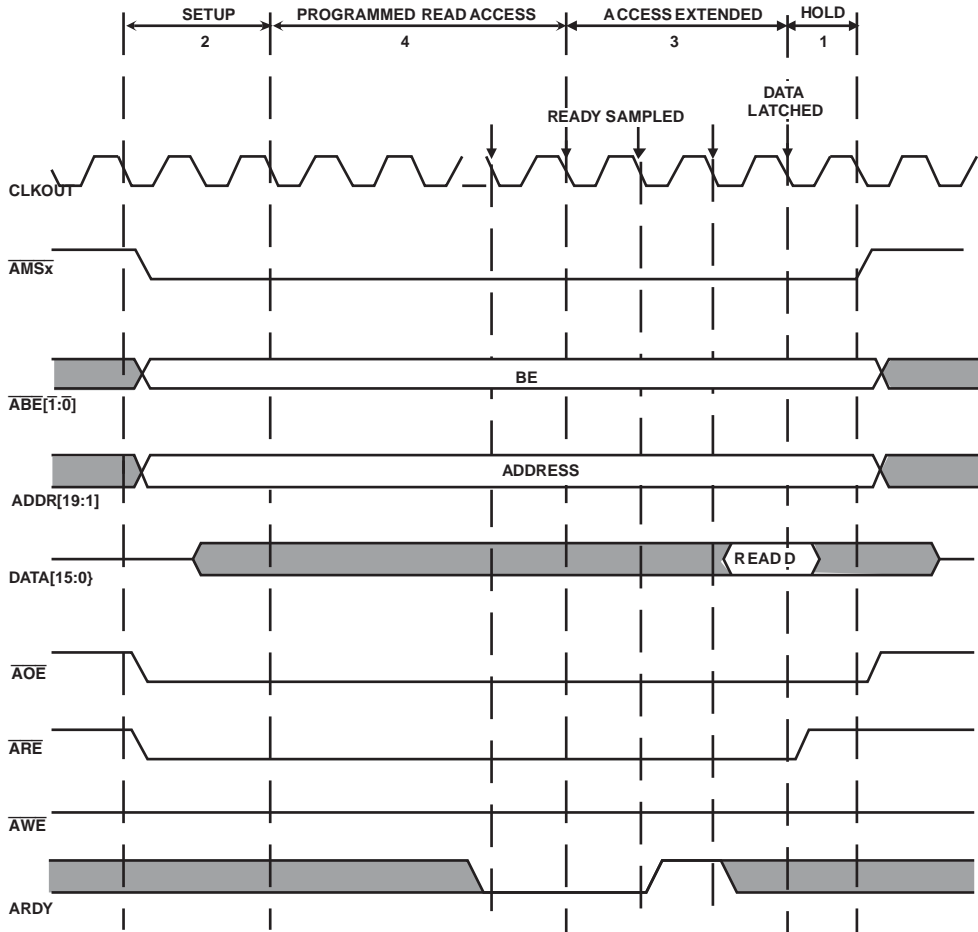


Figure 6-25. Inserting Wait States Using ARDY

Asynchronous Flash Mode Writes and Reads

Figure 6-26 shows an asynchronous flash write bus cycle followed by a read bus cycle to the same bank. Timing is programmed with setup = 1 cycle, write access = 2 cycles, read access = 2 cycles, hold = 2 cycles, and transition = 1 cycle. The bus cycles are identical to the asynchronous mode case, except for the behavior of $\overline{A0E}$. In this case, $\overline{A0E}$ is used to indicate a valid address (\overline{ADV}).

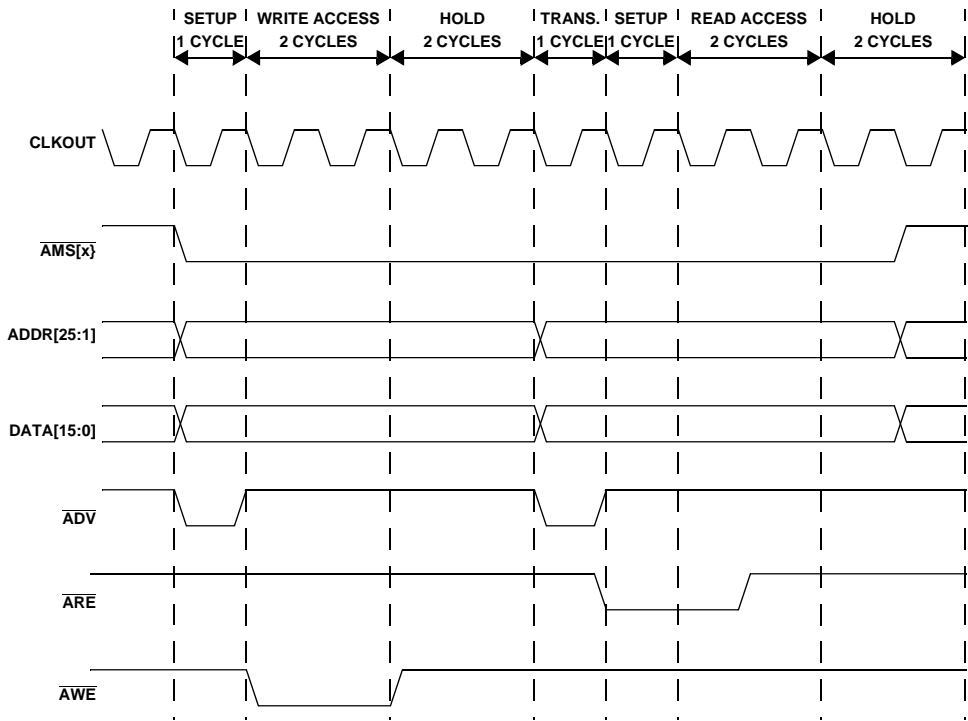


Figure 6-26. Asynchronous Flash Write and Read Bus Cycle

Asynchronous Memory Interface

Asynchronous Page Mode Reads

Figure 6-27 shows an asynchronous page read bus cycle. Timing is programmed with setup = 1 cycle, read access = 3 cycles, hold = 1 cycle, and transition = 1 cycle. One wait state (as specified in the PGWS field of the EBIU_FCTL register) is added to each access in the open page. \overline{AOE} is used to indicate a valid address (\overline{ADV}).

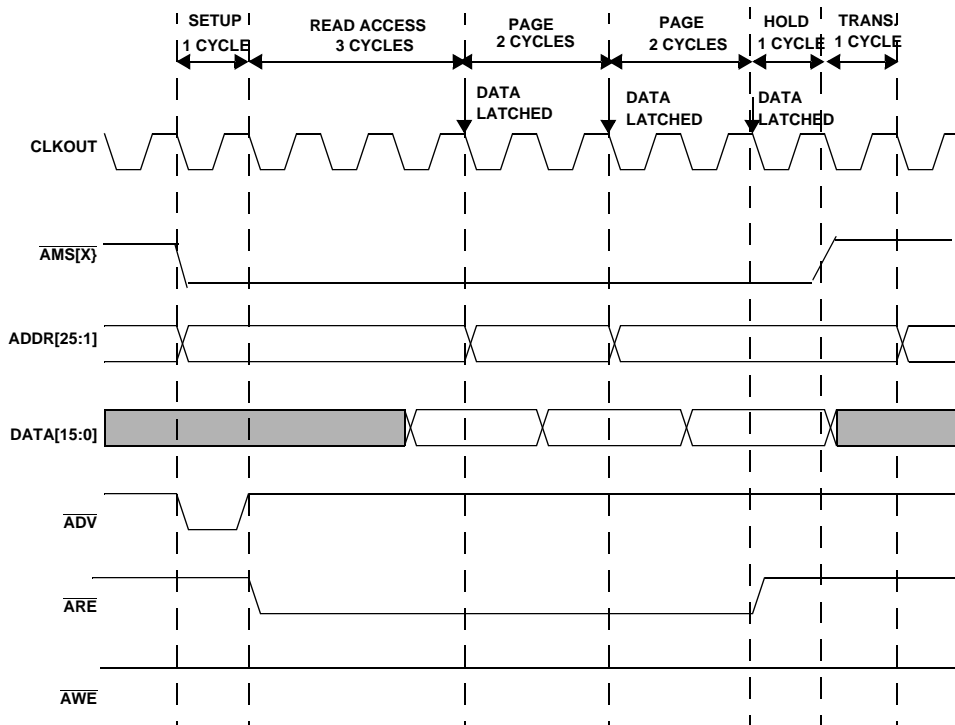


Figure 6-27. Asynchronous Page Mode Read Bus Cycle

Note: Asynchronous Page Mode is only valid for read operations.

Synchronous Burst Mode Read

Figure 6-28 shows a synchronous burst read bus cycle. Timing is programmed with setup = 3 cycles, read access = 2 cycles, hold = 1 cycle, and transition = 1 cycle. The burst clock frequency is $SCLK/2$. The initial burst access is extended using $ARDY$ and the subsequent beats of the burst are latched on every rising CLK edge. \overline{AOE} is used to indicate a valid address (\overline{ADV}) and $ADDR25$ (pin $ADDR25$) is used as the burst clock (CLK).

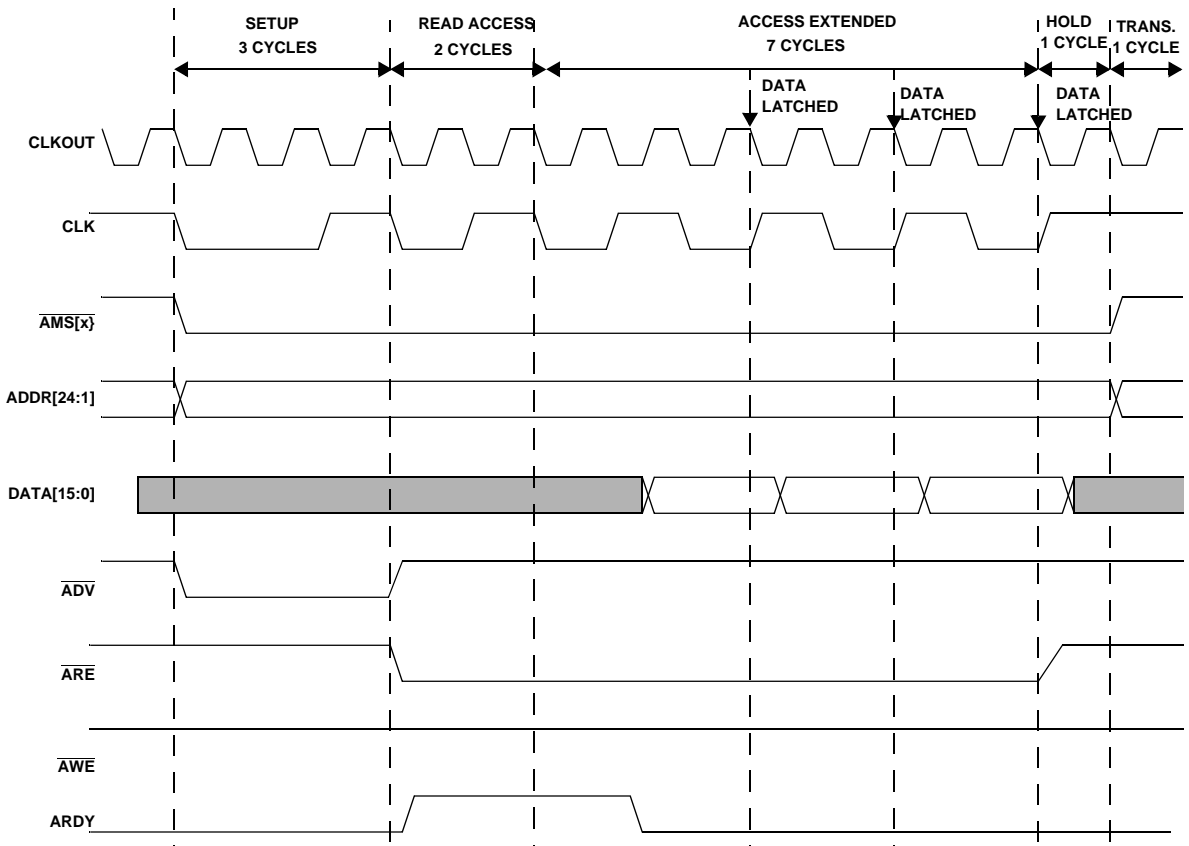


Figure 6-28. Synchronous Burst Mode Read Bus Cycle

Asynchronous Memory Interface

Note: Synchronous mode is only valid for read operations, but does support both burst and non-burst operations.

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.

When the external device requires access to the bus, it asserts the bus request ($\overline{\text{BR}}$) signal. The $\overline{\text{BR}}$ signal is arbitrated with NFC, ATAPI, and ASYNC requests. If no internal request is pending, the external bus request is 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 ($\overline{\text{BG}}$) 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 $\overline{\text{BR}}$, the processor deasserts $\overline{\text{BG}}$ and continues execution from the point at which it stopped.

The processor asserts the $\overline{\text{BGH}}$ pin when it is ready to start another external port access, but is held off because the bus was previously granted. When the bus is granted, the `BGSTAT` bit in the `EBIU_ARBSTAT` 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.


7 PIXEL COMPOSITOR

This chapter describes the pixel compositor (PIXC) and includes the following sections:

- [“Overview” on page 7-2](#)
- [“Interface Overview” on page 7-3](#)
- [“Description of Operation” on page 7-5](#)
- [“Functional Description” on page 7-10](#)
- [“Programming Model” on page 7-35](#)
- [“PIXC Registers” on page 7-35](#)
- [“Programming Examples” on page 7-47](#)

Overview

The pixel compositor (PIXC) for the ADSP-BF54x processor provides data overlay, transparent color, and color space conversion support for active (TFT) flat-panel digital color/monochrome LCD displays or analog NTSC/PAL video output. The color space conversion and text/graphic overlay capabilities, along with visual effect controls, such as transparency control, shorten the processing time on an image data stream, reduce power consumption and save system board space by removing the need for external glue logic.

 The PIXC DMA channels should be configured for 32-bit transfers in order to ensure correct operation.

Features

The PIXC includes these features:

- Hardware-based graphics and text overlays
- YUV 4:2:2 or RGB888 input data formats
- Programmable color space conversion on the main image or the overlay image data path
- Overlay content transparency ratio control
- Transparent color, specified in the desired color space (RGB or YUV)
- Two DMA input channels and one DMA output channel
- Image data stream outputs for active-matrix TFT LCD panels or analog NTSC/PAL displays

Interface Overview

A top-level micro architecture diagram of the PIXC appears in [Figure 7-1](#). As shown in [Figure 7-1](#), the PIXC requires three DMA channels: one for the image data, one for the overlay data and one for storing the results back to L3, L2 or L1 memory. Frame C can also be fed back to the PIXC for multiple stages of processing, taking the place of frame A when this happens. The EPPi can then format the data for an LCD (for example, RGB888 to RGB666 or RGB565).

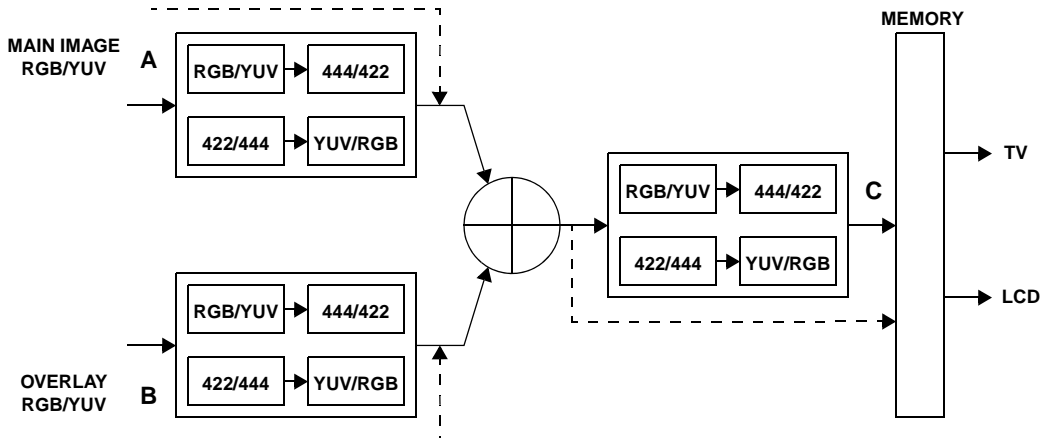


Figure 7-1. Pixel Compositor Top-Level Diagram

Only one color conversion or chroma resampling takes place for each given use of the PIXC block. The user can assume that a color space converter is present in each of the three places shown above. In reality, any input format (for both image and overlay) and any output format can be supported using a single appropriately-positioned color space converter.

Interface Overview

Figure 7-2 shows a more detailed functional view of the PIXC block and shows the relevant connections to the DAB and system interrupt controller (SIC).

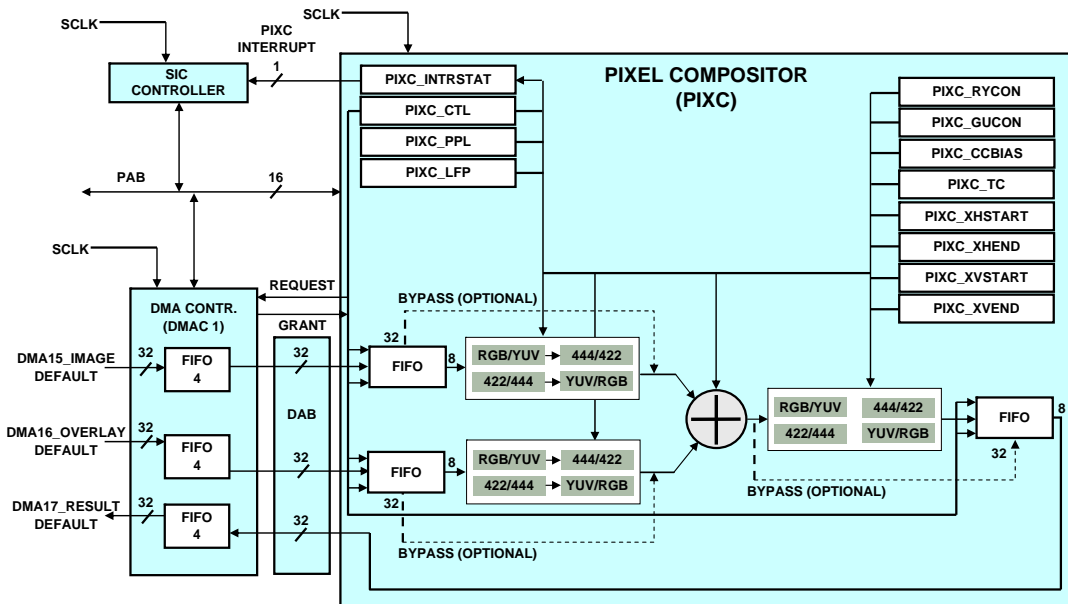


Figure 7-2. Pixel Compositor Functional Block Diagram

Description of Operation

This section describes the operation of the pixel compositor (PIXC).

General Description

The PIXC is used to combine and format the data streams required by a wide variety of digital LCD panels and NTSC/PAL analog video encoders. It provides all the control needed to allow two data streams from two separate data buffers to be combined and converted into appropriate formats for both LCD panels and video output displays. The main image buffer provides the basic background image presented in the data stream. The overlay image buffer allows the user to add foreground text and graphics on top of the main image data stream. This feature is useful for printing additional graphical or textual information on the screen, such as symbols or a menu, while showing the main image in the background.

Overlay is an option and can be enabled or disabled. If it is disabled, the blender/compositor is bypassed and the data stream from the main image buffer goes directly to memory, with optional color space conversion.

Transparent color is just a special case of blending, masking off the blend operation on a pixel-by-pixel basis. In other words, if the overlay region consists of sub-regions that need to be transparent, this can be done by having these sub-regions in any particular color convenient to the programmer, and enabling the transparent color feature of the PIXC. Then, if the color data for a given overlay pixel matches the specified transparent color, the overlay function is masked for that pixel and its data is taken solely from the main image buffer, which is stored in memory in either YUV 4:2:2 interleaved format or RGB888 format.

Description of Operation

Regardless of the data format or buffer structure, each color element is 8 bits wide. If overlay is enabled, a graphics/text overlay data buffer is defined in memory. The color space converter can switch positions among any of the three locations shown in [Figure 7-1 on page 7-3](#); it can be in the image data path, the overlay data path, or after the blender. The exact position of the color space converter depends on the input and output data formats, which are discussed in more detail in the following sections.

Two dedicated DMA channels, with 32-bit bus widths, are used to transfer data from the main image data buffer and the overlay image data buffer into two separate PIXC FIFO buffers, where the data is then unpacked. Each of these FIFO buffers is 32 bits wide and contains 8 entries. The overlay data buffer can not be larger than the image buffer, and the overlay can be set to affect only selected portions of the main image. The position of the overlay in the main image is controlled by memory-mapped registers (MMRs) in the PIXC. In the blender, (8-bit) pixel elements from two buffers are mixed together. One dedicated DMA channel transfers the combined pixel data back to memory.

Since the end display may be a TV (NTSC/PAL) or an LCD panel, and since the image/overlay input buffers may be in either RGB888 or YUV 4:2:2 format, a color space conversion may be needed. The color space conversion is selected according to the input data stream format of the PIXC. A YUV-to-RGB format conversion is necessary if the end display is an LCD and if either of the PIXC input data streams is in YUV 4:2:2 format. Similarly, an RGB-to-YUV format conversion is necessary if the end display is a TV and if either of the PIXC input data streams is in RGB888 format.

If the final display device is an LCD, the output RGB data stream is always be packed in RGB 8-bit serial format when transferring back to memory. Similarly, if the final display device is a TV, the YUV data stream is always packed in YUV 4:2:2 interleaved format when transferring back to memory.

Data Buffer Formats

For the implementation of overlay, the PIXC needs two input data streams from two separate data buffers, a main image buffer and an overlay buffer. The input data in these buffers must be in YUV 4:2:2 or RGB888 format (the main image data and the overlay data can be in different formats). The output data is also in one of these two formats, depending on the output display device being used.

Operation in YUV 4:2:2 Format

Each Y/U/V component is stored in 8 bits of data. The PIXC only accepts a YUV 4:2:2 interleaved format, in the following sequence:

V1, Y1, U1, Y2, V3, Y3, U3, Y4 ...

(Two components with the same suffix number (for example, V1 and U1) implies they are extracted from the same pixel.)

It is the user's responsibility to ensure that the YUV source data to the PIXC is in the correct interleaved format. Therefore, data preprocessing may be necessary in order to meet this requirement.

Figure 7-3 and Figure 7-4 illustrate correct PIXC input buffer structure and data stream format.

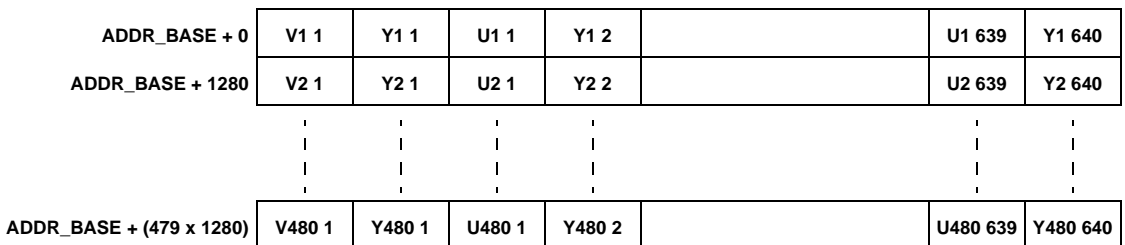


Figure 7-3. YUV 4:2:2 Data in an Interleaved Data Buffer Structure

Description of Operation

B31	B16 B15	B0
Y2	U1	Y1
Y4	U3	Y3
...
...

Figure 7-4. YUV 4:2:2 Expected Data Stream Format to PIXC

The number of pixels per line in YUV mode must be an even number (for both input buffers), and the first chroma component in each line must be a V component.

Operation in RGB888 Format

Each R/G/B component is stored in 8 bits of data. [Figure 7-5](#) and [Figure 7-6](#) illustrate correct PIXC input buffer structure and data stream format.

ADDR_BASE + 0	R1 1	G1 1	B1 1	R1 2	G1 640	B1 640
ADDR_BASE + 1920	R2 1	R2 1	B2 1	B2 2	G2 640	B2 640
	⋮	⋮	⋮	⋮	⋮	⋮
ADDR_BASE + (479 x 1920)	R480 1	G480 1	B480 1	R480 2	G480 640	B480 640

Figure 7-5. RGB888 Data Expected Data Buffer Structure

B31	B16	B15	B0
R12	B11	G11	R11
G13	R13	B12	G12
...
...

Figure 7-6. RGB888 Expected Data Stream Format to PIXC

For operation in RGB format, the total number of pixels in both input buffers must be a multiple of 4, so that the image boundary aligns with a 32-bit DMA word boundary.

DMA Channels

Three peripheral DMA channels can be assigned to the PIXC, as follows:

- A first input DMA channel is used for transferring either a part of the image or the entire main image data to the PIXC from memory (L3, L2, or L1).
- A second input DMA channel is used for transferring the overlay image to the PIXC from memory (L3, L2, or L1).
- An output DMA channel is used for transferring the blended data to memory (L3, L2 or L1).

For more information, see [“Direct Memory Access” on page 5-1](#).

The two input DMA channels take the image data and overlay graphics/text data from their buffers into two separate FIFOs.

Functional Description

Whenever the PIXC is enabled, at least two DMA channels should be enabled and configured appropriately: the image DMA channel and the output DMA channel. Furthermore, when the overlay function of PIXC is enabled, the overlay DMA channel should be enabled and configured as well.

Functional Description

The PIXC implements the following main functions:

- Graphics/text overlay (including video overlay for small frame sizes)
- Transparency control (alpha blending) of the overlay pixel data
- Transparent color (chroma keying) of the overlay stream
- Color space conversion for LCD panels or NTSC/PAL displays

Data Overlay

Overlay is an optional function, so it can be enabled or disabled. If it is disabled, all overlay functionality is bypassed, and a single data stream from the main image data buffer goes directly to the image output buffer, after an optional format conversion. If it is enabled, the blender combines the pixel data from the two image input buffers.

The overlay image is located in a user-defined rectangle within the main image and, in most cases, the overlay image is smaller than the main image. [Figure 7-7](#) illustrates an example of the main and overlay image regions on a screen, where a foreground triangle overlay sits on top of the main image in the background. Although the figure does not show this explicitly, (H-Start, V-Start) can equal (0,0).

In certain situations, it may be beneficial to only DMA the region of the main image that is affected by the overlay instead of bringing in the entire main image. For example, in a situation where the intent is to overlay an

image over the main image and to store the result back over the main image, one could setup a 2D-DMA to only bring in the area of the main image that is affected by the overlay. This may reduce the amount of DMA activity thus potentially improving system performance.

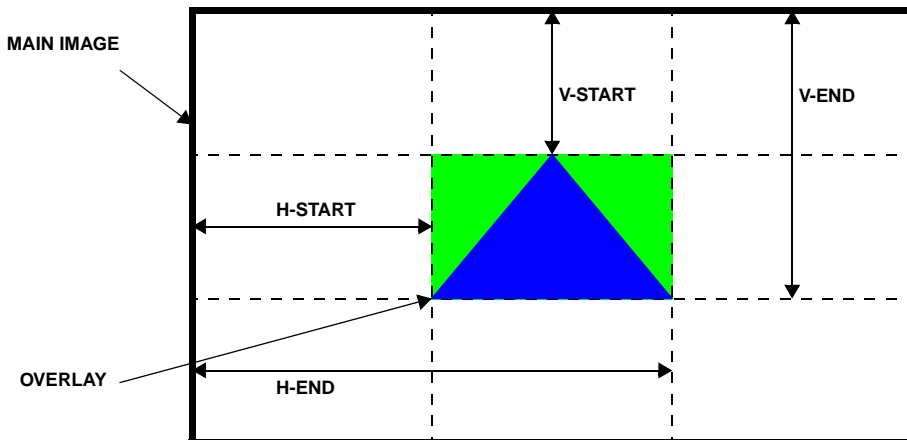


Figure 7-7. Main Image and Overlay Image Region

There are two steps to implement the overlay process:

1. Defining an overlay buffer

The user must define a rectangular region that covers the whole overlay region no matter what shape the overlay content is. The overlay buffer holds the pixel data in the entire rectangular overlay region, which can include some areas where there is no overlay. In memory, these areas have to be filled with the transparent color value. (See [“Transparency Control”](#) on page 7-16.)

Functional Description

2. Configuring the overlay DMA

The user must define a single DMA descriptor for the overlay data transfer. The user must also fill the overlay coordinate registers in the PIXC with appropriate values. The overlay coordinate register set consists of two pairs of registers that specify the top left corner (H-Start, V-Start) and bottom right corner (H-End, V-End) of the overlay, along with a 4-bit register that specifies the α (transparency ratio) value. Each overlay is thus completely specified by a set of five registers. The widths and addresses of these registers are given in “[PIXC Registers](#)” on page 7-35.

There is a set of an additional five such registers that can be used to specify a second overlay region, so that two separate overlay blocks can be defined simultaneously. Furthermore, either or both of these overlay coordinate register sets can be enabled or disabled at one time, since separate enable bits (OVR_A_EN and OVR_B_EN) exist in the PIXC control register for each of the overlay register sets.

If there are more than two overlay blocks needed in a given application, the two sets of overlay registers must be managed by the user to perform the additional overlays. This can be done using an interrupt service routine, where the interrupt from the PIXC is used to re-program the overlay coordinate registers.

The PIXC can generate an interrupt under two conditions: at the end of the last valid overlay and at the end of a frame.

Either of these interrupts can be enabled or disabled. However, the PIXC only has one interrupt line output, so it raises an interrupt (under the appropriate condition) when either of these two interrupts is triggered. If both interrupts are enabled, the interrupt status register of the PIXC indicates which of the two conditions caused the interrupt to occur. Once the PIXC generates an interrupt, it stalls the pixel processing until software

(ISR) clears the interrupt. However, the FIFOs do not stall and keep filling up even when the PIXC is in a stalled state. Both interrupts can be cleared by writing a 1 to the respective interrupt status bits.

After each interrupt (whether it is a last-valid-overlay interrupt or an end-of-frame interrupt), the PIXC restarts processing with coordinate register set A. In other words, at the time of clearing the interrupt:

- If coordinate set A is enabled ($OVR_A_EN = 1$), the PIXC assumes that the first incoming data over the DAB is to be overlaid on the area specified in coordinate set A.
- If coordinate set A is disabled ($OVR_A_EN = 0$), and coordinate set B is enabled ($OVR_B_EN = 1$), the PIXC assumes that the first incoming data over the DAB is to be overlaid on the area specified in coordinate set B.
- If both coordinate sets are disabled, the PIXC flushes the overlay FIFO and make no more data requests on the overlay DMA channel.




The overlay enable bits OVR_A_EN and OVR_B_EN should only be changed inside the interrupt service routines of the PIXC interrupts, or when the overlay block is disabled.

Note that the module enable bit ($PIXC_EN$) is the root enable for the PIXC. Both OVR_A_EN and OVR_B_EN are gated with $PIXC_EN$, so if $PIXC_EN$ is set to zero, the individual overlay enable bits have no effect, and the module remains disabled. When $PIXC_EN$ is programmed to zero, both the image and overlay FIFOs are flushed and no more DMA requests are made on either of the DMA channels.

Once the DMAs are enabled, the PIXC keeps track of the current pixel being displayed from the main image data by reading from two user-programmable registers: $PIXC_PPL$, which stores the number of pixels per line, and $PIXC_LPF$, which stores the number of lines per frame of the display device.

Functional Description

When the pixel count reaches the top left corner (H-Start, V-Start) of overlay data, the PIXC starts the overlay. When the pixel count reaches the top right corner (H-End, V-Start) of overlay data, the PIXC stops the overlay. It starts again at the next line at (H-Start, V-Start+1) and stop at (H-End, V-Start + 1), and so on until the entire overlay frame is processed.

 Internally, the start of the overlay DMA would have been pre-empted by the PIXC before the actual processing of the first overlay pixel, and DMA data would have been requested until the overlay FIFO were full. Similarly, the overlay DMA does not stop at the end of a line. The overlay FIFO continues to be filled with DMA data, even when the current pixel is not an overlay pixel, but the supply of overlay pixels from the overlay FIFO is simply halted.

The PIXC decides whether or not to perform overlay mixing for the current pixel by using the various PIXC register values. Therefore:

- The `PIXC_PPL` and `PIXC_LPF` must be programmed correctly (and cannot be 0).
- The `HSTART` and `HEND` must be less than or equal to `PIXC_PPL`.
- The `VSTART` and `VEND` must be less than or equal to `PIXC_LPF`.

The user can define multiple rectangular regions covering several separate overlays, using the same number of DMA descriptors, where each DMA descriptor corresponds to an overlay region.

Multiple overlay regions are split into two cases:

1. Overlay regions with no horizontal overlap.

This is a straightforward case (shown in [Figure 7-8](#)). Software can maintain separate areas in memory for both overlay regions, with separate H-Start, V-Start, H-End, and V-End coordinates for each

region. After the first overlay is completed, the DMA chain pointer can load the next overlay parameters (index, count, and modifier) to the DMA registers of the corresponding DMA channel.

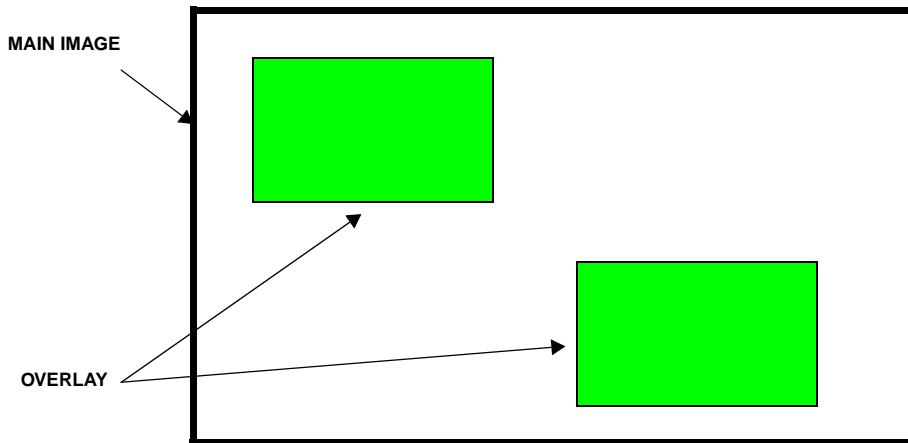


Figure 7-8. Overlay Regions With No Horizontal Overlap

2. Overlay regions with horizontal overlap

In these cases (see [Figure 7-9](#)), software has to maintain a combined overlay region in memory. This includes some in-between area where there is no overlay. This region of memory has to be filled with the transparent color value (explained below). The H-Start, V-Start, H-End and V-End coordinates contain the values of the combined overlay region.

Functional Description

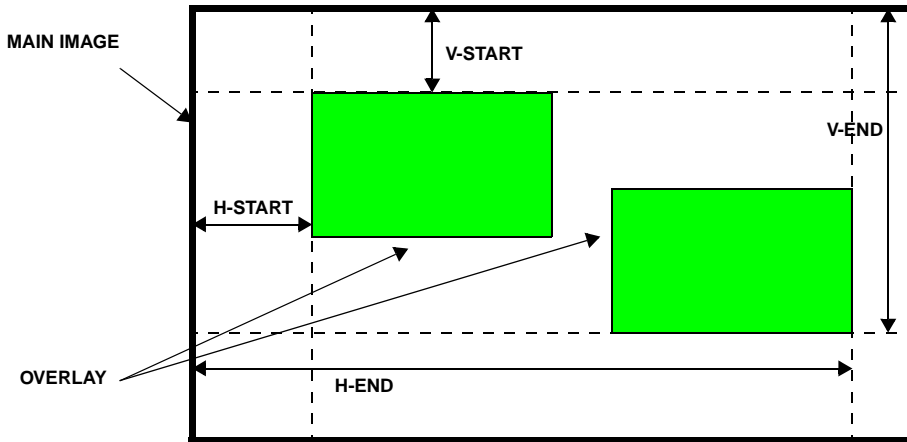


Figure 7-9. Overlay Regions With Horizontal Overlap

Transparency Control

When the overlay function is enabled, each overlay pixel is combined with each main image pixel to generate the displayed output pixel to be displayed. Each pixel combination is controlled by a transparency ratio value alpha (α), a 4-bit value that determines the proportion of overlay and main image that contribute to the output pixel. The pixel combination algorithm can be expressed as:

$$C = \frac{B(\alpha + 1)}{16} + \frac{A(15 - \alpha)}{16}$$

- A: 8-bit pixel data in main frame buffer (“background”)
- B: 8-bit pixel data in overlay buffer (“foreground”)

- C: 8-bit combined pixel data
- α : Transparency ratio code, which is a 4-bit value present in a memory-mapped register

Table 1 lists the multiplying factors for various α values.

Table 7-1. Multiplying Factors for Various α Values

α	Overlay Multiplying Factor	Image Multiplying Factor
0	1/16	15/16
1	2/16	14/16
2	3/16	13/16
3	4/16	12/16
4	5/16	11/16
5	6/16	10/16
6	7/16	9/16
7	8/16	8/16
8	9/16	7/16
9	10/16	6/16
10	11/16	5/16
11	12/16	4/16
12	13/16	3/16
13	14/16	2/16
14	15/16	1/16
15	1	0



Passing the image alone can be achieved by disabling the overlay function.

Functional Description

Rounding is performed at the output of the blender, which rounds the combined pixel data to the nearest integer value.

Transparent Color

A transparent color is a specific color that is removed from one image to reveal another “behind” it. This technique is also referred to as chroma keying. The principal subject is photographed or filmed against a background having a single color, usually in the blue or green spectrums. When the phase of the chroma signal corresponds to the pre-programmed state associated with the background color(s) behind the principal subject, the signal from the alternate background (which in this case comes from the main image channel) is inserted in the composite signal and presented at the output. When the phase of the chroma signal deviates from that associated with the background color(s) behind the principal subject, the picture data associated with the principal subject (in this case, the overlay image) is presented at the output. [Figure 7-10](#) illustrates this concept.

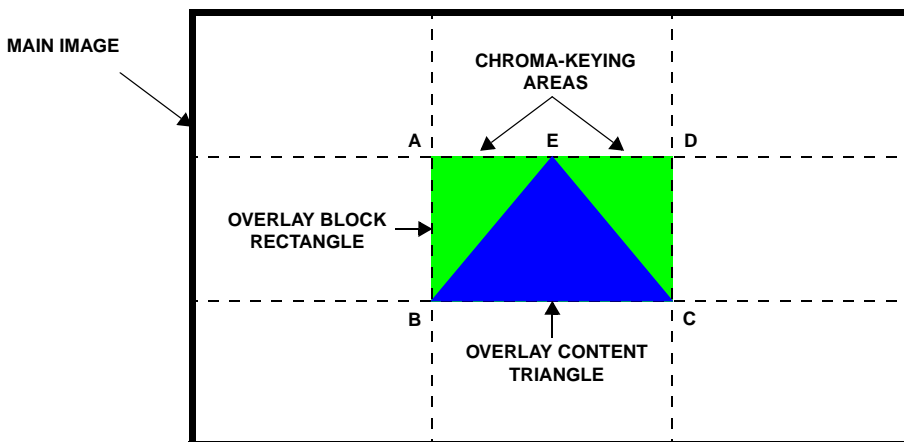



Figure 7-10. Transparent Color (Chroma Keying)

In order to display the main image in the two triangle areas ΔABE and ΔCDE in overlay block ABCD, the data in the overlay buffer corresponding to the pixels in the triangle areas ΔABE and ΔCDE must hold a specific value, called the transparent color.

The PIXC provides a 24-bit MMR (storing three 8-bit color components), for each of the two overlay blocks, in order to designate a particular RGB or YUV value as the transparent color. The transparent color must be in the same format (YUV 4:2:2 or RGB888) as the overlay data, regardless of whether or not a color space conversion is present in the overlay data path. The PIXC then compares each input pixel value on the overlay channel with this transparent color. If there is a match, the overlay pixel at this location is ignored by the blender, and the main image pixel at that location is assigned 100% weight.

 If YUV 4:2:2 is the overlay channel input data format, artifacts may occur at the edge of the transparent color region. In this case, it is preferable to set the `UDS_MOD` bit to 0 (duplicating-dropping mode), in order to get better control of the U and V components at the edge of the transparent color region.

Color Space Conversion

As shown in [Figure 7-1 on page 7-3](#), depending on the input data format and display device used, there may be a color space conversion performed on the data stream of the PIXC. If the input data is in YUV format, a YUV-to-RGB conversion can be performed for output to an LCD panel. If the input data is in RGB format, a RGB-to-YUV conversion can be performed for output to NTSC/PAL displays. The color space conversion may happen on any of the three paths (for example, the main image data path, the overlay image data path, or the combined data path). Register bits are used to specify the input, overlay and output formats.

Functional Description

The color space converter block has three main cases of operation:

1. Both the image and the overlay data are in the same format
2. The image and the overlay data are in different formats
3. Color space conversion only

These are all described in the following section, along with several special usage cases. Note that various scenarios may be shown in the same figure based on the output device chosen, though only a single output destination is supported at one time.

Case 1 - Image and Overlay in the Same Format

Both input data streams (main image and overlay) are in the same format, either YUV 4:2:2 or RGB888, so a color space conversion may be performed after alpha blending, depending on the output type. See [Figure 7-11](#) and [Figure 7-12](#).

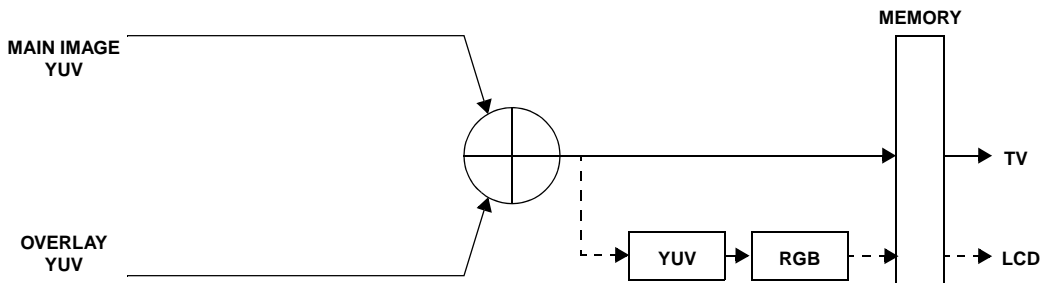


Figure 7-11. Both Input Data Streams in YUV 4:2:2 Format

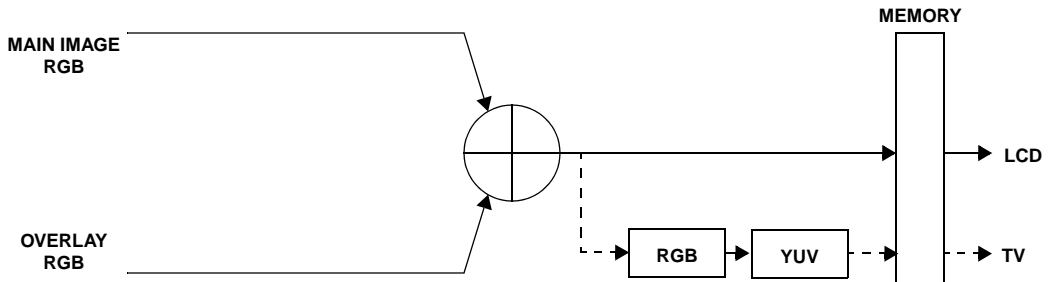


Figure 7-12. Both Input Data Streams in RGB888 Format

Case 2 - Image and Overlay in Different Formats

In this case, the two input data streams are not in the same format. The PIXC has to perform a color space conversion on either the main input stream or the overlay input stream (depending on the required output format) before alpha blending can take place. See [Figure 7-13](#) and [Figure 7-14](#).

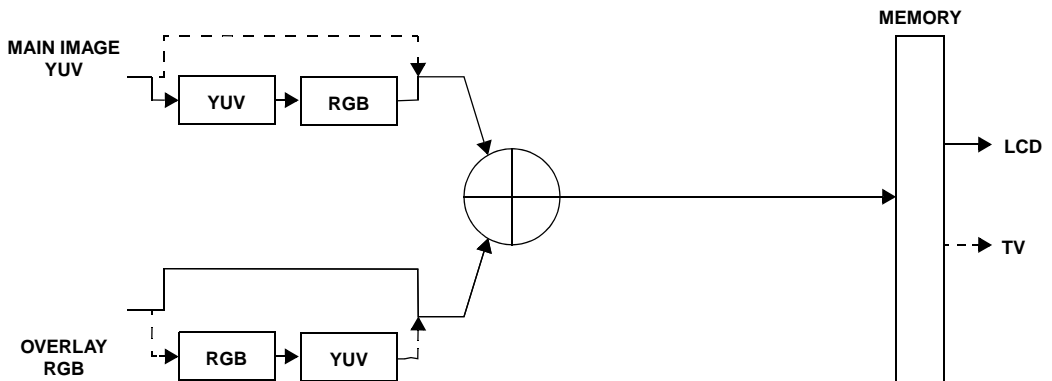


Figure 7-13. Main Image in YUV 4:2:2 and Overlay in RGB888

Functional Description

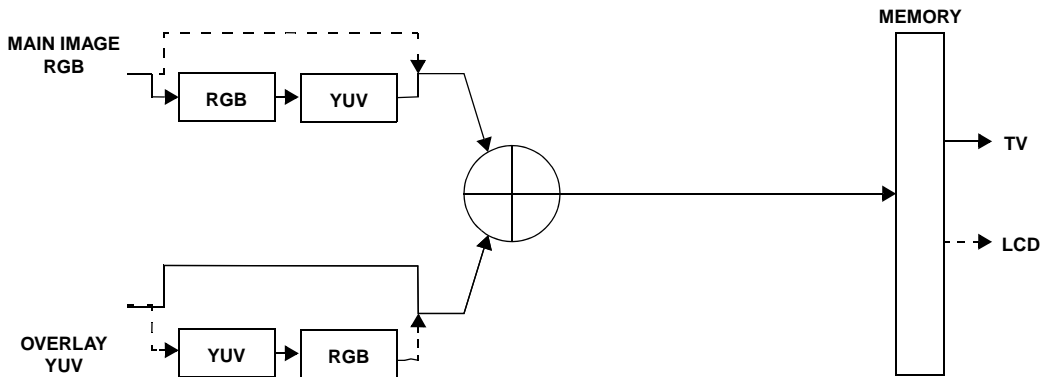


Figure 7-14. Main Image in RGB888 and Overlay in YUV 4:2:2

Case 3 - Color Space Conversion Only

In this case, there is no overlay blending. The main image is brought into the PIXC, the color space converted, and then sent back to memory. See [Figure 7-15](#) and [Figure 7-16](#).

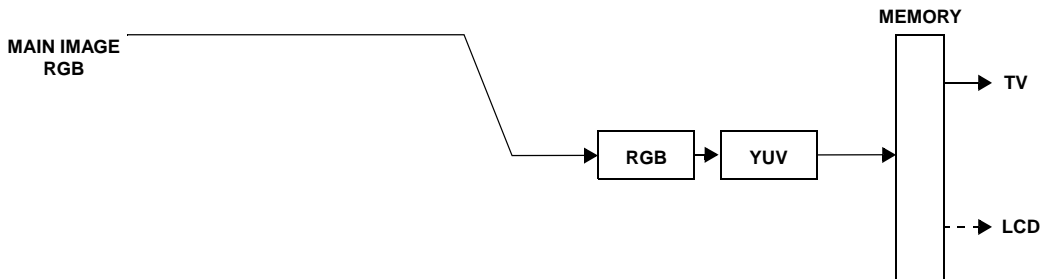


Figure 7-15. Main Image in RGB888 and Output in YUV 4:2:2
(No Overlay)

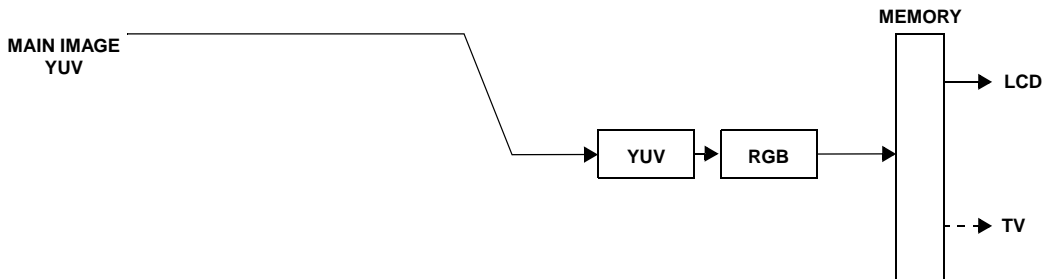


Figure 7-16. Main Image in YUV 4:2:2 and Output in RGB888
(No Overlay)

i For this mode, the register settings are: `PIXC_EN = 1`, `OVR_A_EN = 0`
and `OVR_B_EN = 0`.

Color Space Conversion Matrix Equations

The PIXC color space conversion block implements the following matrix equation:

$$K \times \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ B_3 \end{bmatrix} + \begin{bmatrix} A_{14} \\ A_{24} \\ A_{34} \end{bmatrix} = \begin{bmatrix} C_1 \\ C_2 \\ C_3 \end{bmatrix}$$

The A_{xx} coefficients are 10-bit signed values represented in two's complement format. $A_{11} \dots A_{33}$ are coefficient multipliers (for most cases, it is sufficient to specify these as integers between -512 and 511), and A_{14} , A_{24} , A_{34} are simply offsets added to the result for each row. B_1 , B_2 , B_3 represent the input pixel component values (for example, YUV or RGB) and C_1 , C_2 , C_3 are the output pixel component values. Output pixel values are rounded to the nearest integer.

Functional Description

The constant K equals 1/512. For example, to set A_{11} 's effective value to 0.299, this coefficient's MMR should be programmed to $\text{ROUND}(.299*512)$, or 153. If a coefficient needs to be programmed with a value greater than 1, an extra bit exists in each coefficient's MMR to specify if an extra multiply by 4 must be performed after multiplying the input value by its coefficient. However, this setting can only be specified for an entire row, so if this bit is set, all the coefficients for that row (A_{x1} - A_{x3}) should be calculated as $\text{ROUND}(\text{coeff}*512/4)$. In other words, the constant K effectively becomes 1/128 for that row.

For reference, the matrix equations representing conversion between YUV and RGB formats are:

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.168 & -0.330 & 0.498 \\ 0.498 & -0.417 & -0.081 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix}$$
$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.000 & 0.000 & 1.397 \\ 1.000 & -0.343 & -0.711 \\ 1.000 & 1.765 & 0.000 \end{bmatrix} \begin{bmatrix} Y \\ U \\ V \end{bmatrix} + \begin{bmatrix} -179 \\ 135 \\ -226 \end{bmatrix}$$



For YUV-to-RGB conversion, the PIXC expects the input data to be arranged in the following order: VYUY, VYUY, and so on (see [Figure 7-3](#)). As a result, if the input data is instead arranged as UYVY, UYVY, and so on, then, the columns A_{x2} and A_{x3} of the coefficient matrix are swapped.

For RGB-to-YUV conversion, the PIXC arranges the output data by default in the following order: VYUY, VYUY, and so on. If the output data is desired to instead be arranged as UYVY, UYVY, and so on, then, rows A_{2x} and A_{3x} of the coefficient and bias matrices are swapped.

Color Space Converter Output Thresholds

Each PIXC output sample is 8 bits wide, whether it is an R, G, B, Y, U or V component value. Therefore, any output sample must be in the 0 to 255 range. Since all the coefficients are programmable, some of the inputs, when operated upon by the coefficients, may produce an output outside the 0 to 255 range. In such cases, the PIXC clips the output component's value to 0 or 255.

YUV Conversion Modes

When the color space converter operates between two color spaces, it requires all components of each pixel to be present in the data stream. Therefore, the PIXC internally upsamples the YUV 4:2:2 data stream before a YUV-to-RGB conversion and similarly downsamples the YUV 4:4:4 data stream after a RGB-to-YUV conversion. The resampling always takes place between YUV 4:2:2 and YUV 4:4:4 formats, but a certain flexibility is provided with regard to how the resampling is done by the PIXC in each case.

Upsampling

A YUV 4:2:2-to-YUV 4:4:4 conversion can be performed either by averaging or by duplicating the pixel components. The `UDS_MOD` bit in the `PIXC_CTL` register specifies the upsampling mode. The default setting of this bit is 0, which corresponds to duplication of the chroma components (Us and Vs) from the odd pixels to the even pixels:

YUV 4:2:2 input:	V1Y1, U1Y2, V3Y3, U3Y4, ...
YUV 4:4:4 conversion:	Y1U1V1, Y2U1V1, Y3U3V3, Y4U3V3, ...

Functional Description

Setting the `UDS_MOD` bit to 1 enables the averaging of the chroma components of the preceding and succeeding pixels to obtain the intermediate chroma value. In other words, two consecutive odd-numbered pixels' chroma components are averaged to obtain the intermediate even-numbered pixel's chroma components:

YUV 4:2:2 input:	V1Y1, U1Y2, V3Y3, U3Y4, ...
YUV 4:4:4 conversion:	Y1U1V1, Y2U2V2 [$U2=(U1+U3)/2$, $V2=(V1+V3)/2$], Y3U3V3, Y4U4V4 [$U4=(U3+U5)/2$, $V4=(V3+V5)/2$], ...

If the sum of the preceding and succeeding pixels' U/V components is an odd number, the average is rounded down (truncated to an integer value).

Since the last pixel on a line is always an even-numbered pixel, the last odd pixel value on that line is used as the last even pixel value during upsampling.

Downsampling

A YUV 4:4:4-to-YUV 4:2:2 conversion can be performed either by averaging or by dropping the pixel components. The `UDS_MOD` bit also governs the downsampling mode. Setting the `UDS_MOD` bit to 0 (default) enables the dropping of the chroma components of the even numbered pixels:

YUV 4:4:4 input:	Y1U1V1, Y2U2V2, Y3U3V3, Y4U4V4, ...
YUV 4:2:2 conversion:	V1Y1, U1Y2, V3Y3, U3Y4, ...

Setting the `UDS_MOD` bit to 1 enables the averaging of the chroma components of two consecutive pixels to obtain a single chroma value for a pixel pair:

YUV 4:4:4 input:	Y1U1V1, Y2U2V2, Y3U3V3, Y4U4V4, ...
YUV 4:2:2 conversion:	V12Y1, U12Y2, V34Y3, U34Y4, ...
	$[U12 = (U1+U2)/2, U34=(U3+U4)/2]$
	$[V12 = (V1+V2)/2, V34=(V3+V4)/2]$

PIXC Actions

Table 2 lists the PIXC actions that take place based on any possible combination of image, overlay, and output data formats.

Table 7-2. PIXC Actions

Image Data format	Overlay Data format	Output Data format	PIXC Actions
YUV	No overlay	RGB	US followed by CSC
RGB	No overlay	YUV	CSC followed by DS
YUV	YUV	YUV	US in both paths followed by DS before output
YUV	RGB	RGB	US in image path, CSC in image path
YUV	YUV	RGB	US in both paths, followed by CSC
YUV	RGB	YUV	CSC in overlay path, US in image path, DS before output
RGB	YUV	YUV	CSC in image path, US in overlay path, DS before output
RGB	YUV	RGB	US in overlay path, CSC in overlay path
RGB	RGB	YUV	CSC followed by DS
RGB	RGB	RGB	No CSC, No US, No DS

- CSC = Color Space Conversion
- US = Upsampling
- DS = Downsampling
- YUV = YUV 4:2:2 format
- RGB = RGB888 format

Functional Description

Recommendations

For best results, the overlay should start on an odd-numbered pixel so that the U and V components of the image and the overlay are aligned. Otherwise artifacts may occur in the combined image.

When both the image and the overlay are in YUV 4:2:2 format and the output is also in YUV 4:2:2 format, the duplicating-dropping mode (UDS_MOD) should be used to prevent a low-pass filtering effect on the images.

Special Usage Cases

There are ways by which the PIXC can be made to operate on certain data formats that it does not support in any standard modes. For example, YUV 4:4:4 is similar to RGB 888 with respect to the number of pixels per 32-bit DMA word. So the PIXC can be configured to work with the YUV 4:4:4 data format by intelligently programming the IMG_FORM, OVR_FORM, and OUT_FORM bit fields and the color space conversion coefficients.

These special usage cases are shown in [Figure 7-17](#) through [Figure 7-20](#).

Example 1 - Currently Defined Mode

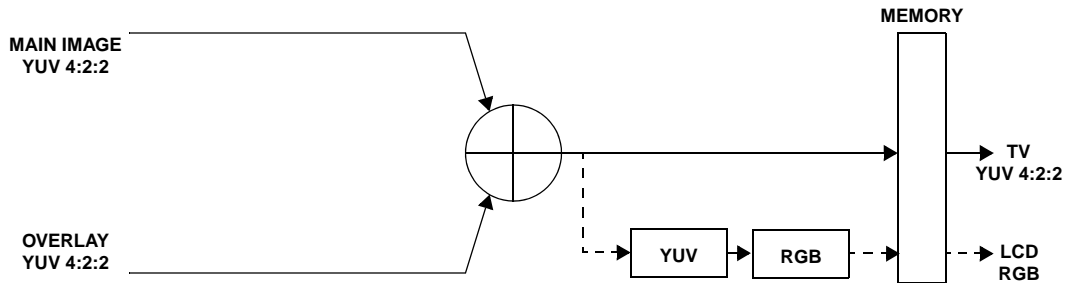


Figure 7-17. Example 1 - Currently Defined Mode

Example 1 - Special Usage of This Mode

- `IMG_FORM = YUV`
- `OVR_FORM = YUV`
- `OUT_FORM = RGB`
- All CSC coefficients = 1

Functional Description

In the special usage of this mode, YUV 4:2:2 inputs produce a blended YUV 4:4:4 data stream. A CSC matrix with coefficients of 1 is needed.

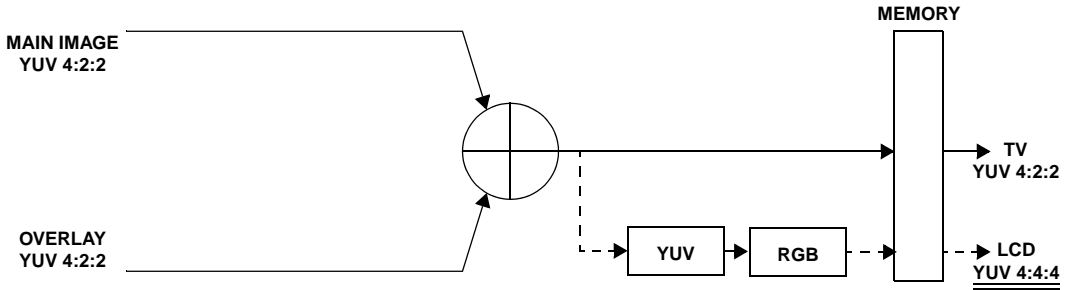


Figure 7-18. Example 1 - Special Usage

Example 2 - Currently Defined Mode

- `IMG_FORM = RGB`
- `OVR_FORM = RGB`
- `OUT_FORM = YUV`
- All CSC coefficients = 1

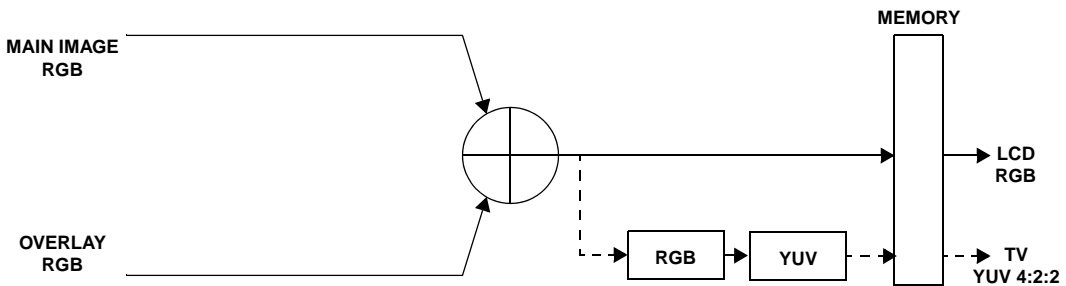


Figure 7-19. Example 2 - Currently Defined Mode

Example 2 - Special Usage of This Mode

In the special usage of this mode, YUV 4:4:4 input produces a blended YUV 4:2:2 or YUV 4:4:4 data stream. A CSC matrix with coefficients of 1 is needed.

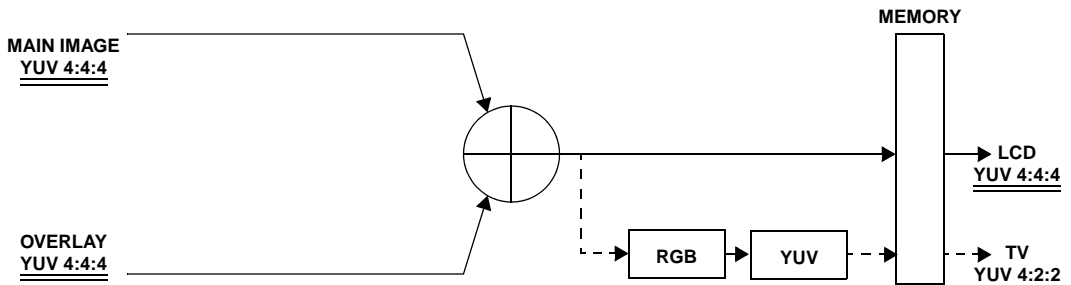


Figure 7-20. Example 2 - Special Usage

Functional Description

Example 3 - Currently Defined Mode

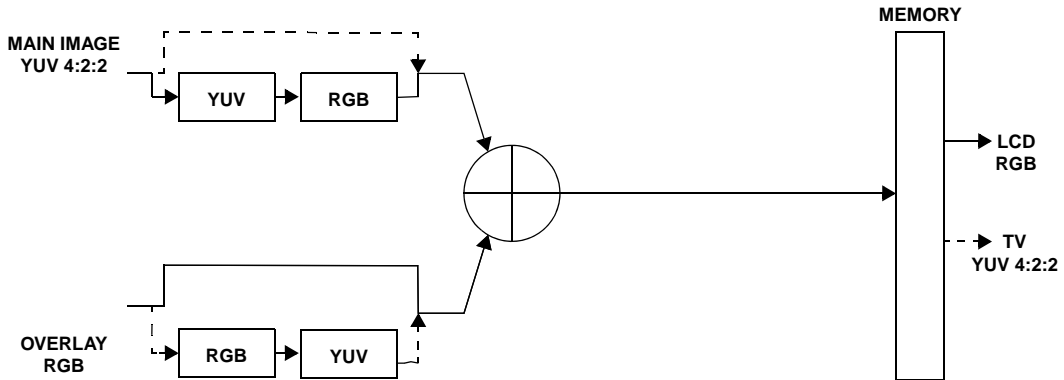


Figure 7-21. Example 3 - Currently Defined Mode

Example 3 - Special Usage of This Mode

In the special usage of this mode, a YUV 4:4:4 input stream and a YUV 4:2:2 input stream can be blended to produce either a YUV 4:4:4 or a YUV 4:2:2 output stream.

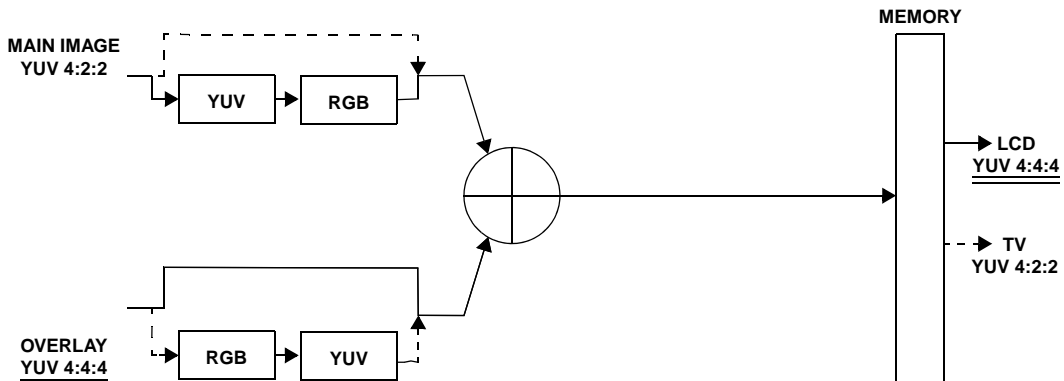


Figure 7-22. Example 3 - Special Usage

If the image format is YUV 4:2:2 and the overlay format is YUV 4:4:4:

- For YUV 4:2:2 output, program `IMG_FORM = YUV`, `OVR_FORM = RGB`, and `OUT_FORM = YUV`. Also program all the CSC coefficients to 1.
- For YUV 4:4:4 output, program `IMG_FORM = YUV`, `OVR_FORM = RGB`, and `OUT_FORM = RGB`. Also program all the CSC coefficients to 1.

If the image format is YUV4:4:4 and the overlay format is YUV 4:2:2, simply interchange `IMG_FORM` and `OVR_FORM` in the above programming cases.

Example 4 - Currently Defined Mode

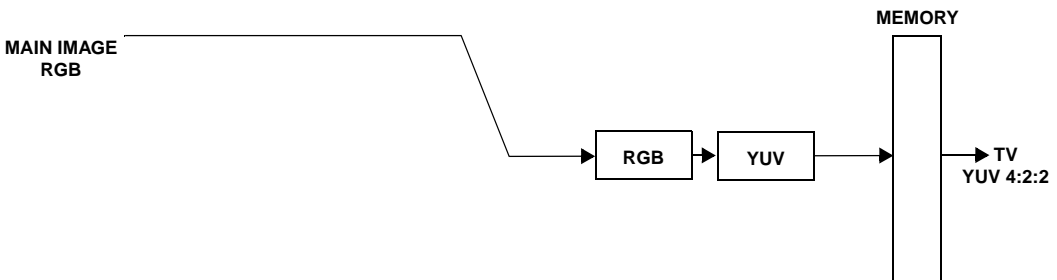


Figure 7-23. Example 4 - Currently Defined Mode

Example 4 - Special Usage of This Mode

- `PIXC_EN = 1`
- `OVR_A_EN = 0`
- `OVR_B_EN = 0`
- `IMG_FORM = RGB`

Functional Description

- `OUT_FORM = YUV`
- All CSC coefficients = 1

In the special usage of this mode, a simple downsampling from YUV 4:4:4 to YUV 4:2:2 is performed. Only color space conversion is enabled, using the `PIXC_EN` bit. A CSC matrix with coefficients of 1 is needed.

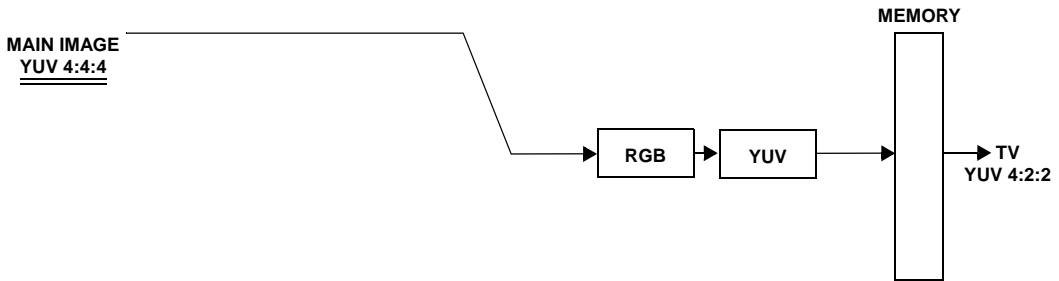


Figure 7-24. Example 4 - Special Usage

Programming Model

The following sections describe the PIXC programming model.

The output destination of the PIXC can be either an L3 frame buffer or an L2/L1 line buffer. As a recommendation for saving DMA bandwidth:

- If the size of the overlay content is relatively big, it is more efficient to send, through the DMA, the output of the PIXC to an L2/L1 line buffer, and then send the data from that line buffer directly through the EPPI to the display device.
- If the size of the overlay content is relatively small, it is more efficient to send, through the DMA, the output of the PIXC back to the L3 frame buffer, and then send the data from that frame buffer through the EPPI to the display device.

PIXC Registers

The PIXC has memory-mapped registers (MMRs) that regulate its operation. These registers are listed in [Table 7-3](#). Descriptions and bit diagrams for each of these MMRs are provided in the following sections.

Table 7-3. List of PIXC Memory-Mapped Registers


Name	Width	Address	Function
PIXC_CTL on page 7-38	16	0xFFC04400	Overlay enable, resampling mode selection, input/output data format selection, transparent color enable, watermark level selection, image/overlay FIFO status.
PIXC_PPL on page 7-39	16	0xFFC04404	Holds the number of pixels per line of the display.
PIXC_LPF on page 7-39	16	0xFFC04408	Holds the number of lines per frame of the display.

PIXC Registers

Table 7-3. List of PIXC Memory-Mapped Registers (Cont'd)

Name	Width	Address	Function
PIXC_AHSTART on page 7-40	16	0xFFC0440C	Contains horizontal start pixel information of the overlay data (set A).
PIXC_AHEND on page 7-40	16	0xFFC04410	Contains horizontal end pixel information of the overlay data (set A).
PIXC_AVSTART on page 7-41	16	0xFFC04414	Contains vertical start pixel information of the overlay data (set A).
PIXC_AVEND on page 7-40	16	0xFFC04418	Contains vertical end pixel information of the overlay data (set A).
PIXC_ATRANSP on page 7-42	16	0xFFC0441C	Contains the transparency ratio (set A).
PIXC_BHSTART on page 7-40	16	0xFFC04420	Contains horizontal start pixel information of the overlay data (set B).
PIXC_BHEND on page 7-40	16	0xFFC04424	Contains horizontal end pixel information of the overlay data (set B).
PIXC_BVSTART on page 7-41	16	0xFFC04428	Contains vertical start pixel information of the overlay data (set B).
PIXC_BVEND on page 7-41	16	0xFFC0442C	Contains vertical end pixel information of the overlay data (set B).
PIXC_BTRANSP on page 7-42	16	0xFFC04430	Contains the transparency ratio (set B).
PIXC_INTRSTAT on page 7-42	16	0xFFC0443C	Overlay interrupt configuration/status.
PIXC_RYCON on page 7-43	32	0xFFC04440	Color space conversion matrix register. Contains the R/Y conversion coefficients.
PIXC_GUCON on page 7-44	32	0xFFC04444	Color space conversion matrix register. Contains the G/U conversion coefficients.
PIXC_BVCON on page 7-45	32	0xFFC04448	Color space conversion matrix register. Contains the B/V conversion coefficients.
PIXC_CCBIAS on page 7-46	32	0xFFC0444C	Bias values for the color space conversion matrix.
PIXC_TC on page 7-47	32	0xFFC04450	Holds the transparent color value.

All PIXC registers have a default value of zero, except the transparency ratio registers which have a default value of 0xF.

 The programmer should avoid writing to any of the MMRs when the module is enabled. Writing to the MMRs during the module enabled state can lead to unpredictable behavior of the PIXC. All MMRs can be read when the PIXC is in the enabled state, and this does not cause any change of status in the PIXC, but register writes should happen only when the PIXC is disabled, or stalled by an interrupt condition.

The following sections provide bit descriptions of the PIXC registers.

PIXC Control (PIXC_CTL) Register

The PIXC_CTL register (Figure 7-25) provides overlay enable, resampling mode selection, input/output data format selection, transparent color enable, watermark level selection, and image/overlay FIFO status.

PIXC Control Register (PIXC_CTL)

Read/Write

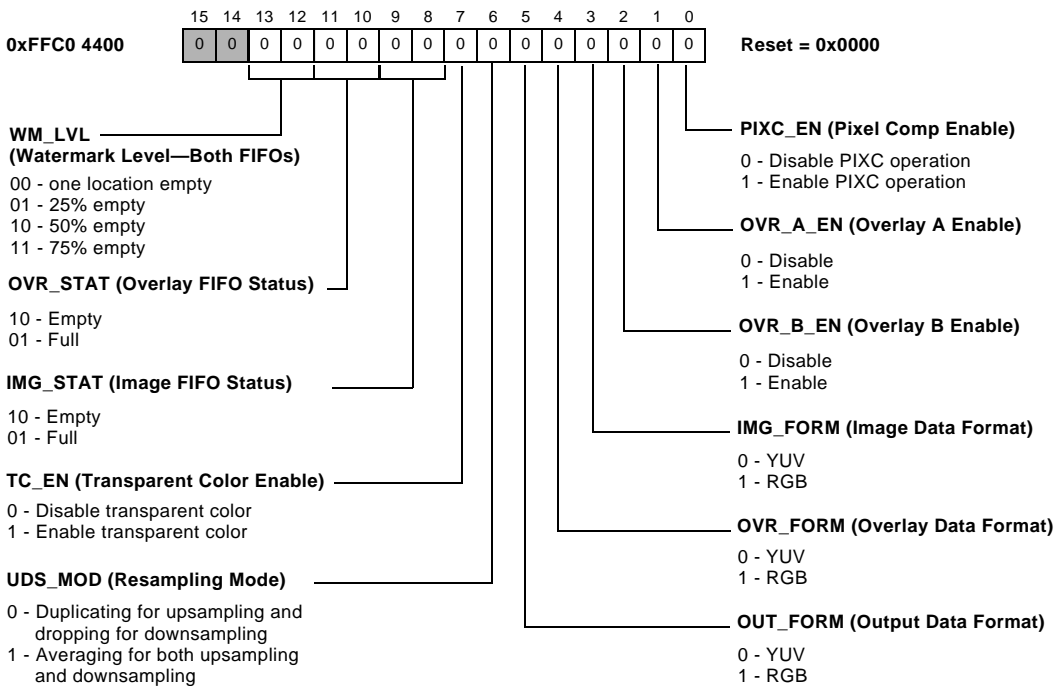


Figure 7-25. PIXC Control (PIXC_CTL) Register



Watermarking is described in the ADSP-BF54x Hardware Reference (Volume 2 of 2).

PIXC Pixels Per Line (PIXC_PPL) Register

The `PIXC_PPL` register (Figure 7-26) provides the number of pixels per line of the display.

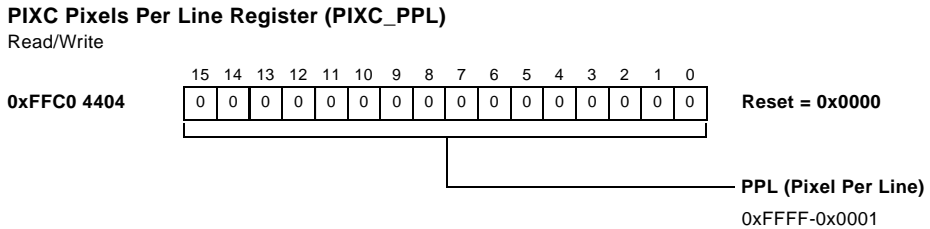


Figure 7-26. PIXC Pixels Per Line (PIXC_PPL) Register

PIXC Lines Per Frame (PIXC_LPF) Register

The `PIXC_LPF` register (Figure 7-26) provides the number of lines per frame of the display.

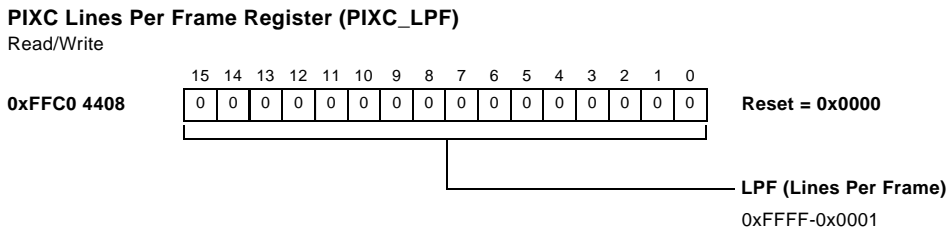


Figure 7-27. PIXC Lines Per Frame (PIXC_LPF) Register

PIXC Horizontal Start (PIXC_xHSTART) Registers

The PIXC_AHSTART and PIXC_BHSTART registers (Figure 7-28) provide the horizontal start pixel coordinates of the overlay data.

PIXC Overlay x Horizontal Start Registers (PIXC_xHSTART)

Read/Write

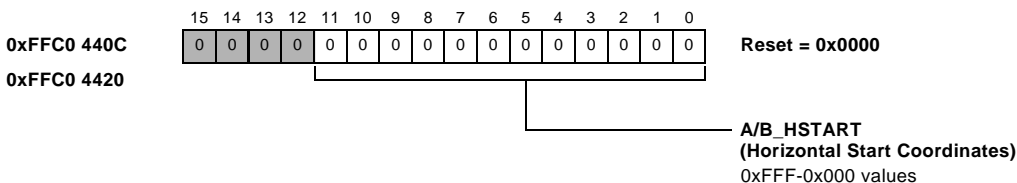


Figure 7-28. PIXC Horizontal Start (PIXC_xHSTART) Registers

PIXC Horizontal End (PIXC_xHEND) Registers

The PIXC_AHEND and PIXC_BHEND registers (Figure 7-29) provide the horizontal end pixel coordinates of the overlay data.

PIXC Overlay x Horizontal End Registers (PIXC_xHEND)

Read/Write

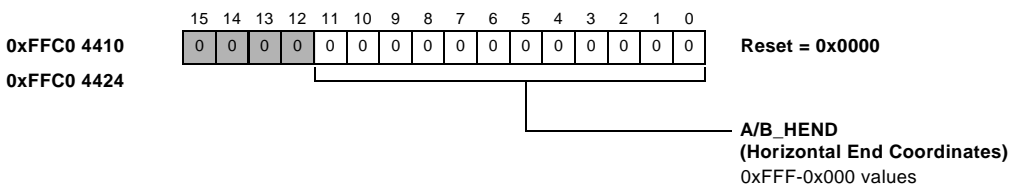


Figure 7-29. PIXC Horizontal End (PIXC_xHEND) Registers

PIXC Vertical Start (PIXC_xVSTART) Registers

The `PIXC_AVSTART` and `PIXC_BVSTART` registers (Figure 7-30) provide the vertical start pixel coordinates of the overlay data.

PIXC Overlay x Vertical Start Registers (PIXC_xVSTART)

Read/Write

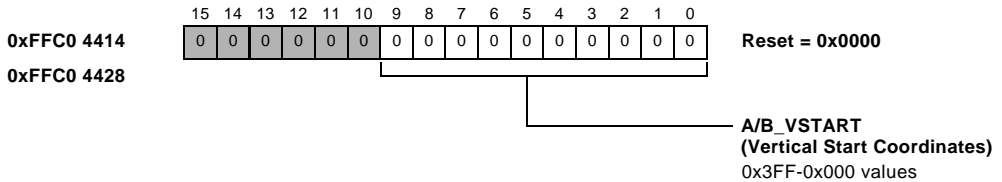


Figure 7-30. PIXC Vertical Start (PIXC_xVSTART) Registers

PIXC Vertical End (PIXC_xVEND) Registers

The `PIXC_AVEND` and `PIXC_BVEND` registers (Figure 7-31) provide the vertical end pixel coordinates of the overlay data.

PIXC Overlay x Vertical End Registers (PIXC_xVEND)

Read/Write

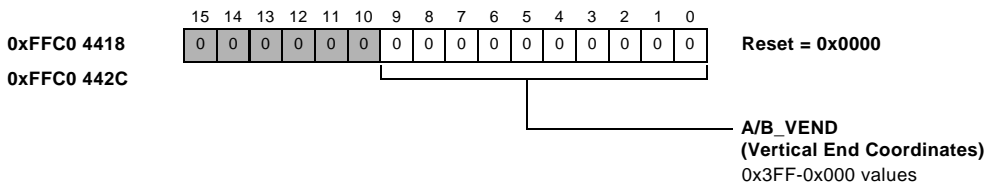


Figure 7-31. PIXC Vertical End (PIXC_xVEND) Registers

PIXC Transparency Value (PIXC_xTRANSP) Registers

The PIXC_ATRANSF and PIXC_BTRANSP registers (Figure 7-32) provide the overlay transparency ratio values.

PIXC Overlay x Transparency Value Registers (PIXC_xTRANSP)

Read/Write

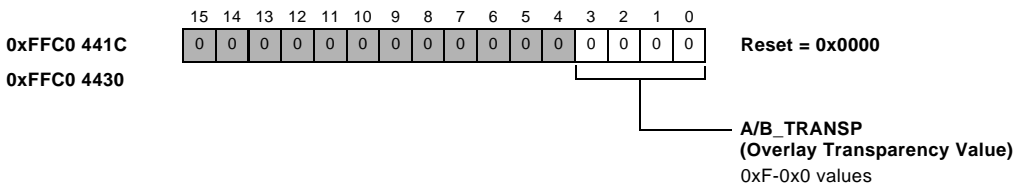


Figure 7-32. PIXC Transparency Value (PIXC_xTRANSP) Registers

PIXC Interrupt Status (PIXC_INTRSTAT) Register

The PIXC_INTRSTAT register (Figure 7-33) provides overlay interrupt configuration and status information.

PIXC Interrupt Status Register (PIXC_INTRSTAT)

Read/Write/W1C

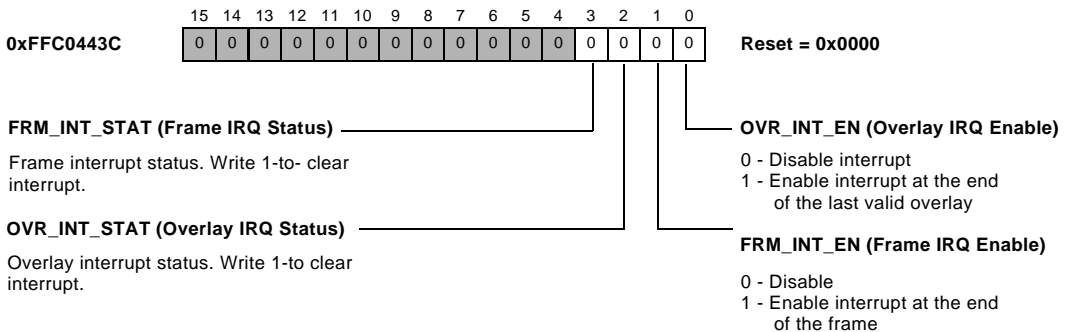


Figure 7-33. PIXC Interrupt Status (PIXC_INTRSTAT) Register

PIXC R/Y Conversion Coefficient (PIXC_RYCON) Register

The PIXC_RYCON register (Figure 7-34) provides the R/Y conversion coefficients in the color space conversion matrix.

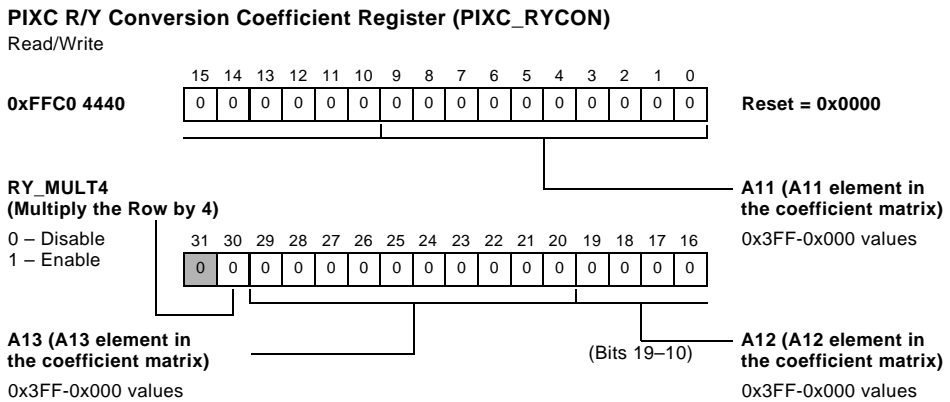


Figure 7-34. PIXC R/Y Conversion Coefficient (PIXC_RYCON) Register

PIXC G/U Conversion Coefficient (PIXC_GUCON) Register

The PIXC_GUCON register (Figure 7-35) provides the G/U conversion coefficients in the color space conversion matrix.

PIXC G/U Conversion Coefficient Register (PIXC_GUCON)

Read/Write

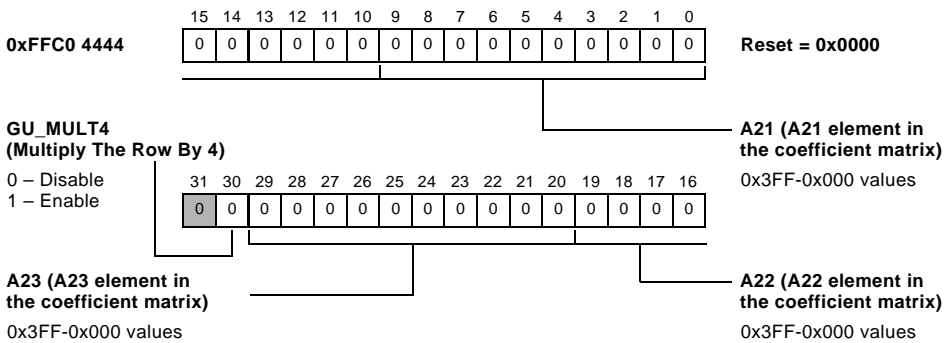


Figure 7-35. PIXC G/U Conversion Coefficient (PIXC_GUCON) Register

PIXC B/V Conversion Coefficient (PIXC_BVCON) Register

The PIXC_BVCON register (Figure 7-36) provides the B/V conversion coefficients in the color space conversion matrix.

PIXC B/V Conversion Coefficient Register (PIXC_BVCON)

Read/Write

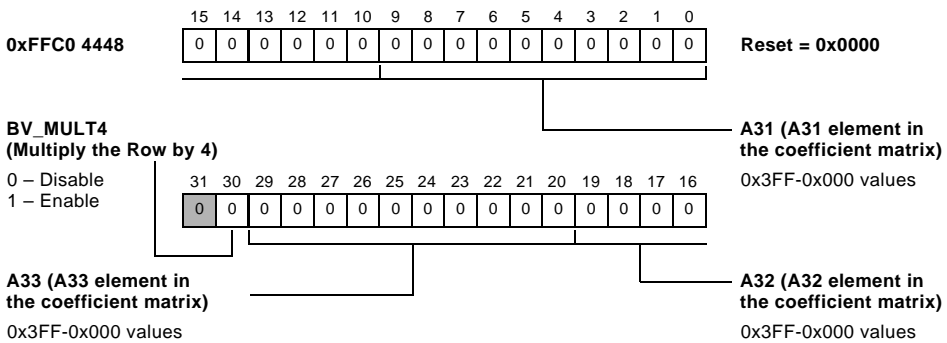


Figure 7-36. PIXC B/V Conversion Coefficient (PIXC_BVCON) Register

PIXC Color Conversion Bias (PIXC_CCBIAS) Register

The PIXC_CCBIAS register (Figure 7-37) provides the bias values in the color space conversion matrix.

PIXC Color Conversion Bias Register (PIXC_CCBIAS)

Read/Write

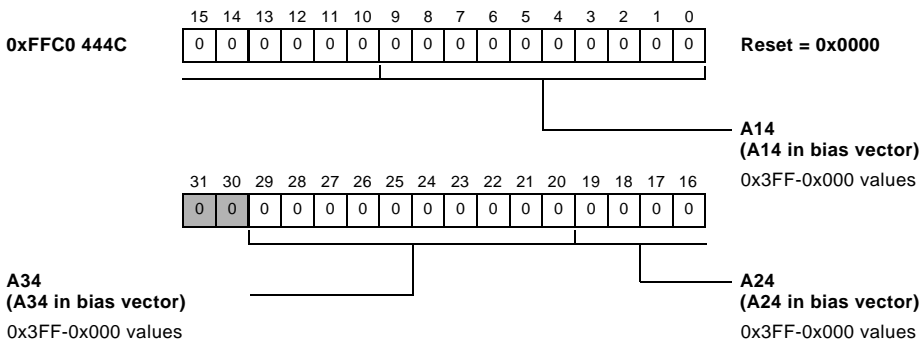


Figure 7-37. PIXC Color Conversion Bias (PIXC_CCBIAS) Register

PIXC Transparency Color Value (PIXC_TC) Register

The PIXC_TC register (Figure 7-38) provides the transparent color value.

PIXC Transparency Color Value Register (PIXC_TC)

Read/Write

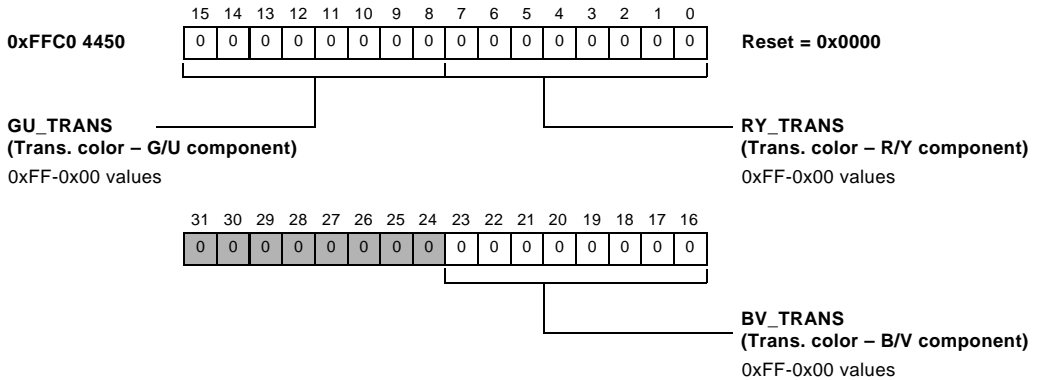


Figure 7-38. PIXC Transparency Color Value (PIXC_TC) Register

Programming Examples

Programming examples to be provided when code is available.

Programming Examples

8 HOST DMA PORT

This chapter describes the Host DMA port (HOSTDP) and includes the following sections:

- “Overview” on page 8-1
- “Interface Overview” on page 8-3
- “Description of Operation” on page 8-4
- “Programming Model” on page 8-22
- “Host DMA Port Registers” on page 8-24
- “Programming Examples” on page 8-31

Overview

The Host DMA port (HOSTDP) facilitates a host device external to the ADSP-BF54x Blackfin processor to be a direct memory access (DMA) master and transfer data back and forth. The host device always masters the transactions and the Blackfin processor is always a DMA slave device.



Pay particular attention to nomenclature involving the Host DMA port. The Host DMA port is sometimes abbreviated as HOSTDP. All register and pin names have a `HOST_` prefix. The HOSTDP is a

Overview

peripheral on the ADSP-BF54x processor, which is referred to as the slave processor or Blackfin slave. The host processor is also referred to as the host, master, external host, or external master.

When using one of the HOSTDP boot modes, the boot kernel does not disable the HOSTDP module or the associated DMA channels when the boot completes.

The HOSTDP is enabled through the peripheral access bus (PAB) interface. Once enabled, the DMA is controlled by an external host. The external host can then program the DMA to send/receive data to any valid internal or external memory location.

Features

The HOSTDP controller includes the following features:


- External master to configure DMA READ/WRITE data transfers and read port status
- Flexible asynchronous memory protocol for external interface
- 8/16-bit external data interface to host device
- Half-duplex operation
- Little/big endian data transfer
- Internal FIFO which holds sixteen 32-bit words
- Acknowledge mode allows flow control on host transactions
- Interrupt mode guarantees a burst of FIFO depth host transactions
- Ability to enable and disable data reads/writes
- DMA bandwidth control

Interface Overview

Table 8-1 defines the pins for the HOSTDP interface. The interface uses a flexible asynchronous memory interface, which can be gluelessly connected to a variety of host processors.

Table 8-1. HOSTDP External Pins

Pin	Description
Port D - HOST_DATA <15:0>	16-bit data port
PG5- $\overline{\text{HOST_CE}}$	Chip enable for the HOSTDP
PG7 - $\overline{\text{HOST_WR}}$	Write strobe
PG6- $\overline{\text{HOST_RD}}$	Read strobe
PH3 - HOST_ADDR	Address pin 0: data port access 1: configuration port access
PH4 - HOST_ACK (HRDY/FRDY)	Flow control pin: HRDY-acknowledge mode and FRDY- interrupt mode

 Due to the Blackfin processor's use of multiplexed pins, utilizing the Host DMA port can preclude the use of other peripherals. EPP2 is unavailable when using the HOSTDP, and EPP1 can be used in 8-bit mode if the HOSTDP is also in 8-bit mode. Refer to [“General-Purpose Ports” on page 9-1](#) for a complete description of the pin multiplexing scheme.

Description of Operation

The following sections describe the operation of the HOSTDP interface.

Architecture

The HOSTDP block diagram, shown in [Figure 8-1](#), illustrates the overall architecture of the HOSTDP.

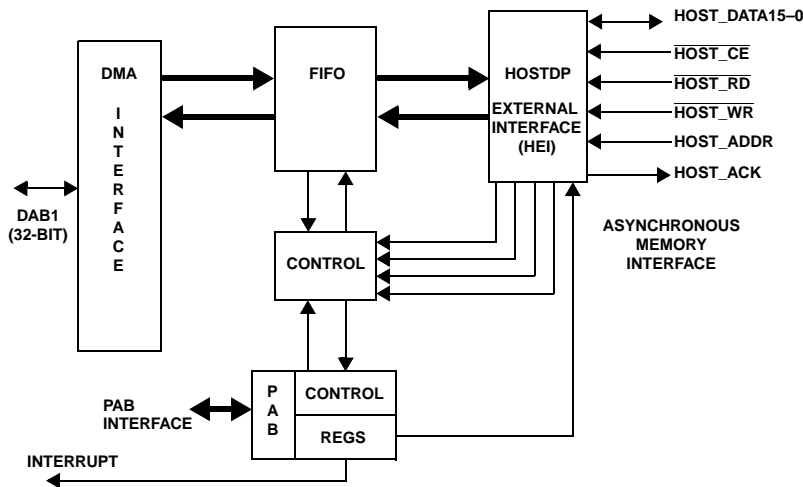


Figure 8-1. HOSTDP Block Diagram

The HOSTDP is enabled/disabled through PAB writes to the **HOST_CONTROL** register. Once enabled, the HOSTDP interfaces to the external world using asynchronous memory protocol and handshakes with the DMA controller internally using the DMA access bus (DAB). The

HOSTDP allows the external host to program the DMA to transfer data in either direction. The HOSTDP can be broken into five functional blocks, identified as follows:

- **Host External Interface (HEI):** This block interfaces to the external host and based on inputs from the host device initiates data or control message transfers.
- **PAB Interface:** The HOSTDP is programmed/queried for status by reads or writes to appropriate registers in this block through the PAB.
- **FIFO:** A dual-port FIFO is used for data transfers and can store up to sixteen 32-bit words.
- **Control:** The control block handles the HOSTDP's different states as well as the handshakes between the external host device and DMA interfaces.
- **DMA Interface:** This block is connected to the DAB and interacts with the DMA to transfer control messages and data between DMA and external host device.

Functional Description

The following sections describe the functional operation of the Host DMA port (HOSTDP).

HOSTDP Configuration

Before any data transfer can occur, the DMA engine must be configured by the host processor. Because the host is unaware of the internal state of the Host DMA port peripheral and its associated DMA activity, the host processor is required to check the `ALLOW_CNFG` bit in `HOST_STATUS` register before attempting configuration writes. Additionally, this status read sets

Description of Operation

some internal states inside the Host DMA port. Configuration requires seven 16-bit words to be written in the following order to the configuration port before host read data or host write data operations can occur:

- HOST_CONFIG
- START_ADDR.L
- START_ADDR.H
- XCOUNT
- XMODIFY
- YCOUNT
- YMODIFY

The only word different from the standard DMA described in the DMA chapter is the HOST_CONFIG word. Each bit is described there. Refer to [Figure 8-2](#) for a description.

HOSTDP Config Word (HOST_CONFIG)

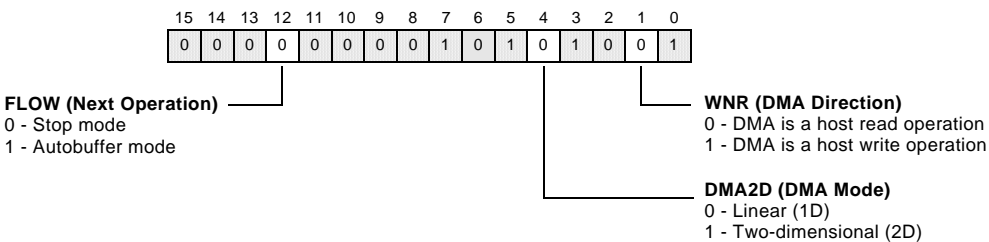


Figure 8-2. HOSTDP Configuration Word

Additional information for the `HOST_CONFIG` bits includes the following:

- **Host DMA Direction** (`WNR`):
If this bit is written high, DMA writes to memory (host write). If this bit is written low, DMA reads from memory (host read).
- **Host DMA Mode** (`DMA2D`):
If written low, it is linear one dimensional (1D) DMA. If written high, it is two-dimensional mode (2D).
- **FLOW** (`FLOW`):
When this bit is cleared, the DMA runs in `STOP` mode. When this bit is set, the DMA runs in `AUTOBUFFER` mode.

For information on how these words are used to configure the DMA, refer to [“Direct Memory Access” on page 5-1](#).

Before accessing the data port, the host processor must write all seven descriptor words. The `HOSTDP` module does not forward descriptors to the DMA channel until it has received all seven words. Similarly, the host processor is not permitted to provide new descriptor data before all data words of the former work unit are transferred. However, the host can truncate an initiated transfer using the `DMA_FINISH` control command. As always, `ALLOW_CNFG` in the `HOST_STATUS` register must be polled before writing a new configuration to the Host DMA port. Please see [“Control Commands Between the External Host and HOSTDP” on page 8-20](#) for additional information.

Additional latency is incurred when a host read data operation follows a host write data operation. Even though the configuration for the host read is complete, the DMA engine must first empty the FIFO for the host write operation and then change directions and start filling the FIFO for the host read data operation.

Description of Operation

HOSTDP Transactions

The HOSTDP is enabled by writing to the `HOSTDP_EN` bit of the `HOST_CONTROL` register. In order to disable the HOSTDP, the `HOSTDP_RST` bit must be asserted before clearing `HOSTDP_EN`. There are four types of host port transactions. Each type of access is controlled by the `HOST_ADDR` and whether the `HOST_RD` or `HOST_WR` signal is asserted.

When chip enable ($\overline{\text{HOST_CE}}$) is inactive, the HOSTDP stays idle. Modes listed in [Table 8-2](#) are only possible when $\overline{\text{HOST_CE}}$ is active.

Table 8-2. Types of Host Port Transactions

Address	$\overline{\text{HOST_RD}}$	$\overline{\text{HOST_WR}}$	$\overline{\text{HOST_CE}}$	Function
0x0	0	1	0	Host read data operation
0x0	1	0	0	Host write data operation
0x1	1	0	0	Host write configuration or control command
0x1	0	1	0	Host read <code>HOST_STATUS</code> register


Host Read Status

The Host DMA port is robust against on-the-fly changes of data direction. However, in acknowledge mode, it is encouraged not to initiate a new work unit with different data direction before the `FIFOEMPTY` bit in the `HOST_STATUS` register is cleared. This is to avoid excessive wait states inserted by `HRDY`.

The external host can read the `HOST_STATUS` register at any time. By performing this operation, the external host can query the status of the HOSTDP. Note that in 8-bit configurations, the host can only read the lower byte of the `HOST_STATUS` register. `HOST_STATUS` can also be read through a PAB access. When accessed through the PAB, all 16 bits of `HOST_STATUS` are always read. The contents of `HOST_STATUS` are detailed in [“Host DMA Port Registers” on page 8-24](#).

Host Read Data and Host Write Data Operations

After the HOSTDP is configured and enabled by way of PAB accesses and the DMA channel is configured through host write configuration accesses, data can be transferred.

 All DMAs between the HOSTDP FIFO and memory are 32-bit transactions. It is important to realize this when setting `XMODIFY` and `YMODIFY`. The amount of data moved between the host processor and the HOSTDP must be a multiple of the FIFO depth (sixteen 32-bit words). The user is required to set the `XCOUNT/YCOUNT` values and to also ensure that the correct number of host data reads or host data writes are performed.

A host write data operation is used to transfer data from the host to the slave processor. The host performs write transactions and the HOSTDP writes the data from these transactions into its FIFO. The DMA engine concurrently moves data from the HOSTDP's FIFO to the location in memory specified by the DMA configuration words.

A host read data operation is used to transfer data from the slave processor to the host. The DMA engine moves data from the specified location in the Blackfin processor slave's memory into the HOSTDP's FIFO. The host performs read accesses to read data out of this FIFO.

In the case of host writes, the host processor must “pad” the end of the transfer with dummy data to ensure this (for example, if the host wants 31 words it must send an extra dummy word to equal 32). In the case of host reads, dummy reads must be performed at the end and the host can then throw away the results. This is true in both interrupt mode and acknowledge mode.

Since all DMAs from the HOSTDP are 32-bits, data is packed into 32-bit words in the HOSTDP FIFO on host data write operations. Data (32-bit) in the FIFO is unpacked into 8-bit or 16-bit words (depending on the `HOSTDP_DATA_SIZE` setting in `HOST_CONTROL`) for transmission during host

Description of Operation

data read operations. Because all DMAs are 32-bits and the data bus is either 8-bits or 16-bits, the total of $XCOUNT * YCOUNT$ is $1/4$ (8-bit) or $1/2$ (16-bit) the number of data reads or writes the host processor performs.

HOSTDP Modes of Operation

There are two modes of flow control in the HOSTDP: acknowledge mode and interrupt mode. These two modes provide flow control between the host and the slave processor by way of a single hardware signal pin. This signal has different names depending upon the mode of operation. The flow control mode is configured by the slave processor when enabling the HOSTDP (see `HOST_CONTROL` register).


In acknowledge mode, the signal is called `HRDY` and is used to add wait states to a host transaction when the HOSTDP is not ready to transfer data. The `HRDY` signal is level-sensitive.

In interrupt mode, the signal is called `FRDY` and is used as an edge-triggered signal. This signal is connected to the host as an interrupt input. A falling edge on it signals the host that the HOSTDP is ready for a guaranteed FIFO depth number of back-to-back transactions. For host write operations, this occurs when the FIFO is empty. For host read operations, this occurs when the FIFO is full.

Acknowledge Mode

For host data write operations, `HRDY` negates when the FIFO is full, thereby inserting wait states. As soon as the DMA engine moves data out of the FIFO, `HRDY` asserts, indicating to the host the host data write operation is complete.

For host data read operations, $HRDY$ will negate when the FIFO is empty, thereby inserting wait states. As soon as the DMA engine moves data into the FIFO, $HRDY$ asserts indicating to the host the host data read operation is complete.

 The $HRDY$ signal must be pulled high by an external pull-up resistor by default at power-up/reset and when the HOSTDP is not enabled. $HRDY$ is only driven when $\overline{HOST_CE}$ is asserted low.

When the host is performing a host write configuration or $HOST_STATUS$ reads, $HRDY$ always remains asserted and no wait states are added.

Acknowledge Mode Timing Diagrams

This section gives further details on the HOSTDP timings for acknowledge mode. The host processor must follow these rules on every bus cycle independent of the nature of the access and the status of slave processor.

(It is assumed that the Blackfin slave processor has booted and the HOSTDP is functional.)

As discussed in the following section, $HRDY$ has an external pull-up register:

1. If $\overline{HOST_CE}$ is high, $HRDY$ is three-stated (not driven).
2. $HRDY$ is driven by the slave processor only when $\overline{HOST_CE}$ is asserted low by the external host device.
3. If $\overline{HOST_CE}$ as well as either $\overline{HOST_RD}$ or $\overline{HOST_WR}$ are asserted and $HOST_ADDR$ is high (configuration port access or status read), $HRDY$ remains driven high (READY).

Description of Operation

4. If $\overline{\text{HOST_CE}}$ as well as either $\overline{\text{HOST_RD}}$ or $\overline{\text{HOST_WR}}$ are asserted, and HOST_ADDR is low (data port access), one of two things happen:
 - a. If $\overline{\text{HOST_RD}}$ is asserted and the desired FIFO data can be transferred on the data bus pins within time T , HRDY remains driven high. If $\overline{\text{HOST_WR}}$ is asserted and if the data can be stored in the FIFO within time T , HRDY remains high.
 - b. If the desired FIFO data cannot be transferred on the data bus pins or stored in the FIFO within time T , HRDY is driven low quickly. At some later time after the FIFO data can be transferred, HRDY is driven high.

The two timing diagrams, shown in [Figure 8-3](#) and [Figure 8-4](#), are necessary to understand the function of HRDY .

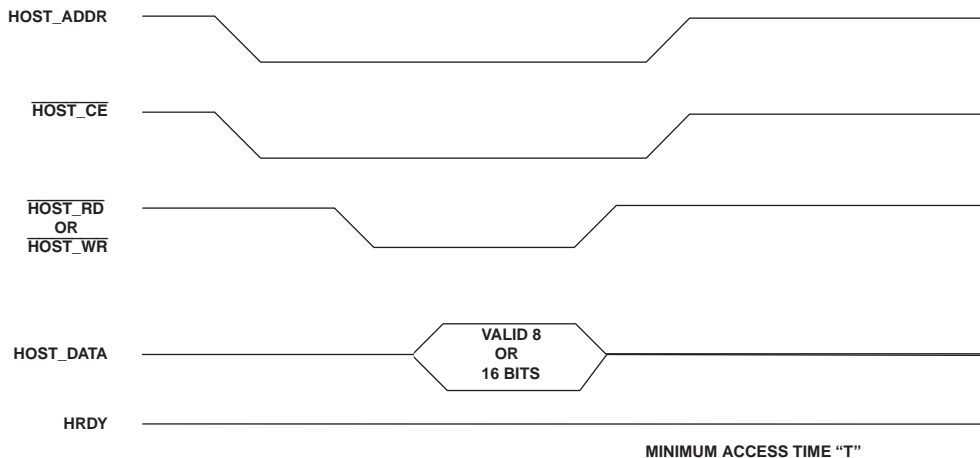


Figure 8-3. No Delay in Host Bus Cycle

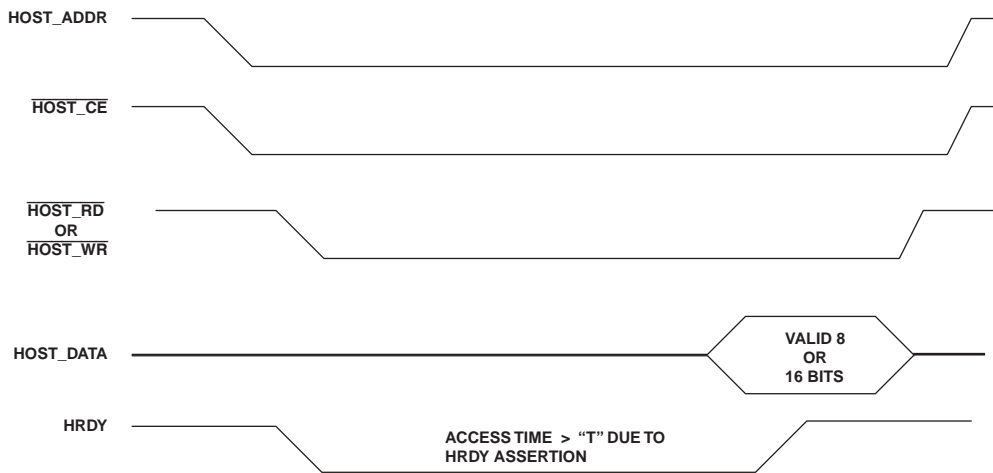


Figure 8-4. Delay in Host Bus Cycle Caused by HRDY

Host Bus Timeout

In acknowledge mode, an optional host bus timeout feature is implemented as a mechanism to alert the host when a programmed period of time has expired during a host read/write data transaction and the HOSTDP is still unable to complete the transaction with HRDY assertion. This condition can occur when the internal shared DMA bus has a lot of traffic from other peripherals on it. (This situation should never occur in a working system, but could occur if a mistake was made in software. An example is mistakenly disabling the DMA channel while the HOSTDP is attempting to transfer data.) An internal timer is started when $\overline{\text{HOST_CE}}$ and either $\overline{\text{HOST_RD}}$ or $\overline{\text{HOST_WR}}$ are asserted. The timer is reset whenever HRDY is asserted.

The feature can be enabled by the BT_EN bit in the HOST_CONTROL register. When enabled, the HOSTDP generates an interrupt when a pre-programmed timeout value set in the HOST_TIMEOUT register expires. In a typical application, the interrupt service routine toggles a GPIO pin which

Description of Operation

is connected to the host processor to alert it of this condition. Additionally, the interrupt service routine can perform writes to the `HOST_CONTROL` register to perform the following:

- Stop the DMA channel by clearing the `DMAEN` bit in the `DMAx_CONFIG` register
- Write the `HRDY_OVR` bit in the `HOST_CONTROL` register to assert the `HRDY` pin to allow the host bus cycles to continue while the host is being signaled of this condition by way of a GPIO pin
- Disable the `HOSTDP` by clearing the `HOSTDP_EN` bit in the `HOST_CONTROL` register

The actual timeout value can be programmed in the `HOSTDP_TOUT` register.

Because it is important for the host to be aware that a timeout condition occurred, it is required that the host processor read the `HOST_STATUS` register and check the `HOSTDP_TOUT` bit. The ADSP-BF54x slave processor reads the actual bit, allowing it to take the timeout interrupt, and write-one-to-clear the `HOSTDP_TOUT` bit. The host processor reads a special shadow version of this bit which remains set until the host has read it or a hard reset occurs.

Interrupt Mode

The `FRDY` signal acts as an edge-sensitive (high-to-low transition) signal to provide an interrupt to the external host to indicate when data transfer can proceed. The interrupt provided by the slave processor to the external host device by way of the `FRDY` signal is used to indicate the status of the Host DMA port's FIFO. Host data read and host data write accesses are described next. The host device always masters the transactions and the Blackfin processor is always a DMA slave device.

In interrupt mode, the `FRDY` signal always is driven by the slave processor and does not require an external pull-up resistor.

For host write operations, the `FRDY` signal transitions from high to low whenever the FIFO is empty, causing an interrupt to the host to tell it to write to `HOSTDP`. The host can then perform a buffer depth number of write cycles to fill the FIFO. During these writes, the `FRDY` signal transitions high again, but this is ignored by the host. After the FIFO's contents have been moved to memory by the DMA engine, the FIFO becomes empty. At this time, `FRDY` will once again transition from high to low to interrupt the host to do another buffer depth number of write cycles to fill the FIFO. This process continues until the configured number of words have been transferred.

For host read operations, the `FRDY` signal transitions from high to low whenever the FIFO is full, causing an interrupt to the host to tell it to read from the `HOSTDP`. The host can then perform a buffer depth number of read cycles to empty the FIFO. During these reads, the `FRDY` signal transitions high again, but this is ignored by the host. The DMA engine fills the FIFO from data in memory. Once the FIFO becomes full again, the `FRDY` signal once again transitions from high to low to interrupt the host to do another buffer depth number of write cycles to fill the FIFO. This process continues until the configured number of words have been transferred.

In interrupt mode, the `FRDY` signal always reflects the status of the FIFO. For host configuration writes or host reads of `HOST_STATUS`, accesses always meets the minimum cycle time T and the `FRDY` signal is not used for flow control of these accesses.

Description of Operation

Figure 8-5 shows the timing of the interrupt mode transactions. The total number of words in the transfer are divided into blocks that contain a FIFO depth's number of words. These blocks are transferred whenever a high-to-low transition occurs on the `FRDY` signal.

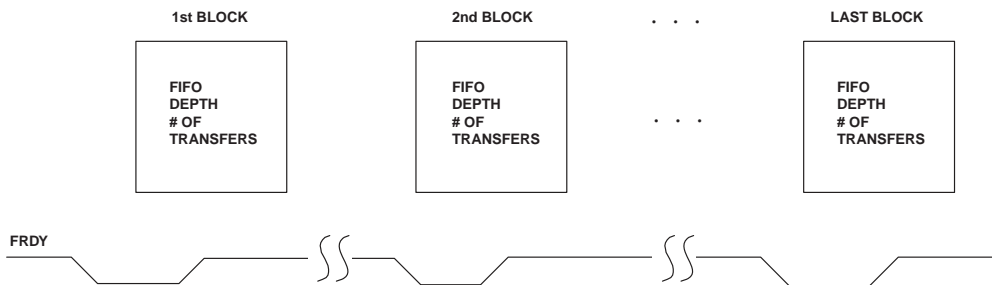


Figure 8-5. Interrupt Mode Bus Cycles

DMA STOP Mode and AUTOBUFFER Mode

The `FLOW` bit in `HOST_CONFIG` word controls whether the DMA channel runs in stop mode or autobuffer mode.

In stop mode, the DMA performs a block transfer once as programmed by the `HOST_CONFIG`, `XCOUNT/YCOUNT`, `XMODIFY/YMODIFY`, `START_ADDR.L/H` registers. To perform another block transfer requires the host to reconfigure these parameters. For stop mode, the interrupt service routine is required to set the `DMA_CMPLT` bit in the `HOST_STATUS` register. This prepares the `HOSTDTP` for the next transfer. The host is not required to poll the `DMA_CMPLT` bit before starting a new work unit.

In autobuffer mode, the DMA performs continuous block transfers based on the parameters programmed by the `HOST_CONFIG`, `XCOUNT/YCOUNT`, `XMODIFY/YMODIFY`, `START_ADDR.L/H` registers. Once the number of words specified by `XCOUNT/YCOUNT` are transferred, the DMA engine sets its address pointer back to `START_ADDR.L/H` and performs another block transfer. For autobuffer mode, the interrupt service routine should only

set the `DMA_CMPLT` bit in the `HOST_STATUS` register when it wishes to complete the transfers. After this bit is set, the `HOSTDP` block expects to be reprogrammed with a new set of DMA register values.

Bus Widths and Endian Order

The `HOSTDP` can be programmed to be 16-bits wide or 8-bits wide. Additionally, the byte order can be programmed as little endian or big endian. All ensuing data and configuration transactions with the host occur in the programmed endianness setting.

For 16-bit transfers, shown in [Figure 8-6](#), the upper and lower bytes are based on the big/little endian setting. When set to little endian, the order of the bytes on the `HOST_DATA[15:0]` bus is unchanged. For big endian, the upper and lower bytes of `HOST_DATA[15:0]` are swapped before being stored internally.

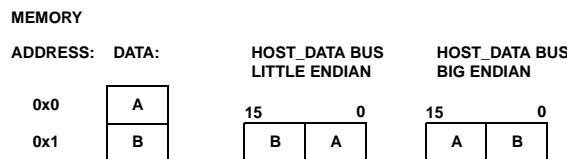


Figure 8-6. 16-Bit Transfer Byte Order

Description of Operation

For 8-bit transfers, the order in which the bytes are sent are based on the bit/little endian setting as shown in [Figure 8-7](#). Consider a 16-bit word stored in internal memory:

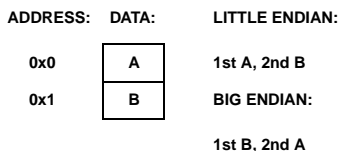



Figure 8-7. 8-Bit Transfer Byte Order

Access Control

Configurations only occur when they are allowed. The `ALLOW_CNFG` bit does not go low after configuration words are written if the access type is disallowed. In the case of a disallowed configuration, the configuration words are not driven on the DAB bus, and DMA controller does not get programmed. There is no NACK provided to the host in the event of a disallowed configuration.

By default, the HOSTDP module prohibits the external host from performing host data read and writes. Blackfin software is required to enable host reads or writes. Host data reads and writes are enabled or disabled separately by the `EHR` and `EHW` bits in the `HOST_CONTROL` register. Once enabled, the host can perform read or write transactions. Writes to the configuration port, control commands and status reads are permitted regardless of the `EHR` and `EHW` settings. It is very important that the `EHR/EHW` bits are written once before ever allowing configuration from the Host and then not changed later.

 For more information, see the memory configuration discussion in [“Security” on page 15-1](#).

In acknowledge mode, if the transactions are disabled, host writes are still allowed on the bus, but the actual write data is ignored. Similarly, host reads still occur on the bus, but the data returned is indeterminate.

In interrupt mode, transitions on `FRDY` never occur.

i The host cannot interrogate the HOSTDP to see whether only read or write access is granted. Therefore, keep the `EHR` and `EHW` settings global without altering them.

Improving HOSTDP DMA Bus Bandwidth

Since the HOSTDP can be configured as a 16-bit wide parallel interface, data can move into and out of the peripheral quickly as compared to other serial peripherals on the chip. A FIFO is used to buffer this data and internal DMA bus requests are made judiciously to minimize the amount of DMA bandwidth that is used on the DMA bus. DAB bus arbitration overhead and direction change penalties are minimized. This is the default behavior (`BDR=1` in `HOST_CONTROL`) and the Host DMA port generally follows this behavior, shown in [Table 8-3](#), for receive (host write) operations:

Table 8-3. Host Write Operations

32-Bit Words in FIFO	DMA Request Freq (SCLK cycles)	Bursts per DMA Request
1 – 4	24	Up to 4
5 – 8	16	4
9 – 12	8	4
>12	2	0

For example, if there are ten words written into the FIFO by the host processor, on the eighth `SCLK` cycle, DMA is requested. Once the DAB approves the request, it transfers four words. Assuming the host processor does not write any new words to the FIFO, the HOSTDP again requests

Description of Operation

DMA 16 cycles later and another four words are transferred. Twenty-four SCLK cycles later, the remaining two words are transferred. Note that words stored in the FIFO are 32 bits.

For transmit (host read) operation, the values look similar. Refer to [Table 8-4](#).

Table 8-4. Host Read Operation

32-bit Words in FIFO	DMA Request Freq (SCLK cycles)	Bursts per DMA Request
0 – 4	2	0
5 – 8	8	4
9 – 12	16	4
>12	24	Up to 4

This default behavior can be overridden by clearing the burst DMA requests (BDR) bit in the `HOST_CONTROL` register. This allows the HOSTDP to perform internal DMA bus requests whenever there is a single word of data in the FIFO for host writes and at least one empty slot for host reads. In this case, DMA bus requests are made more often. This allows higher throughput through the HOSTDP at the expense of the other peripherals on the chip.

Control Commands Between the External Host and HOSTDP

Control commands can be sent from the host to the HOSTDP by writing to the configuration port with bits 3 and 2 of the data high. When the Host DMA port is waiting for configuration, a control command cannot be sent because it will be misinterpreted as a configuration write. After configuration is finished, control commands can be issued at any time. If the host is unsure of whether configuration is pending, it needs to read the `HOST_STATUS` register to check.

The commands that are supported are shown in [Table 8-5](#).

Table 8-5. Control Commands

HOST_DATA[7:0]	Command
8'b000111xx	Host IRQ
8'b001011xx	DMA finish
8'b001111xx to 8'b111111xx	Ignored

The host IRQ command provides a mechanism for the host to interrupt the HOSTDTP. When the host writes a host IRQ command to the configuration port, the `HIRQ` bit in the `HOST_STATUS` register is set and a HOSTDTP status interrupt is signaled.

The handshake bit (`HSBK`) in `HOST_STATUS` can be set or cleared anytime by the slave processor. This bit can be used as a flag which the host can read. In an application, the host might interrupt with the host IRQ command requesting information. The interrupt service routine could then set or clear the `HSBK` bit. The host could then read the status register and test for the value of the `HSBK` bit.

The DMA finish command performs all the same functions as the HOSTDTP reset (`HOSTDTP_RST`) bit in `HOST_CONTROL`, except modifying the `HOST_STATUS` register contents. In addition, it stops any DMA activity. The DMA FINISH command may not complete right away, instead it completes only after the DAB state machine has moved to a particular idle state.

There are additional restrictions on when a DMA Finish command may be sent by the host processor. Refer to "DMA Control Commands" on page 5-39 of the "Direct Memory Access" chapter for more information.

When the HOSTDTP module receives a `FINISH` command from the host during a write operation, the DMA channel's FIFO is still drained gracefully and requests a DMA completion interrupt. However, the

Programming Model

HOSTDP's FIFO is flushed immediately. To avoid loss of data, the host may want to wait until the `FIFOEMPTY` bit in `HOST_STATUS` is asserted before issuing the finish command.

Programming Model

BF54x Slave

Figure 8-8 shows how to enable the Host DMA port. It shows how to properly set up interrupt service routines for both host read and write which clear the interrupts and prepare the HOSTDP for to be configured by the host again.

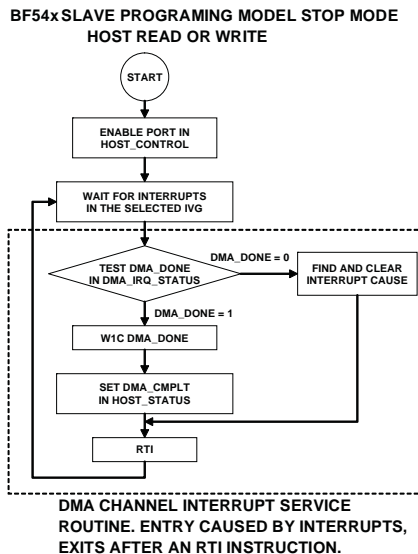


Figure 8-8. Enable the Host DMA Port

Host Processor

Figure 8-9 and Figure 8-10 demonstrate how to program a host processor to send a configuration to the ADSP-BF54x slave. They also show when to send or receive data in both acknowledge and interrupt modes.

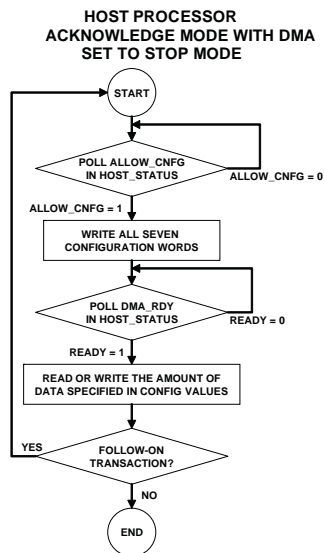


Figure 8-9. Program Host Processor, Part 1

Host DMA Port Registers

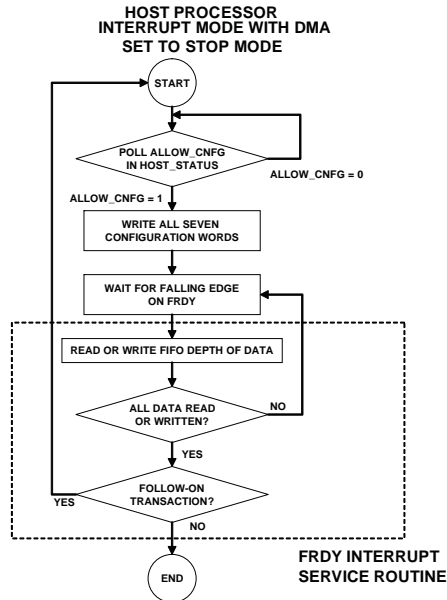


Figure 8-10. Program Host Processor, Part 2

Host DMA Port Registers

Descriptions and bit diagrams for each of the MMRs discussed in this chapter are provided in the following sections:

- “Host DMA Port Control (HOST_CONTROL) Register” on page 8-25
- “Host DMA Port Status (HOST_STATUS) Register” on page 8-27
- “HOSTDP Timeout (HOST_TIMEOUT) Register” on page 8-29

Host DMA Port Control (HOST_CONTROL) Register

The HOSTDP control register (HOST_CONTROL), shown in [Figure 8-11](#), is used to enable the HOSTDP module as well as to establish transfer modes of operation.

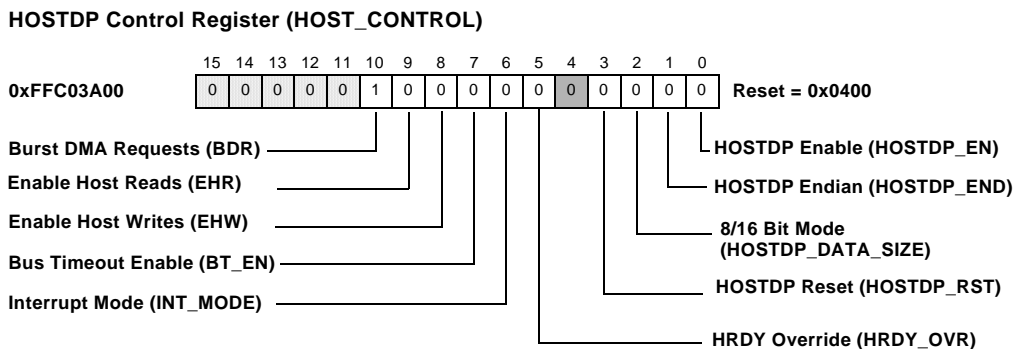


Figure 8-11. HOSTDP Control (HOST_CONTROL) Register

Additional information for the HOST_CONTROL register bits include:

- HOSTDP Enable (HOSTDP_EN):**
 This bit enables the HOSTDP interface. This bit controls the muxing of the shared HOSTDP and PPI pins. Before disabling the HOSTDP, always reset it first.
- Little/Big Endian (HOSTDP_END):**
 When set, this bit swaps the lower and upper byte of data when reading or writing to HOSTDP FIFO. A value of 0 represents little endian and a value of 1 represents big endian.

Host DMA Port Registers

- **8/16-Bit Host Data Transfer** (HOSTDP_DATA_SIZE):
This bit sets the HOSTDP external data transfer width. This bit, along with HOSTDP_EN, controls the muxing of the HOSTDP data pins and the EPPI pins. A value of 0 is 8-bit data and a value of 1 is 16-bit
- **HOSTDP Reset** (HOSTDP_RST):
This is a soft reset which does not affect the contents of HOST_CONTROL. Programming this bit causes the FIFO to flush, turns off the DMA channel, and returns the HOSTDP to a state where it is waiting for configuration. It also causes HOST_STATUS to clear to the same value as a hard reset with the exception of the BTE bit, which is always the same as BT_EN in HOST_CTL. Host DMA port reset will not complete right away, instead it completes only after the DAB state machine has moved to a particular idle state. This bit is always read as a binary 0.
- **Host Ready Override** (HRDY_OVR):
Setting this bit high forces HRDY high. If HRDY_OVR bit is written high, HRDY is driven high for all remaining FIFO transfers. Also, the ALLOW_CNFG bit is driven low to prevent accidental configurations.
- **Interrupt Mode** (INT_MODE):
This bit, when set, is used to select interrupt mode. When cleared, it selects acknowledge Mode. A value of 0 selects acknowledge mode and a value of 1 selects interrupt mode.
- **Bus Timeout Enable** (BT_EN):
This bit, when set, enables HOSTDP's interrupt to occur when a current host transaction has not finished before a programmed timeout value occurs.
- **Enable HOSTDP Write** (EHW):
This bit, when set, enables HOSTDP's writes to occur. If disabled, host writes appear to occur on the pins, but the actual write data is ignored.

- **Enable HOSTDP Read (EHR):**
This bit, when set, enables HOSTDP’s reads to occur. If disabled, host reads return zero data.
- **Burst DMA Requests (BDR):**
When set, as by default, the HOSTDP’s module groups multiple data words and requests DMA bursts to the DAB bus. When cleared, every individual data word requests its separate DMA transfer.

Host DMA Port Status (HOST_STATUS) Register

The HOSTDP status register (HOST_STATUS), shown in Figure 8-12, holds the key status information of the HOSTDP. Bits in this register are read by the external host to query status of transaction. This register can also be read and written through PAB. Note the differences in how to write and clear bits as well as the many bits which are read-only.

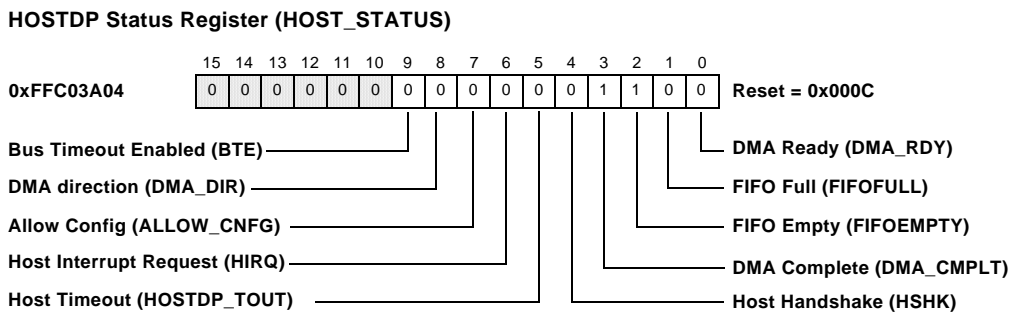



Figure 8-12. HOSTDP Status (HOST_STATUS) Register

Host DMA Port Registers

Additional information for the `HOST_STATUS` register bits include:

- **DMA Ready** (`DMA_RDY`) - read-only:
This bit is set one cycle after the last control word (`YMODIFY`) is written to the DMA. The bit is cleared when the `COMPLETE` bit is set by software.
- **FIFO Full** (`FIFOFULL`) - read-only:
This bit is set when the HOSTDP FIFO is full.
- **FIFO Empty** (`FIFOEMPTY`) - read-only:
This bit is set when the HOSTDP FIFO is empty.
- **DMA Complete** (`DMA_CMPLT`) - write-1-to-set:
This bit must be set by software in the interrupt service routine called when the DMA operation is completed. This bit is cleared after the last control word (`YMODIFY`) is written to the DMA controller.
- **HOSTDP Handshake** (`HSHK`) - read/write:
This bit is set and cleared by software and functions as a general-purpose handshake bit. Often it is used to indicate an error to the host device. This bit does not control HOSTDP hardware and is cleared by the `HOSTDP_RST` bit.
- **HOSTDP Timeout** (`HOSTDP_TOUT`) - write-1-to-clear:
This bit is set when the HOSTDP time-out occurs. When set, it requests a HOSTDP status interrupt. The interrupt service routine (ISR) must write this bit to one to clear it.
- **HOSTDP Interrupt Request** (`HIRQ`) - write-1-to-clear:
This bit is set when the host writes a HOSTDP IRQ control command to the configuration port. When set, this bit requests a HOSTDP status interrupt. The interrupt service routine (ISR) must write this bit to one to clear it.

- **Allow Configurations** (`ALLOW_CNFG`) - read-only:
The host processor is required to poll this bit to see when the Host DMA port is enabled and configuration writes are allowed. This bit is cleared when the last configuration word (`YMODIFY`) is written by the host. The bit is set again when the descriptor is completely passed to the DMA channel.
- **DMA Direction** (`DMA_DIR`) - read-only:
This bit is set to 0 when DMA is set for read and set to 1 for DMA writes. It reflects the `WNR` bit in the `DMA_CONFIG` word. If a former work unit was active, the bit does not update until the `DMA_CPLT` bit is set by software.
- **Bus Timeout Enabled** (`BTE`) - read-only:
This bit is just a copy of the `BT_EN` bit in the `HOST_CONTROL` register. The host can read this bit to determine if software has enabled the bus timeout feature.

 This bit must be set by the interrupt service routine software which is called when the DMA is finished.

HOSTDP Timeout (`HOST_TIMEOUT`) Register

The HOSTDP timeout feature is previously described in “[Acknowledge Mode](#)” on page 8-10.

HOSTDP Timeout Register (`HOST_TIMEOUT`)

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
<code>0xFFC03A08</code>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Reset = <code>0x0000</code>

Figure 8-13. HOSTDP Timeout (`HOST_TIMEOUT`) Register

Host DMA Port Registers

The HOSTDP timeout register (HOST_TIMEOUT), shown in [Figure 8-13](#), holds the timeout value. A timer is loaded with this value when a host transaction is started. If HOSTDP does not respond with HRDY within the programmed amount of time, the TIMEOUT bit in the HOST_STATUS register is set and an interrupt is generated. This feature takes effect only when the BT_EN bit in the HOST_CONTROL register is set to 1.

The length of the timeout generated by this register is governed by the following equation:

$$\text{timeout} = (2^{16} * \text{HOST_TIMEOUT}) / (\text{sclk_freq})$$

For example, using an SCLK frequency of 133 MHz and HOST_TIMEOUT = 0x7ED, the timeout period is approximately one second.

Programming Examples

Listing 8-1. Enable 8-Bit HOSTDP data in pin MUXing

```

/* Enable 8-bit HOSTDP data in pin MUXing */

P5.H = hi(PORTD_FER);
P5.L = lo(PORTD_FER);

R5.L = PD15 | PD14 | PD13 | PD12 | PD11 | PD10 | PD9 | PD8 | nPD7
      | nPD6 | nPD5 | nPD4 | nPD3 | nPD2 | nPD1 | nPD0;

w[P5] = R5.L;

P5.H=hi(PORTD_MUX);
P5.L=lo(PORTD_MUX);

R5.H=hi(MUX(1,1,1,1,1,1,1,1,0,0,0,0,0,0,0));
R5.L=lo(MUX(1,1,1,1,1,1,1,1,0,0,0,0,0,0,0));

[P5] = R5;

/* Enable 16-bit HOSTDP data in pin MUXing */

P5.H = hi(PORTD_FER);
P5.L = lo(PORTD_FER);

R5.L = PD15 | PD14 | PD13 | PD12 | PD11 | PD10 | PD9 | PD8 | PD7

```

Programming Examples

```
| PD6 | PD5 | PD4 | PD3 | PD2 | PD1 | PD0;
```

```
w[P5] = R5.L;
```

```
P5.H=hi(PORTD_MUX);
```

```
P5.L=lo(PORTD_MUX);
```

```
R5.H=hi(MUX(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1));
```

```
R5.L=lo(MUX(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1));
```

```
[P5] = R5;
```

```
/* Enable HOSTDP Control Signals in pin MUXing */
```

```
P5.H = hi(PORTG_FER);
```

```
P5.L = lo(PORTG_FER);
```

```
R5.L = nPG15 | nPG14 | nPG13 | nPG12 | PG11 | nPG10 | nPG9 | nPG8  
| PG7 | PG6 | PG5 | nPG4 | nPG3 | nPG2 | nPG1 | nPG0;
```

```
w[P5] = R5.L;
```

```
P5.H=hi(PORTH_MUX);
```

```
P5.L=lo(PORTH_MUX);
```

```
R5.H=hi(MUX(0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0));
```

```
R5.L=lo(MUX(0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0));
```

```
[P5] = R5;
```

```

P5.H = hi(PORTH_FER);

P5.L = lo(PORTH_FER);

R5.L = nPH15 | nPH14 | nPH13 | nPH12 | nPH11 | nPH10 | nPH9 | nPH8
| nPH7 | nPH6 | nPH5 | PH4 | PH3 | nPH2 | nPH1 | nPH0;

w[P5] = R5.L;

P5.H=hi(PORTH_MUX);

P5.L=lo(PORTH_MUX);

R5.H=hi(MUX(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0));

R5.L=lo(MUX(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0));

[P5] = R5;

/* Enable 16-bit HOSTDP */

P5.H = hi(HOST_CONTROL);

P5.L = lo(HOST_CONTROL);

R5 = HOSTDP_EN /* HOSTDP Enable */
| nHOSTDP_END /* Little endian transfers */
| HOSTDP_DATA_SIZE /* 16-bit Data Size */
| nINT_MODE /* Acknowledge Mode */
| nBT_EN /* Bus timeout disabled */

```

Programming Examples

```
| EHW /* Enable Host Writes */  
| EHR /* Enable Host Reads */  
| BDR (z); /* Burst DMA Requests On */
```

9 GENERAL-PURPOSE PORTS

This chapter describes general-purpose ports, pin multiplexing, general-purpose input/output (GPIO) functionality, and pin interrupts. This chapter includes the following sections:

- [“Overview” on page 9-1](#)
- [“Module Overview” on page 9-3](#)
- [“Pin Multiplexing Scheme” on page 9-4](#)
- [“GPIO Functionality” on page 9-21](#)
- [“Pin Interrupts” on page 9-23](#)
- [“Programming Model” on page 9-26](#)
- [“Port Registers” on page 9-30](#)
- [“Programming Examples” on page 9-60](#)

Overview

The general-purpose ports cover three jobs:

- Pin multiplexing scheme
- GPIO functionality
- Pin interrupts

This chapter characterizes each of the three topics in detail.

Features

The peripheral pins are functionally organized into ten general-purpose ports designated port A through port J. These ports feature:

- Up to 152 general-purpose I/O (GPIO) pins
- Input mode, output mode, and open-drain mode of GPIO operation
- Port multiplexing controlled by individual pin-per-pin base
- Identical port multiplexing scheme on all ADSP-BF54x Blackfin processor family derivatives
- No glue hardware required for unused pins
- Four interrupt channels dedicated to pin interrupts
- All port pins provide interrupt functionality
- Byte-wide pin-to-interrupt assignment

Module Overview

A simplified illustration of the GPIO and pin interrupt signal flow is shown in [Figure 9-1](#).

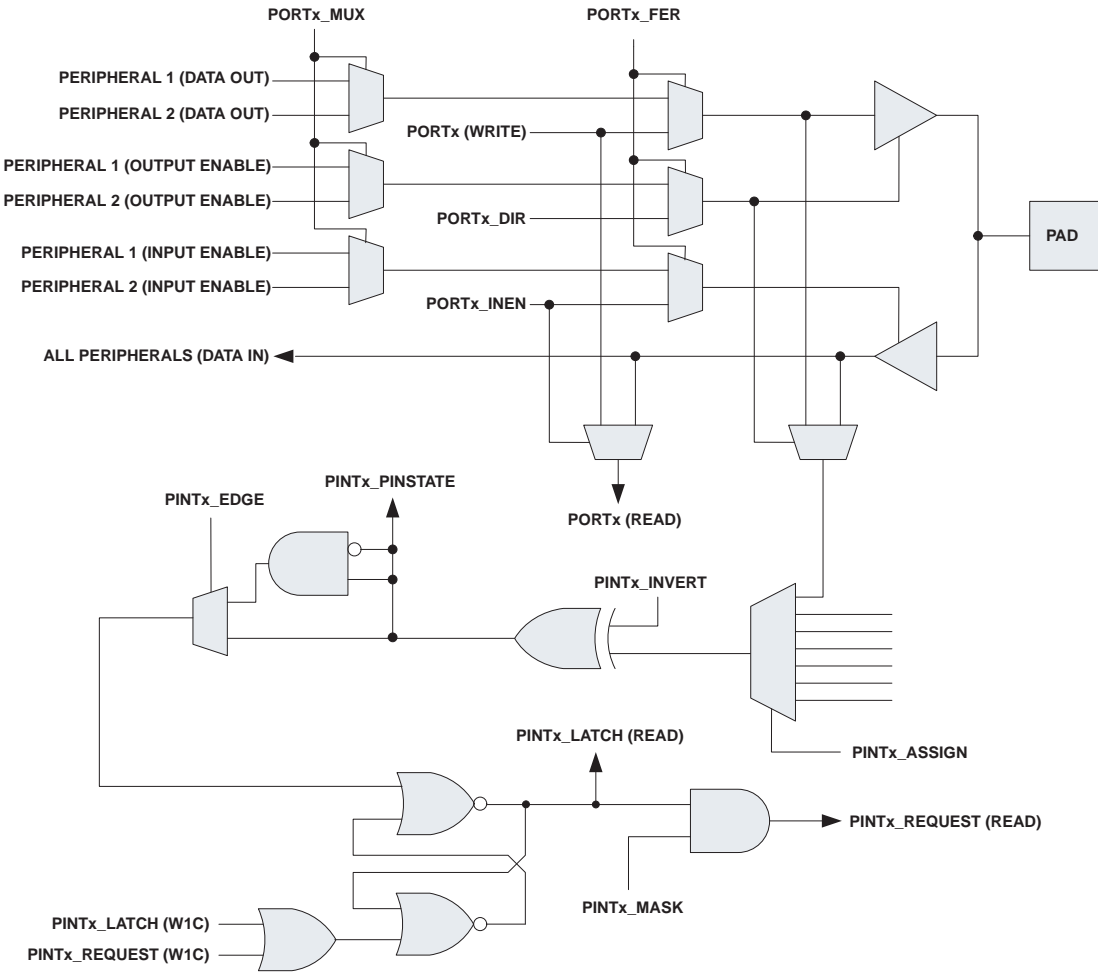


Figure 9-1. Simplified GPIO and Pin Interrupt Signal Flow

Pin Multiplexing Scheme

External Interfaces

The pin multiplexing hardware can be seen as a layer between the on-chip peripherals and the pads of the silicon. All pins grouped into the ports “port A” to “port J” are controlled by this unit.

Internal Interfaces

All MMR registers of the pin multiplexing, GPIO and pin interrupt control blocks can be accessed through the PAB bus. There is no DMA support. Every one of the four pin interrupt modules has its own and dedicated interrupt request output signal that connects directly to the SIC controller, as shown in [Figure 9-2 on page 9-23](#).

Pin Multiplexing Scheme

ADSP-BF54x Blackfin processors feature a rich set of on-chip peripherals. Each set of peripherals has a combination of input and output signals associated with them. In total, these are many more signals than pins available on the processors. Therefore, a powerful pin multiplexing scheme provides best flexibility to external application space.

Table 9-1 shows all peripheral signals that are accessible off the chip through the general-purpose ports. The individual members of the ADSP-BF54x Blackfin processor family do not feature all the listed peripherals at the same time. Note that some signals are optional and are not necessarily required in all operating modes.

Table 9-1. General-Purpose and Special Function Signals

Module	Signals	On Ports	ADSP-BF549	ADSP-BF548	ADSP-BF547	ADSP-BF544	ADSP-BF542
EBIU (async)	Address (22)	H, I					
	Bus Handshake (3)	J	x	x	x	x	x
	Clock (1)	I					
	Ready (1)	J					
NAND Flash Controller	Control (2)	J	x	x	x	x	x
ATAPI	Control (8)	J	x	x	x	-	x
	Reset (1)	H					
HostDMA Port (HOSTDP)	Data (16)	D					
	Control (3)	B, G, H	x	x	x	x	-
	Address (1)	H					
	Acknowledge (1)	H					
SD/SDIO Controller	Data (4)	C					
	Clock (1)	C	x	x	x	-	x
	Command (1)	C					
EPPIO	Data (24)	D, F					
	Clock (1)	G	x	x	x	x	-
	Frame Sync (3)	G, H					

Pin Multiplexing Scheme

Table 9-1. General-Purpose and Special Function Signals (Cont'd)

Module	Signals	On Ports	ADSP-BF549	ADSP-BF548	ADSP-BF547	ADSP-BF544	ADSP-BF542
EPPI1	Data (16)	D					
	Clock (1)	E	x	x	x	x	x
	Frame Sync (3)	E, H					
EPPI2	Data (8)	D					
	Clock (1)	G	x	x	x	x	x
	Frame Sync (3)	G, H					
SPORT0	Data (4)	C					
	Clock (2)	C	x	x	x	-	-
	Frame Sync (2)	C					
SPORT1	Data (4)	D					
	Clock (2)	D	x	x	x	x	x
	Frame Sync (2)	D					
SPORT2	Data (4)	A					
	Clock (2)	A	x	x	x	x	x
	Frame Sync (2)	A					
SPORT3	Data (4)	A					
	Clock (2)	A	x	x	x	x	x
	Frame Sync (2)	A					
SPI0	Data (2)	E					
	Clock (1)	E					
	Slave Select (1)	E	x	x	x	x	x
	Slave Enable (3)	E					

Table 9-1. General-Purpose and Special Function Signals (Cont'd)

Module	Signals	On Ports	ADSP-BF549	ADSP-BF548	ADSP-BF547	ADSP-BF544	ADSP-BF542
SPI1	Data (2)	G	x	x	x	x	x
	Clock (1)	G					
	Slave Select (1)	G					
	Slave Enable (3)	G					
SPI2	Data (2)	B	-x	-x	x	-	-
	Clock (1)	B					
	Slave Select (1)	B					
	Slave Enable (3)	B					
UART0	Data (2)	E	x	x	x	x	x
UART1	Data (2)	H	x	x	x	x	x
	Control (2)	E					
UART2	Data (2)	B	x	x	x	-	-
UART3	Data (2)	B	x	x	x	x	x
	Control (2)	B					
High Speed USB OTG			x	x	x	-	x
CAN0 ¹	Data (2)	G	x	x	-	x	x
CAN1 ¹	Data (2)	G	-	x	-	x	-
TWI0	Data (1)	E	x	x	x	x	x
	Clock (1)	E					
TWI1	Data (1)	B	x	x	x	x	-
	Clock (1)	B					
Timer 0-7	PWM/Capture/Clock (8)	A, B	x	x	x	x	x
	Alternate Clock Input (8)	A					
	Alternate Capture Input (7)	A, B, E, G, H					

Pin Multiplexing Scheme

Table 9-1. General-Purpose and Special Function Signals (Cont'd)

Module	Signals	On Ports	ADSP-BF549	ADSP-BF548	ADSP-BF547	ADSP-BF544	ADSP-BF542
Timer 8-10	PWM/Capture/Clock (3)	H					
	Alternate Clock Input (3)	H	x	x	x	x	-
	Alternate Capture Input (3)	H					
Up/ Down Counter	Up / Dir (1)	H					
	Down / Gate (1)	H	x	x	x	x	x
	Zero Marker (1)	G					
KEYPAD	Rows (8)	D, E	x	-x	x	-	x
	Columns (8)	D, E					
MXVR	Data (2)	H					
	Clock (2)	C	x	-	-	-	-
	Control (2)	G, H					
GPIOs	GPIOs (152)	A-J	x	x	x	x	x

1 Automotive only.

Read from Page 0x05 of the on-chip OTP memory when determining whether a module is available on a respective ADSP-BF54x Blackfin processor. For details, see [“System Reset and Booting” on page 17-1](#).

The peripheral pins of the ADSP-BF54x Blackfin processors are functionally organized into ten general-purpose ports which are designated Port A through port J. Most ports consist of 16 pins; a few have fewer. By default, all port pins are configured for GPIO operation after reset. In total, there are 152 GPIO-capable pins. Pin interrupt functionality is covered by a separate functional block.

The individual ports are discussed in the following sections.

Port A

Port A consists of 16 pins, referred to as PA0 to PA15, as shown in [Table 9-2](#). Besides the 16 GPIOs, this port homes all SPORT2 and SPORT3 signals. If the secondary data pins are not needed, the corresponding pins can be used for general-purpose timer purposes.

Table 9-2. Port A Pin Configuration

Pin	GPIO	PORTA_MUX	1st Function (MUX = 00)	2nd Function (MUX = 01)	3rd Function (MUX = 10)	4th Function (MUX = 11)	Additional Use
PA0	PA0	1:0	SPORT2 TFS	-	-	-	-
PA1	PA1	3:2	SPORT2 DTSEC	TMR4	-	-	-
PA2	PA2	5:4	SPORT2 DTPRI	-	-	-	-
PA3	PA3	7:6	SPORT2 TSCLK	-	-	-	-
PA4	PA4	9:8	SPORT2 RFS	-	-	-	-
PA5	PA5	11:10	SPORT2 DRSEC	TMR5	-	-	-
PA6	PA6	13:12	SPORT2 DRPRI	-	-	-	-
PA7	PA7	15:14	SPORT2 RSCLK	-	-	-	TACLK0 ¹
PA8	PA8	17:16	SPORT3 TFS	-	-	-	TACLK1 ¹
PA9	PA9	19:18	SPORT3 DTSEC	TMR6	-	-	-
PA10	PA10	21:20	SPORT3 DTPRI	-	-	-	TACLK2 ¹
PA11	PA11	23:22	SPORT3 TSCLK	-	-	-	TACLK3 ¹
PA12	PA12	25:24	SPORT3 RFS	-	-	-	TACLK4 ¹
PA13	PA13	27:26	SPORT3 DRSEC	TMR7	-	-	TACLK5 ¹
PA14	PA14	29:28	SPORT3 DRPRI	-	-	-	TACLK6 ¹
PA15	PA15	31:30	SPORT3 RSCLK	-	-	-	TACI7 ¹ , TACLK7 ¹

¹ To enable timer alternate capture and clock inputs, either the GPIO or the multiplexed peripheral must enable the pin input driver. This driver is not enabled by the timer.

Pin Multiplexing Scheme

Port B

Port B consists of 15 pins, referred to as PB0 to PB14, as shown in [Table 9-3](#). Besides the 15 GPIOs, this port homes TW1, UART2, UART3 and SPI2 signals. If the SPI2 slave select signals are not needed, the corresponding pins can be used for general-purpose timer purposes.

Table 9-3. Port B Pin Configuration

Pin	GPIO	PORTB_MUX	1st Function (MUX = 00)	2nd Function (MUX = 01)	3rd Function (MUX = 10)	4th Function (MUX = 11)	Additional Use
PB0	PB0	1:0	TW11 SCL ¹	-	-	-	-
PB1	PB1	3:2	TW11 SDA ¹	-	-	-	-
PB2	PB2	5:4	UART3 RTS	-	-	-	-
PB3	PB3	7:6	UART3 CTS	-	-	-	-
PB4	PB4	9:8	UART2 TX	-	-	-	-
PB5	PB5	11:10	UART2 RX	-	-	-	TAC12 ²
PB6	PB6	13:12	UART3 TX	-	-	-	-
PB7	PB7	15:14	UART3 RX	-	-	-	TAC13 ²
PB8	PB8	17:16	SPI2 SS	TMR0	-	-	-
PB9	PB9	19:18	SPI2 SSEL1	TMR1	-	-	-
PB10	PB10	21:20	SPI2 SSEL2	TMR2	-	-	-
PB11	PB11	23:22	SPI2 SSEL3	TMR3	-	-	HWAIT ³
PB12	PB12	25:24	SPI2 SCK	-	-	-	-
PB13	PB13	27:26	SPI2 MOSI	-	-	-	-
PB14	PB14	29:28	SPI2 MISO	-	-	-	-

- 1 PB_0 and PB_1 are I²C pins which also have GPIO capability. Since the I²C pads can only drive low, the GPIO for these two bits cannot drive a 1. These pads should be used with an external pull-up, so that a 1 is seen when they are not pulling down.
- 2 To enable timer alternate capture and clock Inputs, either the GPIO or the multiplexed peripheral must enable the pin input driver. This driver is not enabled by the timer.

- The Boot Host Wait (HWAIT) signal is a GPIO output that is driven and toggled by the boot kernel at boot time. An external pulling resistor is required for proper operation. A pull-up resistor instructs the HWAIT signal to behave active high (low when ready for data). A pull-down resistor instructs the HWAIT signal to behave active low (high when ready for data). After boot it can be used for other purposes. If PB11 is used for other purposes (for example, timer or SPI operation), the HWAITA signal on PH7 can be used alternatively. The Alternate Host Wait (HWAITA) can be alternatively used instead of HWAIT on PH7 when programming the OTP_ALTERNATE_HWAIT bit in the PBS_MAIN_LO OTP memory page. For details, see [“System Reset and Booting” on page 17-1](#).

Port C

Port C consists of 14 pins, referred to as PC0 to PC13, as shown in [Table 9-4](#). Besides the 14 GPIOs, this port homes SPORT0 and SDIO signals.

Table 9-4. Port C Pin Configuration

Pin	GPIO	PORTC_MUX	1st Function (MUX = 00)	2nd Function (MUX = 01)	3rd Function (MUX = 10)	4th Function (MUX = 11)	Additional Use
PC0	PC0	1:0	SPORT0 TFS	-	-	-	-
PC1	PC1	3:2	SPORT0 DTSEC	MMCLK	-	-	-
PC2	PC2	5:4	SPORT0 DTPRI	-	-	-	-
PC3	PC3	7:6	SPORT0 TSCLK	-	-	-	-
PC4	PC4	9:8	SPORT0 RFS	-	-	-	-
PC5	PC5	11:10	SPORT0 DRSEC	MBCLK	-	-	-
PC6	PC6	13:12	SPORT0 DRPRI	-	-	-	-
PC7	PC7	15:14	SPORT0 RSCLK	-	-	-	-
PC8	PC8	17:16	SD D0	-	-	-	-
PC9	PC9	19:18	SD D1	-	-	-	-
PC10	PC10	21:20	SD D2	-	-	-	-
PC11	PC11	23:22	SD D3	-	-	-	-
PC12	PC12	25:24	SD CLK	-	-	-	-
PC13	PC13	27:26	SD CMD	-	-	-	-

Pin Multiplexing Scheme

Port D

Port D consists of 16 pins, referred to as PD0 to PD15, as shown in [Table 9-5](#). Besides the 16 GPIOs, this port homes data signals of all three EPPI ports and of the host port. Additionally, there are the SPORT1 signals and four columns and four rows of the keypad peripheral.

This port provides flexible configurations, whereby 8-, 16-, or 24-bit EPPI configurations can be balanced against 8- or 16-bit host operation.

Table 9-5. Port D Pin Configuration

Pin	GPIO	PORTD _MUX	1st Function (MUX = 00)	2nd Function (MUX = 01)	3rd Function (MUX = 10)	4th Function (MUX = 11)	Additional Use
PD0	PD0	1:0	PPI1 D0	HOST D8	SPORT1 TFS	PPI0 D18	-
PD1	PD1	3:2	PPI1 D1	HOST D9	SPORT1 DTSEC	PPI0 D19	-
PD2	PD2	5:4	PPI1 D2	HOST D10	SPORT1 DTPRI	PPI0 D20	-
PD3	PD3	7:6	PPI1 D3	HOST D11	SPORT1 TSCLK	PPI0 D21	-
PD4	PD4	9:8	PPI1 D4	HOST D12	SPORT1 RFS	PPI0 D22	-
PD5	PD5	11:10	PPI1 D5	HOST D13	SPORT1 DRSEC	PPI0 D23	-
PD6	PD6	13:12	PPI1 D6	HOST D14	SPORT1 DRPRI	-	-
PD7	PD7	15:14	PPI1 D7	HOST D15	SPORT1 RSCLK	-	-
PD8	PD8	17:16	PPI1 D8	HOST D0	PPI2 D0	KEY ROW0	-
PD9	PD9	19:18	PPI1 D9	HOST D1	PPI2 D1	KEY ROW1	-
PD10	PD10	21:20	PPI1 D10	HOST D2	PPI2 D2	KEY ROW2	-
PD11	PD11	23:22	PPI1 D11	HOST D3	PPI2 D3	KEY ROW3	-
PD12	PD12	25:24	PPI1 D12	HOST D4	PPI2 D4	KEY COL0	-
PD13	PD13	27:26	PPI1 D13	HOST D5	PPI2 D5	KEY COL1	-
PD14	PD14	29:28	PPI1 D14	HOST D6	PPI2 D6	KEY COL2	-
PD15	PD15	31:30	PPI1 D15	HOST D7	PPI2 D7	KEY COL3	-

Port E

Port E consists of 16 pins, referred to as PE0 to PE15, as shown in [Table 9-6](#). Besides the 16 GPIOs, this port homes data signals for SPI0, UART0, and TWI0. Furthermore, there are UART1 hardware flow control signals and PPI1 clock and frame sync signals. If not all signals of the SPI0 are needed in an application, the associated pins can operate as rows and columns for the keypad peripheral.

Table 9-6. Port E Pin Configuration

Pin	GPIO	PORTE_MUX	1st Function (MUX = 00)	2nd Function (MUX = 01)	3rd Function (MUX = 10)	4th Function (MUX = 11)	Additional Use
PE0	PE0	1:0	SPI0 SCK	KEY COL7	-	-	-
PE1	PE1	3:2	SPI0 MISO	KEY ROW6	-	-	-
PE2	PE2	5:4	SPI0 MOSI	KEY COL6	-	-	-
PE3	PE3	7:6	SPI0 SS	KEY ROW5	-	-	-
PE4	PE4	9:8	SPI0 SEL1	KEY COL5	-	-	-
PE5	PE5	11:10	SPI0 SEL2	KEY ROW4	-	-	-
PE6	PE6	13:12	SPI0 SEL3	KEY COL4	-	-	-
PE7	PE7	15:14	UART0 TX	KEY ROW7	-	-	-
PE8	PE8	17:16	UART0 RX	-	-	-	TACIO ¹
PE9	PE9	19:18	UART1 RTS	-	-	-	-
PE10	PE10	21:20	UART1 CTS	-	-	-	-
PE11	PE11	23:22	PPI1 CLK	-	-	-	-
PE12	PE12	25:24	PPI1 FS1	-	-	-	-
PE13	PE13	27:26	PPI1 FS2	-	-	-	-
PE14	PE14	29:28	TWI0 SCL	-	-	-	-
PE15	PE15	31:30	TWI0 SDA	-	-	-	-

¹ To enable timer alternate capture and clock inputs, either the GPIO or the multiplexed peripheral must enable the pin input driver. This driver is not enabled by the timer.

Pin Multiplexing Scheme

Port F

Port F consists of 16 pins, referred to as PF0 to PF15, as shown in [Table 9-7](#). Besides the 16 GPIOs, this port homes 16 data signals of the PPI0 interface. This port can alternatively provide the ATAPI data signals if not multiplexed with the asynchronous bus.

Table 9-7. Port F Pin Configuration

Pin	GPIO	PORTF_MUX	1st Function (MUX = 00)	2nd Function (MUX = 01)	3rd Function (MUX = 10)	4th Function (MUX = 11)	Additional Use
PF0	PF0	1:0	PPI0 D0	ATAPI D0A ¹	-	-	-
PF1	PF1	3:2	PPI0 D1	ATAPI D1A ¹	-	-	-
PF2	PF2	5:4	PPI0 D2	ATAPI D2A ¹	-	-	-
PF3	PF3	7:6	PPI0 D3	ATAPI D3A ¹	-	-	-
PF4	PF4	9:8	PPI0 D4	ATAPI D4A ¹	-	-	-
PF5	PF5	11:10	PPI0 D5	ATAPI D5A ¹	-	-	-
PF6	PF6	13:12	PPI0 D6	ATAPI D6A ¹	-	-	-
PF7	PF7	15:14	PPI0 D7	ATAPI D7A ¹	-	-	-
PF8	PF8	17:16	PPI0 D8	ATAPI D8A ¹	-	-	-
PF9	PF9	19:18	PPI0 D9	ATAPI D9A ¹	-	-	-
PF10	PF10	21:20	PPI0 D10	ATAPI D10A ¹	-	-	-
PF11	PF11	23:22	PPI0 D11	ATAPI D11A ¹	-	-	-
PF12	PF12	25:24	PPI0 D12	ATAPI D12A ¹	-	-	-
PF13	PF13	27:26	PPI0 D13	ATAPI D13A ¹	-	-	-
PF14	PF14	29:28	PPI0 D14	ATAPI D14A ¹	-	-	-
PF15	PF15	31:30	PPI0 D15	ATAPI D15A ¹	-	-	-

¹ ATAPI data and address signals are routed to alternate homes when PORTF_MUX[1:0] == b#01.

Port G

Port G consists of 16 pins, referred to as PG0 to PG15, as shown in [Table 9-8](#). Besides the 16 GPIOs, this port homes EPPI0 control signals, all CAN signals, as well as the SPI1 signals. If additional SPI1 slave select signals are not needed by an application, the associated pins can alternatively function as Host DMA port or EPPI2 control signals. Also, the zero marker input of the counter module is there.

Table 9-8. Port G Pin Configuration

Pin	GPIO	PORTG_MUX	1st Function (MUX = 00)	2nd Function (MUX = 01)	3rd Function (MUX = 10)	4th Function (MUX = 11)	Additional Use
PG0	PG0	1:0	PPI0 CLK	-	-	-	TMRCLK ¹
PG1	PG1	3:2	PPI0 FS1	-	-	-	-
PG2	PG2	5:4	PPI0 FS2	ATAPI A0A ²	-	-	-
PG3	PG3	7:6	PPI0 D16	ATAPI A1A ²	-	-	-
PG4	PG4	9:8	PPI0 D17	ATAPI A2A ²	-	-	-
PG5	PG5	11:10	SPI1 SEL1	HOST CE	PPI2 FS2	CNT CZM	-
PG6	PG6	13:12	SPI1 SEL2	HOST RD	PPI2 FS1	-	-
PG7	PG7	15:14	SPI1 SEL3	HOST WR	PPI2 CLK	-	-
PG8	PG8	17:16	SPI1 SCK	-	-	-	-
PG9	PG9	19:18	SPI1 MISO	-	-	-	-
PG10	PG10	21:20	SPI1 MOSI	-	-	-	-
PG11	PG11	23:22	SPI1 SS	MTXONB	-	-	-
PG12	PG12	25:24	CAN0 TX	-	-	-	-
PG13	PG13	27:26	CAN0 RX	-	-	-	TACI4 ³
PG14	PG14	29:28	CAN1 TX	-	-	-	-
PG15	PG15	31:30	CAN1 RX	-	-	-	TACI5 ³

1 TMRCLK serves all eleven general-purpose timers.

2 ATAPI data and address signals are routed to alternate homes when PORTF_MUX[1:0] == b#01.

Pin Multiplexing Scheme

- To enable timer alternate capture and clock inputs, either the GPIO or the multiplexed peripheral must enable the pin input driver. This driver is not enabled by the timer.

Port H

Port H consists of 14 pins, referred to as PH0 to PH13, as shown in [Table 9-9](#). Besides the 14 GPIOs, this port homes six address lines of the parallel asynchronous memory interface. Furthermore, there are the UART1 data signals and set of miscellaneous control signals, such as Host DMA port strobes, handshaked-memory DMA request strobes, the third EPPI frame syncs, and the up- and down-count inputs of the counter module.

The boot host wait (HWAIT) and alternate boot host wait (HWAITA) are not associated with any hardware block. It is a normal GPIO pin that has a special purpose during booting. For details, see [“System Reset and Booting” on page 17-1](#).

Table 9-9. Port H Pin Configuration

Pin	GPIO	PORTH_MUX	1st Function (MUX = 00)	2nd Function (MUX = 01)	3rd Function (MUX = 10)	4th Function (MUX = 11)	Additional Use
PH0	PH0	1:0	UART1 TX	PPI1 FS3	-	-	-
PH1	PH1	3:2	UART1 RX	EPPI0 FS3	-	-	TACI1 ¹
PH2	PH2	5:4	ATAPI RESET	TMR8	EPPI2 FS3	-	-
PH3	PH3	7:6	HOST ADDR	TMR9	CNT CDG	-	-
PH4	PH4	9:8	HOST ACK	TMR10	CNT CUD	-	-
PH5	PH5	11:10	MTX	DMAR0	-	-	TACI8 ¹ , TACLK8 ¹
PH6	PH6	13:12	MRX	DMAR1	-	-	TACI9 ¹ , TACLK9 ¹
PH7	PH7	15:14	MRXONB	-	-	-	TACI10 ¹ , TACLK10 ¹ , HWAIT ²
PH8	PH8	17:16	A4	-	-	-	
PH9	PH9	19:18	A5	-	-	-	-

Table 9-9. Port H Pin Configuration (Cont'd)

Pin	GPIO	PORTH _MUX	1st Function (MUX = 00)	2nd Function (MUX = 01)	3rd Function (MUX = 10)	4th Function (MUX = 11)	Additional Use
PH10	PH10	21:20	A6	-	-	-	-
PH11	PH11	23:22	A7	-	-	-	-
PH12	PH12	25:24	A8	-	-	-	-
PH13	PH13	27:26	A9	-	-	-	-

- 1 To enable timer alternate capture and clock inputs, either the GPIO or the multiplexed peripheral must enable the pin input driver. This driver is not enabled by the timer.
- 2 The Boot Host Wait (HWAIT) signal is a GPIO output that is driven and toggled by the boot kernel at boot time. An external pulling resistor is required for proper operation. A pull-up resistor instructs the HWAIT signal to behave active high (low when ready for data). A pull-down resistor instructs the HWAIT signal to behave active low (high when ready for data). After boot, it can be used for other purposes. If PH7 is used for other purposes (for example, MXVR operation), the HWAITA signal on PB11 can be used alternatively. HWAITA operation is enabled by programming the OTP_ALTERNATE_HWAIT bit in the PBS_MAIN_LO OTP memory page. For details, see [“System Reset and Booting” on page 17-1](#).

Port I

Port I consists of 16 pins, referred to as PI0 to PI15, as shown in [Table 9-10](#). Besides the 16 GPIOs, this port homes the upper 16 address lines of the parallel asynchronous memory interface and the clock for the synchronous NOR flash interface.

Table 9-10. Port I Pin Configuration

Pin	GPIO	PORTI _MUX	1st Function (MUX = 00)	2nd Function (MUX = 01)	3rd Function (MUX = 10)	4th Function (MUX = 11)	Additional Use
PI0	PI0	1:0	A10	-	-	-	-
PI1	PI1	3:2	A11	-	-	-	-
PI2	PI2	5:4	A12	-	-	-	-
PI3	PI3	7:6	A13	-	-	-	-
PI4	PI4	9:8	A14	-	-	-	-
PI5	PI5	11:10	A15	-	-	-	-

Pin Multiplexing Scheme

Table 9-10. Port I Pin Configuration (Cont'd)

Pin	GPIO	PORTI_MUX	1st Function (MUX = 00)	2nd Function (MUX = 01)	3rd Function (MUX = 10)	4th Function (MUX = 11)	Additional Use
PI6	PI6	13:12	A16	-	-	-	-
PI7	PI7	15:14	A17	-	-	-	-
PI8	PI8	17:16	A18	-	-	-	-
PI9	PI9	19:18	A19	-	-	-	-
PI10	PI10	21:20	A20	-	-	-	-
PI11	PI11	23:22	A21	-	-	-	-
PI12	PI12	25:24	A22	-	-	-	-
PI13	PI13	27:26	A23	-	-	-	-
PI14	PI14	29:28	A24	-	-	-	-
PI15	PI15	31:30	A25	NOR CLK	-	-	-

Port J

Port J consists of 16 pins, referred to as PJ0 to PJ15, as shown in [Table 9-11](#). Besides the 16 GPIOs, this port provides various control signals for the NAND flash, NOR flash, and ATAPI interfaces.

Table 9-11. Port J Pin Configuration

Pin	GPIO	PORTJ_MUX	1st Function (MUX = 00)	2nd Function (MUX = 01)	3rd Function (MUX = 10)	4th Function (MUX = 11)	Additional Use
PJ0	PJ0	1:0	AMC ARDY / NOR WAIT	-	-	-	-
PJ1	PJ1	3:2	NAND CE	-	-	-	-
PJ2	PJ2	5:4	NAND RB	-	-	-	-
PJ3	PJ3	7:6	ATAPI DIOR	-	-	-	-
PJ4	PJ4	9:8	ATAPI DIOW	-	-	-	-
PJ5	PJ5	11:10	ATAPI CS0	-	-	-	-

Table 9-11. Port J Pin Configuration (Cont'd)

Pin	GPIO	PORTJ_MUX	1st Function (MUX = 00)	2nd Function (MUX = 01)	3rd Function (MUX = 10)	4th Function (MUX = 11)	Additional Use
PJ6	PJ6	13:12	ATAPI CS1	-	-	-	-
PJ7	PJ7	15:14	ATAPI DMACK	-	-	-	-
PJ8	PJ8	17:16	ATAPI DMARQ	-	-	-	-
PJ9	PJ9	19:18	ATAPI INTRQ	-	-	-	-
PJ10	PJ10	21:20	ATAPI IORDY	-	-	-	-
PJ11	PJ11	23:22	AMC BR	-	-	-	-
PJ12	PJ12	25:24	AMC BG	-	-	-	-
PJ13	PJ13	27:26	AMC BGH	-	-	-	-

Port Multiplexing Control

By default, after reset, all port pins are in GPIO input mode with their output and input drivers disabled. As a result, all unused port pins can be left unconnected. Disabled pins appear in high-impedance mode to external circuits and are pulled low to internal logic.

Each port has two dedicated MMRs that control the port multiplexing, the 16-bit function enable (`PORTx_FER`) registers, and the 32-bit port multiplexing (`PORTx_MUX`) registers.



In this chapter, the naming convention for registers and bits uses a lowercase “x” to represent A to J. For example, the name `PORTx_FER` represents `PORTA_FER`, `PORTB_FER`, and so on, through `PORTJ_FER`. The bit name `Px0` represents `PA0`, `PB0`, and so on, through `PJ0`. This convention is used in register descriptions common to the ten ports.

Pin Multiplexing Scheme

Each bit in the 16-bit `PORTx_FER` registers represents one port pin. For example, bit 1 of the `PORTA_FER` register sets the PA1 pin to GPIO operation mode when cleared. When set, one of the available peripheral functions becomes active. The PA1 pin can either operate as a secondary transmit data signal of `SPORT2` or as `PWM/capture/clock` pin of Timer 4.

Every pair of bits in the `PORTx_MUX` registers controls the multiplexing between the peripheral functions available to a pin. This is a 2-bit field because some pins provide up to four options. The truth table of the bit field is identical to all ADSP-BF54x Blackfin processor family derivatives, regardless of whether all options are available on a given silicon.

In the case of the PA1 example, bit 3 and bit 2 control the multiplexer of the PA1 pin. The truth table of the entire function enable and multiplexing control is shown in [Table 9-12](#).


Table 9-12. Port Multiplexing Control Example

PORTA_FER [1]	PORTA_MUX [3:2]	PA1 Function
0	00	GPIO
0	01	GPIO
0	10	GPIO
0	11	GPIO
1	00	SPORT2 DTSEC
1	01	TMR4
1	10	Reserved
1	11	Reserved

The port multiplexing scheme provides best granularity, as every pin can be controlled on an individual basis. If `SPORT2` is used in any mode that does not require the secondary transmit data feature, the PA1 pin can still be used as GPIO or as TMR4.

GPIO Functionality

Every port pin can operate in GPIO mode. This is the default after reset and is controlled by the port-specific `PORTx_FER` function enable register. Every port has a dedicated set of MMR registers that control GPIO functionality. Every bit in these registers represents a certain GPIO pin of the specific port. Refer to [Figure 9-2](#) for a related diagram.

 In this chapter, the naming convention for registers and bits uses a lowercase “x” to represent A through J. For example, the name `PORTx_FER` represents `PORTA_FER`, `PORTB_FER`, and so on, through `PORTJ_FER`. The bit name `Px0` represents `PA0`, `PB0`, and so on, through `PJ0`. This convention is used to discuss registers common to the ten ports.

By default, every GPIO is in input mode. The input drivers are not enabled which avoids the need for unnecessary current sinks and the external pulling of resistors on unused or do not care pins.

Input Mode

The default mode of every GPIO pin after reset is input mode, but the input drivers are not enabled. To enable any GPIO input drivers, set the corresponding bits in the input enable register `PORTx_INEN`. When enabled, a read from the `PORTx` register returns the logical state of the input pin. The input signal does not overwrite the state of the flip-flop used for the output case. That state can only be altered by software. If the input driver is enabled, a write to the `PORTx` register can alter the state of the flip-flop, but the change cannot be read back.

Output Mode

Any GPIO pin can be configured for output mode. The GPIO output drivers are enabled by setting the corresponding bits in the direction registers. Direction registers are implemented as a pair of write-1-to-set (W1S)

GPIO Functionality

and write-1-to-clear (W1C) MMRs, called `PORTx_DIR_SET` and `PORTx_DIR_CLEAR`. This way, direction of the signal flow of individual GPIO pins can be altered by separate software threads without mutually impacting other GPIOs on the same port. Both registers return the same value when read. A logical 1 indicates an enabled output.

The state of output pins is controlled by the `PORTx` registers. A logical 0 drives the output low. A logical 1 drives the output high. While the `PORTx` register can be written to alter all GPIOs of a specific port at once, there is also a pair of W1S and W1C MMRs, called `PORTx_SET` and `PORTx_CLEAR` that enable manipulation of individual GPIO outputs. The state of the outputs can be obtained by reading the `PORTx` registers.

Because the state of the GPIO output can already be controlled before the output driver is enabled, it is recommended to first set or clear the flip-flop to avoid any volatile levels on the output.

Open-Drain Mode

Every GPIO can also be used in open-drain mode. To accomplish this, first, clear the respective bit in the `PORTx` or `PORTx_CLEAR` register then set the one bit in the `PORTx_INEN` register. Reads from the `PORTx` register then return the status from the pin and do not return the state of the internal flip-flop. By toggling the output driver through the `PORTx_DIR_SET` and `PORTx_DIR_CLEAR` register pair, the output signal can be pulled low or three-stated as required. Note that the polarity of the driven signal can be inverted when the internal flip-flop is set instead. When a GPIO port is used in open-drain mode, care must be taken not to exceed the V_{IH} operating condition associated with the respective pin.

Pin Interrupts

On the ADSP-BF54x Blackfin processor family, the pin interrupts have been completely decoupled from basic GPIO functionality due to the following set of advantages:

- Flexible mapping scheme enables pins from up to four different ports to be grouped to one common interrupt scheme.
- Interrupts work on input and output pins regardless of whether in GPIO or functional mode.

ADSP-BF54x Blackfin processors have four SIC interrupt channels dedicated to pin interrupt purposes. These channels are managed by four hardware blocks, called PINT0, PINT1, PINT2, and PINT3. Every PINT_x block can sense up to 32 pins. While PINT0 and PINT1 can sense the pins of port A and port B, PINT2 and PINT3 manage all the pins from port C to port J as shown in [Figure 9-2](#).

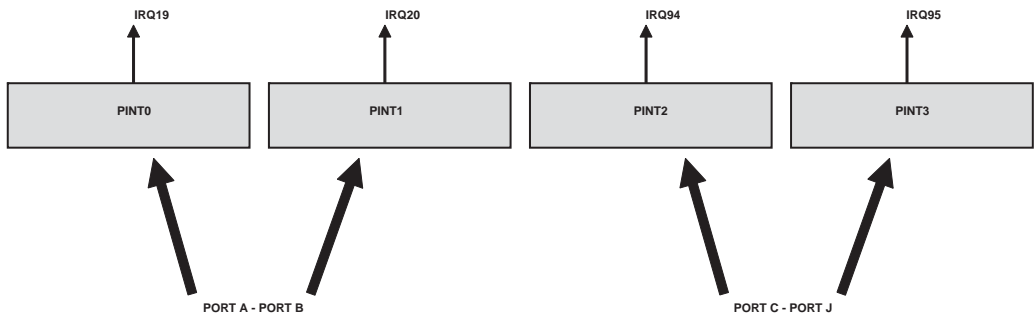


Figure 9-2. Signal Flow

Pin Interrupts

The diagram shown in [Figure 9-1 on page 9-3](#) shows the signal flow from the pin through the `PINTx` module to the SIC controller. Special attention is required with regard to how the pins are assigned to the `PINTx` modules as shown in [Figure 9-3](#).

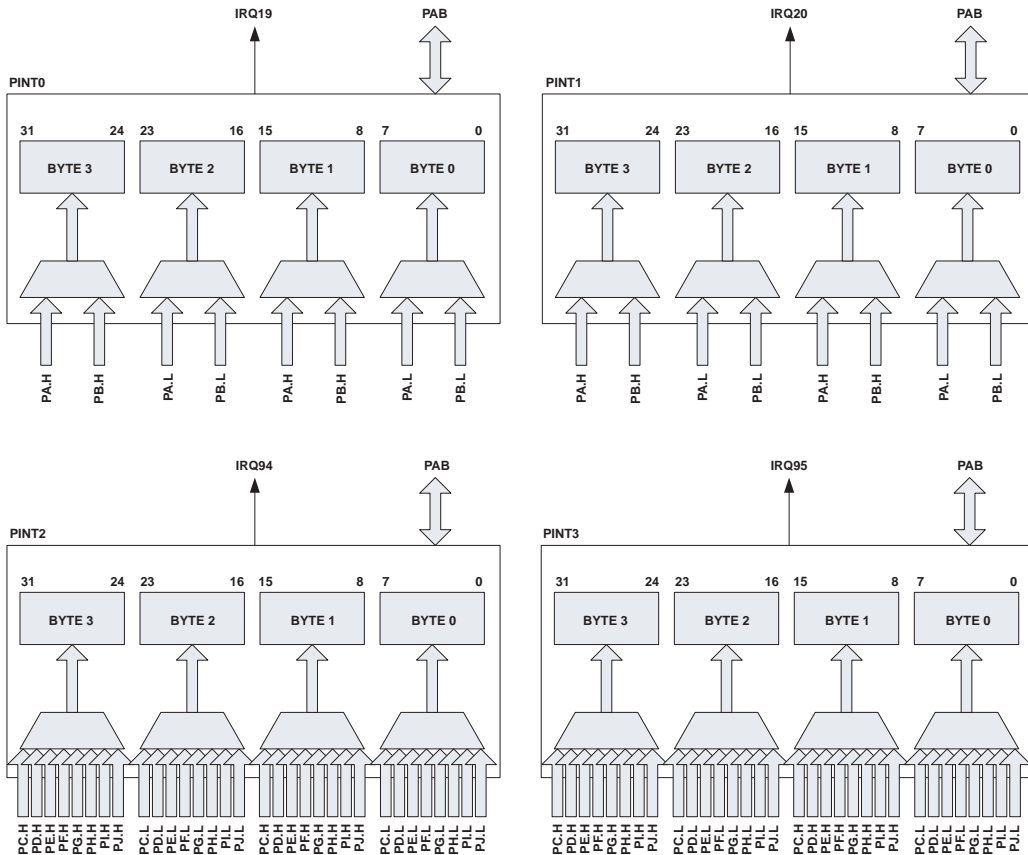


Figure 9-3. Pin-to-Interrupt Assignment

The ten ports are subdivided into 8-bit half ports, resulting in lower and upper half 8-bit units. The `PINTx_ASSIGN` registers control the 8-bit multiplexers shown in [Figure 9-3](#). Lower half units of eight pins can be

forwarded to either byte 0 or byte 2 of either associated `PINTx` block. Upper half units can be forwarded to either byte 1 or byte 3 of the pin interrupt blocks, without further restrictions.

When a half port is assigned to a byte in any `PINTx` block, the state of the eight pins (regardless of GPIO or function, input or output) can be seen in the `PINTx_PINSTATE` register. While neither input nor output drivers of the pin are enabled, the pin state is read as zero. The `PINTx_PINSTATE` register reports the inverted state of the pin if the signal inverter is activated by the `PINTx_INVERT_SET` register. The inverter can be enabled on an individual bit by bit basis. Every bit in the `PINTx_INVERT_SET/CLEAR` register pair represents a pin signal.

As shown in [Figure 9-1 on page 9-3](#), the interrupt can be generated on an active high level of the signal or a raising edge of the signal. The default behavior is level sensitivity. `PINTx_EDGE_SET` register can be used to change the behavior to edge sensitivity. By enabling the inverter using the `PINTx_INVERT_SET` register, the interrupt behavior can be altered to trigger on active-low signals or falling edges.

Regardless whether in level-sensitive or edge-sensitive mode, an interrupt is always latched by the hardware. Latched signals can be read from the `PINTx_LATCH` registers. Latches can only be cleared by software or a hardware reset. To clear, write the `PINTx_REQUEST` or the `PINTx_LATCH` register. If the pin state does not change by the time the interrupt service routine returns, the interrupt is requested again, when in level-sensitive mode.

Because every `PINTx` block groups up to 32 pin signals, the `PINTx_MASK_SET/CLEAR` register pair can control which of the signals can request an interrupt at system level. Software may interrogate the `PINTx_REQUEST` register for signaling pins. `PINTx_REQUEST` bits represent a logical AND between the mask and the latch. When any of these bits is set, an interrupt is forwarded to the SIC controller.

Programming Model

All MMR registers in the pin interrupt module are 32 bits wide. Individual bits of `PINTx` registers represent the associated pins. Nevertheless, the 32 bits can also be seen as four groups of eight bits. Each group can manage up to eight pins out of either the lower or an upper half of any associated port.

Programming Model

[Figure 9-4](#), [Figure 9-5](#), and [Figure 9-6](#) show the programming model of the general-purpose ports. This includes GPIO input and output operation, as well as open-drain mode. [Figure 9-6](#) (the third part of the diagram) illustrates the model of the pin interrupt `PINTx` modules.

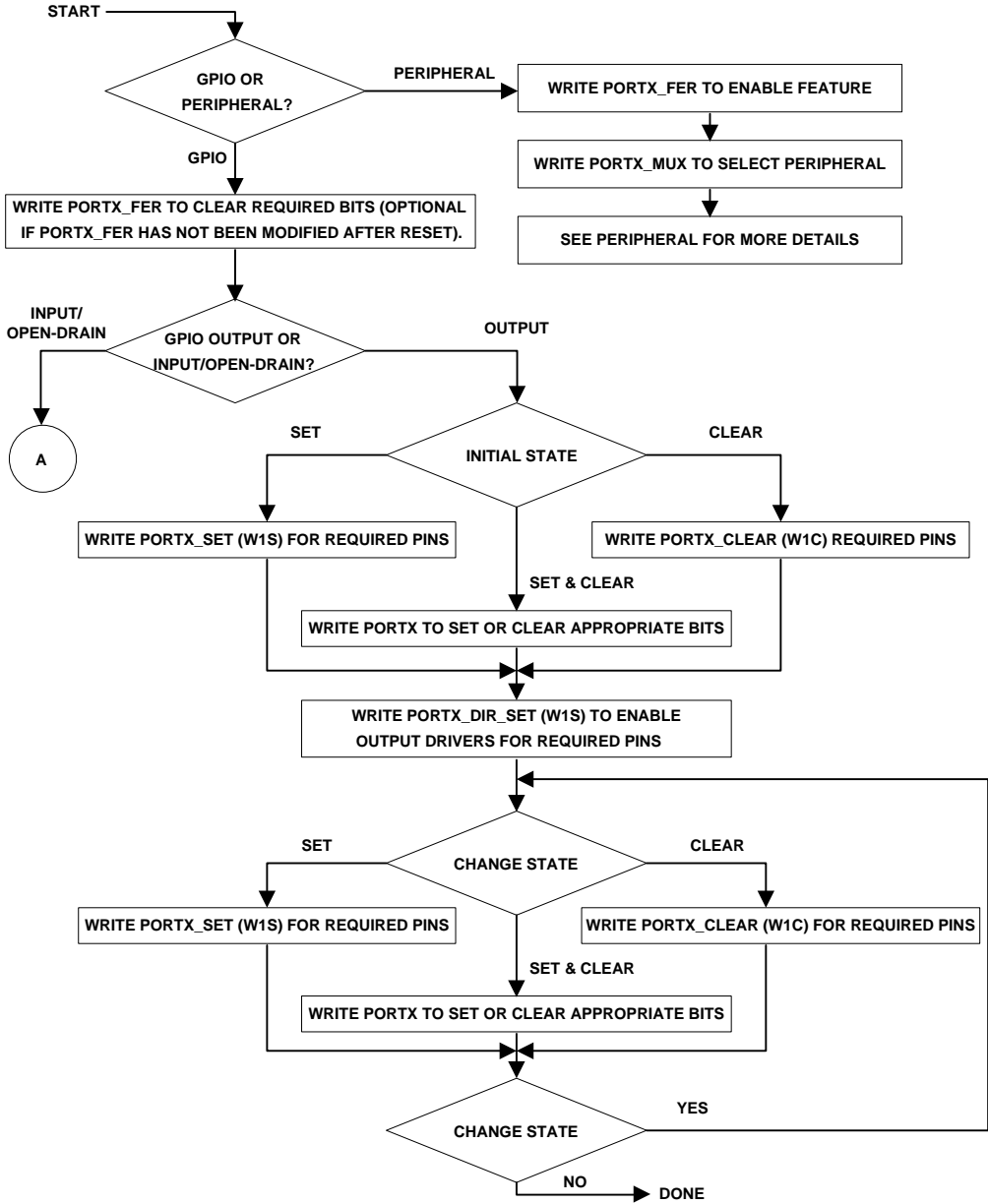
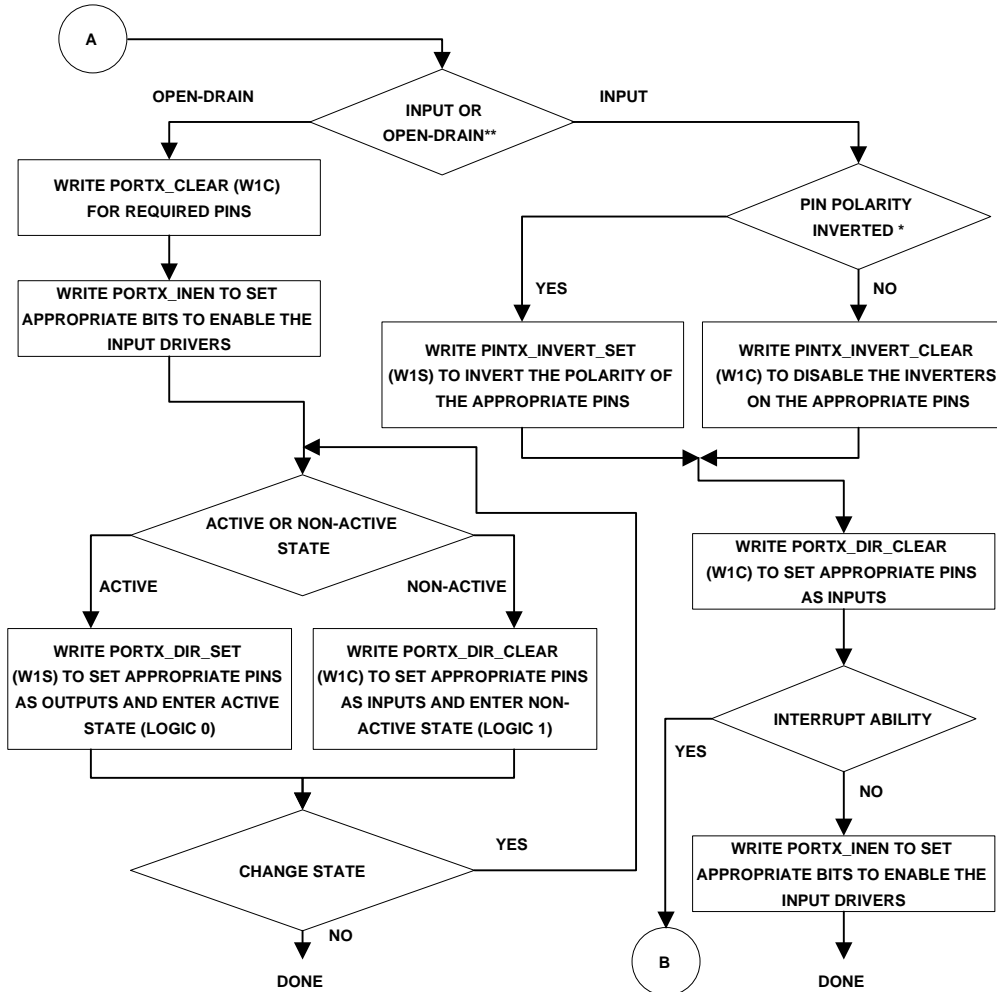


Figure 9-4. GPIO Programming Model Flow (Part 1)

Programming Model



* The pin polarity set at this point will effect the behaviour of the interrupt functionality detailed in the next figure. If the invert bit is set for a given pin, and edge sensitive interrupts are configured. The interrupt will be latched on detection of a falling edge. If the inverse bit is clear, edge sensitive interrupts are generated on the rising edge. For level sensitive interrupts, enabling the inverter will result in interrupts being detected on a low signal. Disabling the inverter will result in interrupts being latched on high signals.

** Open-drain mode assumes an external pull-up resistor is fitted

Figure 9-5. GPIO Programming Model Flow (Part 2)

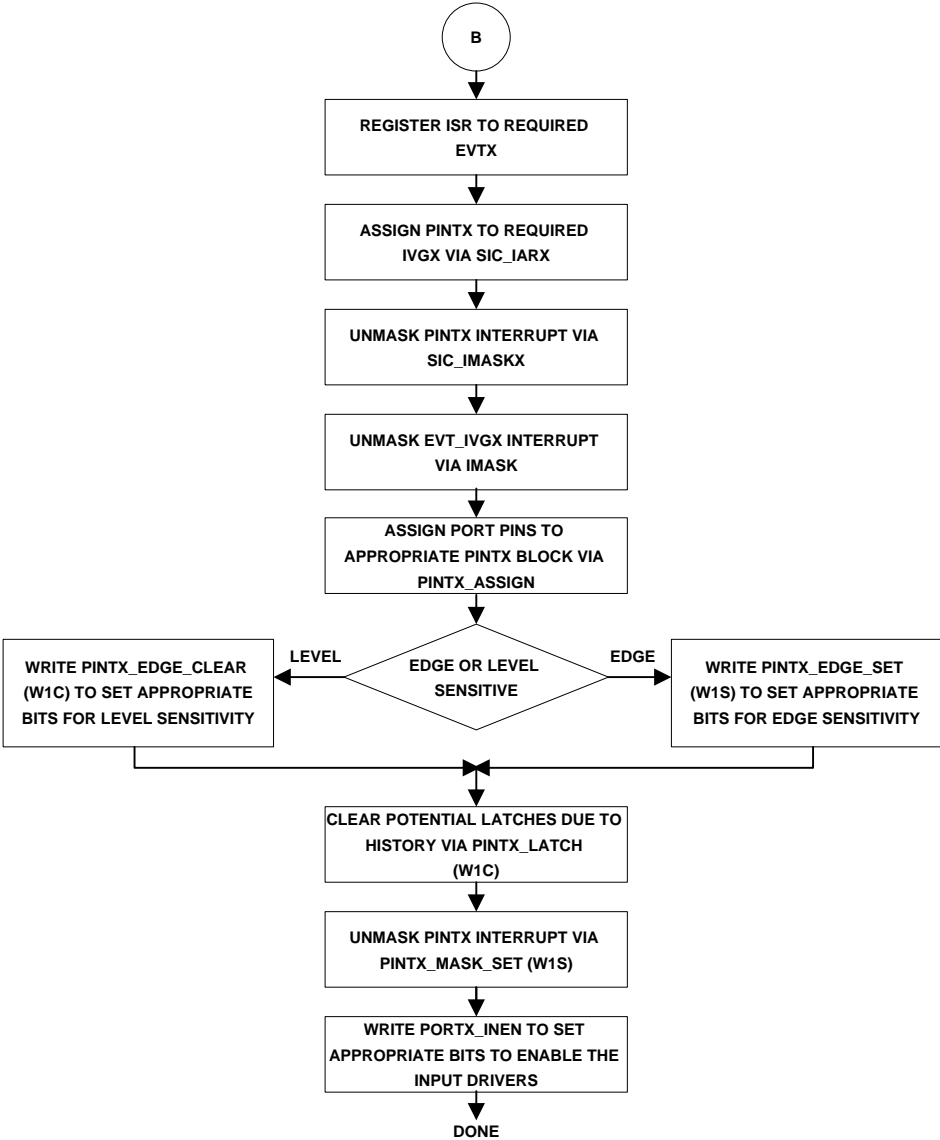


Figure 9-6. GPIO Programming Model Flow (Part 3)

Port Registers

The general-purpose ports are programmed using memory-mapped registers.

[Table 9-13](#) and [Table 9-14](#) list the registers for port control and pin interrupt programming.

Table 9-13. Port Control Registers (Multiplexing and GPIO)

Address Offset	Register Name	More Information begins ...	Supported Operation	Reset Value
0xFFC014C0	PORTA_FER	on page 9-36	R / W	0x0000
0xFFC014C4	PORTA	on page 9-41	R / W	0x0000
0xFFC014C8	PORTA_SET	on page 9-41	R / W1S	0x0000
0xFFC014CC	PORTA_CLEAR	on page 9-41	R / W1C	0x0000
0xFFC014D0	PORTA_DIR_SET	on page 9-39	R / W1S	0x0000
0xFFC014D4	PORTA_DIR_CLEAR	on page 9-39	R / W1C	0x0000
0xFFC014D8	PORTA_INEN	on page 9-40	R / W	0x0000
0xFFC014DC	PORTA_MUX	on page 9-36	R / W	0x00000000 ¹
0xFFC014E0	PORTB_FER	on page 9-36	R / W	0x0000
0xFFC014E4	PORTB	on page 9-41	R / W	0x0000
0xFFC014E8	PORTB_SET	on page 9-41	R / W1S	0x0000
0xFFC014EC	PORTB_CLEAR	on page 9-41	R / W1C	0x0000
0xFFC014F0	PORTB_DIR_SET	on page 9-39	R / W1S	0x0000
0xFFC014F4	PORTB_DIR_CLEAR	on page 9-39	R / W1C	0x0000
0xFFC014F8	PORTB_INEN	on page 9-40	R / W	0x0000
0xFFC014FC	PORTB_MUX	on page 9-36	R / W	0x00000000 ¹
0xFFC01500	PORTC_FER	on page 9-36	R / W	0x0000
0xFFC01504	PORTC	on page 9-41	R / W	0x0000

Table 9-13. Port Control Registers (Multiplexing and GPIO) (Cont'd)

Address Offset	Register Name	More Information begins ...	Supported Operation	Reset Value
0xFFC01508	PORTC_SET	on page 9-41	R / W1S	0x0000
0xFFC0150C	PORTC_CLEAR	on page 9-41	R / W1C	0x0000
0xFFC01510	PORTC_DIR_SET	on page 9-39	R / W1S	0x0000
0xFFC01514	PORTC_DIR_CLEAR	on page 9-39	R / W1C	0x0000
0xFFC01518	PORTC_INEN	on page 9-40	R / W	0x0000
0xFFC0151C	PORTC_MUX	on page 9-36	R / W	0x00000000 ¹
0xFFC01520	PORTD_FER	on page 9-36	R / W	0x0000
0xFFC01524	PORTD	on page 9-41	R / W	0x0000
0xFFC01528	PORTD_SET	on page 9-41	R / W1S	0x0000
0xFFC0152C	PORTD_CLEAR	on page 9-41	R / W1C	0x0000
0xFFC01530	PORTD_DIR_SET	on page 9-39	R / W1S	0x0000
0xFFC01534	PORTD_DIR_CLEAR	on page 9-39	R / W1C	0x0000
0xFFC01538	PORTD_INEN	on page 9-40	R / W	0x0000
0xFFC0153C	PORTD_MUX	on page 9-36	R / W	0x00000000 ¹
0xFFC01540	PORTE_FER	on page 9-36	R / W	0x0000
0xFFC01544	PORTE	on page 9-41	R / W	0x0000
0xFFC01548	PORTE_SET	on page 9-41	R / W1S	0x0000
0xFFC0154C	PORTE_CLEAR	on page 9-41	R / W1C	0x0000
0xFFC01550	PORTE_DIR_SET	on page 9-39	R / W1S	0x0000
0xFFC01554	PORTE_DIR_CLEAR	on page 9-39	R / W1C	0x0000
0xFFC01558	PORTE_INEN	on page 9-40	R / W	0x0000
0xFFC0155C	PORTE_MUX	on page 9-36	R / W	0x00000000 ¹
0xFFC01560	PORTF_FER	on page 9-36	R / W	0x0000
0xFFC01564	PORTF	on page 9-41	R / W	0x0000

Port Registers

Table 9-13. Port Control Registers (Multiplexing and GPIO) (Cont'd)

Address Offset	Register Name	More Information begins ...	Supported Operation	Reset Value
0xFFC01568	PORTF_SET	on page 9-41	R / W1S	0x0000
0xFFC0156C	PORTF_CLEAR	on page 9-41	R / W1C	0x0000
0xFFC01570	PORTF_DIR_SET	on page 9-39	R / W1S	0x0000
0xFFC01574	PORTF_DIR_CLEAR	on page 9-39	R / W1C	0x0000
0xFFC01578	PORTF_INEN	on page 9-40	R / W	0x0000
0xFFC0157C	PORTF_MUX	on page 9-36	R / W	0x00000000 ¹
0xFFC01580	PORTG_FER	on page 9-36	R / W	0x0000
0xFFC01584	PORTG_DIR_CLEAR	on page 9-36	R / W	0x0000
0xFFC01588	PORTG_SET	on page 9-41	R / W1S	0x0000
0xFFC0158C	PORTG_CLEAR	on page 9-41	R / W1C	0x0000
0xFFC01590	PORTG_DIR_SET	on page 9-39	R / W1S	0x0000
0xFFC01594	PORTG	on page 9-41	R / W1C	0x0000
0xFFC01598	PORTG_INEN	on page 9-40	R / W1S	0x0000
0xFFC0159C	PORTG_MUX	on page 9-36	R / W	0x00000000 ¹
0xFFC015A0	PORTH_FER	on page 9-36	R / W	0x0000
0xFFC015A4	PORTH	on page 9-41	R / W	0x0000
0xFFC015A8	PORTH_SET	on page 9-41	R / W1S	0x0000
0xFFC015AC	PORTH_CLEAR	on page 9-41	R / W1C	0x0000
0xFFC015B0	PORTH_DIR_SET	on page 9-39	R / W1S	0x0000
0xFFC015B4	PORTH_DIR_CLEAR	on page 9-39	R / W1C	0x0000
0xFFC015B8	PORTH_INEN	on page 9-40	R / W	0x0000
0xFFC015BC	PORTH_MUX	on page 9-36	R / W	0x00000000 ¹
0xFFC015C0	PORTI_FER	on page 9-36	R / W	0x0000
0xFFC015C4	PORTI	on page 9-41	R / W	0x0000

Table 9-13. Port Control Registers (Multiplexing and GPIO) (Cont'd)

Address Offset	Register Name	More Information begins ...	Supported Operation	Reset Value
0xFFC015C8	PORTI_SET	on page 9-41	R / W1S	0x0000
0xFFC015CC	PORTI_CLEAR	on page 9-41	R / W1C	0x0000
0xFFC015D0	PORTI_DIR_SET	on page 9-39	R / W1S	0x0000
0xFFC015D4	PORTI_DIR_CLEAR	on page 9-39	R / W1C	0x0000
0xFFC015D8	PORTI_INEN	on page 9-40	R / W	0x0000
0xFFC015DC	PORTI_MUX	on page 9-36	R / W	0x00000000 ¹
0xFFC015E0	PORTJ_FER	on page 9-36	R / W	0x0000
0xFFC015E4	PORTJ	on page 9-41	R / W	0x0000
0xFFC015E8	PORTJ_SET	on page 9-41	R / W1S	0x0000
0xFFC015EC	PORTJ_CLEAR	on page 9-41	R / W1C	0x0000
0xFFC015F0	PORTJ_DIR_SET	on page 9-39	R / W1S	0x0000
0xFFC015F4	PORTJ_DIR_CLEAR	on page 9-39	R / W1C	0x0000
0xFFC015F8	PORTJ_INEN	on page 9-40	R / W	0x0000
0xFFC015FC	PORTJ_MUX	on page 9-36	R / W	0x00000000 ¹

¹ May differ on certain derivatives and BMODE settings.

[Table 9-14](#) and [Table 9-13](#) list the registers for pin interrupt and port control programming.

Table 9-14. Pin Interrupt Registers

Address Offset	Register Name	More Information begins...	Supported Operation	Reset Value
0xFFC01400	PINT0_MASK_SET	on page 9-46	R / W1S	0x00000000
0xFFC01404	PINT0_MASK_CLEAR	on page 9-46	R / W1C	0x00000000

Port Registers

Table 9-14. Pin Interrupt Registers (Cont'd)

Address Offset	Register Name	More Information begins...	Supported Operation	Reset Value
0xFFC01408	PINT0_REQUEST	on page 9-48	R / W1C	0x00000000
0xFFC0140C	PINT0_ASSIGN	on page 9-56	R / W	0x00000101
0xFFC01410	PINT0_EDGE_SET	on page 9-51	R / W1S	0x00000000
0xFFC01414	PINT0_EDGE_CLEAR	on page 9-51	R / W1C	0x00000000
0xFFC01418	PINT0_INVERT_SET	on page 9-54	R / W1S	0x00000000
0xFFC0141C	PINT0_INVERT_CLEAR	on page 9-54	R / W1C	0x00000000
0xFFC01420	PINT0_PINSTATE	on page 9-53	RO	0x00000000
0xFFC01424	PINT0_LATCH	on page 9-48	R / W1C	0x00000000
0xFFC01430	PINT1_MASK_SET	on page 9-46	R / W1S	0x00000000
0xFFC01434	PINT1_MASK_CLEAR	on page 9-46	R / W1C	0x00000000
0xFFC01438	PINT1_REQUEST	on page 9-48	R / W1C	0x00000000
0xFFC0143C	PINT1_ASSIGN	on page 9-56	R / W	0x01010000
0xFFC01440	PINT1_EDGE_SET	on page 9-51	R / W1S	0x00000000
0xFFC01444	PINT1_EDGE_CLEAR	on page 9-51	R / W1C	0x00000000
0xFFC01448	PINT1_INVERT_SET	on page 9-54	R / W1S	0x01010000
0xFFC0144C	PINT1_INVERT_CLEAR	on page 9-54	R / W1C	0x00000000
0xFFC01450	PINT1_PINSTATE	on page 9-53	RO	0x00000000
0xFFC01454	PINT1_LATCH	on page 9-48	R / W1C	0x00000000
0xFFC01460	PINT2_MASK_SET	on page 9-46	R / W1S	0x00000000
0xFFC01464	PINT2_MASK_CLEAR	on page 9-46	R / W1C	0x00000000
0xFFC01468	PINT2_REQUEST	on page 9-48	R / W1C	0x00000000
0xFFC0146C	PINT2_ASSIGN	on page 9-56	R / W	0x00000101
0xFFC01470	PINT2_EDGE_SET	on page 9-51	R / W1S	0x00000000

Table 9-14. Pin Interrupt Registers (Cont'd)

Address Offset	Register Name	More Information begins...	Supported Operation	Reset Value
0xFFC01474	PINT2_EDGE_CLEAR	on page 9-51	R / W1C	0x00000000
0xFFC01478	PINT2_INVERT_SET	on page 9-54	R / W1S	0x00000000
0xFFC0147C	PINT2_INVERT_CLEAR	on page 9-54	R / W1C	0x00000000
0xFFC01480	PINT2_PINSTATE	on page 9-53	RO	0x00000000
0xFFC01484	PINT2_LATCH	on page 9-48	R / W1C	0x00000000
0xFFC01490	PINT3_MASK_SET	on page 9-46	R / W1S	0x00000000
0xFFC01494	PINT3_MASK_CLEAR	on page 9-46	R / W1C	0x00000000
0xFFC01498	PINT3_REQUEST	on page 9-48	R / W1C	0x00000000
0xFFC0149C	PINT3_ASSIGN	on page 9-56	R / W	0x02020303
0xFFC014A0	PINT3_EDGE_SET	on page 9-51	R / W1S	0x00000000
0xFFC014A4	PINT3_EDGE_CLEAR	on page 9-51	R / W1C	0x00000000
0xFFC014A8	PINT3_INVERT_SET	on page 9-54	R / W1S	0x00000000
0xFFC014AC	PINT3_INVERT_CLEAR	on page 9-54	R / W1C	0x00000000
0xFFC014B0	PINT3_PINSTATE	on page 9-53	RO	0x00000000
0xFFC014B4	PINT3_LATCH	on page 9-48	R / W1C	0x00000000

Port Multiplexing Registers

The port multiplexing registers are described in the following sections:

- [“Port x Function Enable \(PORTx_FER\) Registers” on page 9-36](#)
- [“Port Multiplexer Control \(PORTx_MUX\) Registers” on page 9-36](#)

For information on using these registers, see [“Pin Multiplexing Scheme” on page 9-4](#).

Port Registers

Port x Function Enable (PORTx_FER) Registers

After reset, all pins default to GPIO mode (See [Figure 9-7](#)). Setting a bit in the port function enable registers enables a peripheral module to take ownership of the pin. The function enable bits impact output control only. Regardless of the setting of the function enable bits, both GPIO and peripherals can still sense the pin input. Once a function is enabled, it is up to the PORTx_MUX registers as to which peripheral takes control.

Port x Function Enable Registers (PORTx_FER)

R/W

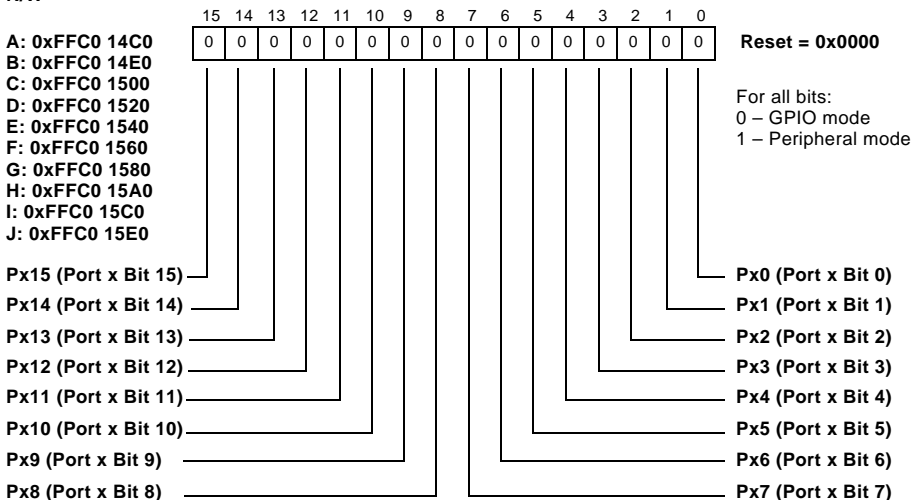


Figure 9-7. Port x Function Enable Registers (PORTx_FER)

Port Multiplexer Control (PORTx_MUX) Registers

The multiplexer controls which peripheral takes ownership of a pin, if not in GPIO mode. Some ports have up to four different functions, while others have just a single function. Two bits are required to describe every multiplexer on an individual pin-by-pin scheme.

As a result, `PORTx_MUX` registers are 32 bits wide. Bit 0 and Bit 1 control the multiplexer of Pin 0. Bit 2 and Bit 3 control the multiplexer of Pin 1. Bit 30 and Bit 31 control the multiplexer of Pin 15.

The value of any `MUXy` bit has no affect on the port pins when the associated `Pxy` bit in the `PORTx_FER` registers is 0. Even if a port has only one function, the `PORTx_MUX` register is still present. For single function ports (no multiplexing is needed), leave the `MUXy` bits at 0 (default).

Normally, the `PORTx_MUX` register is accessed by 32-bit load/store instructions over the PAB bus (See [Figure 9-8](#)). The lower 16 bits can be accessed faster by 16-bit operations, alternately.

Port x Multiplexer Control Registers (PORTx_MUX)

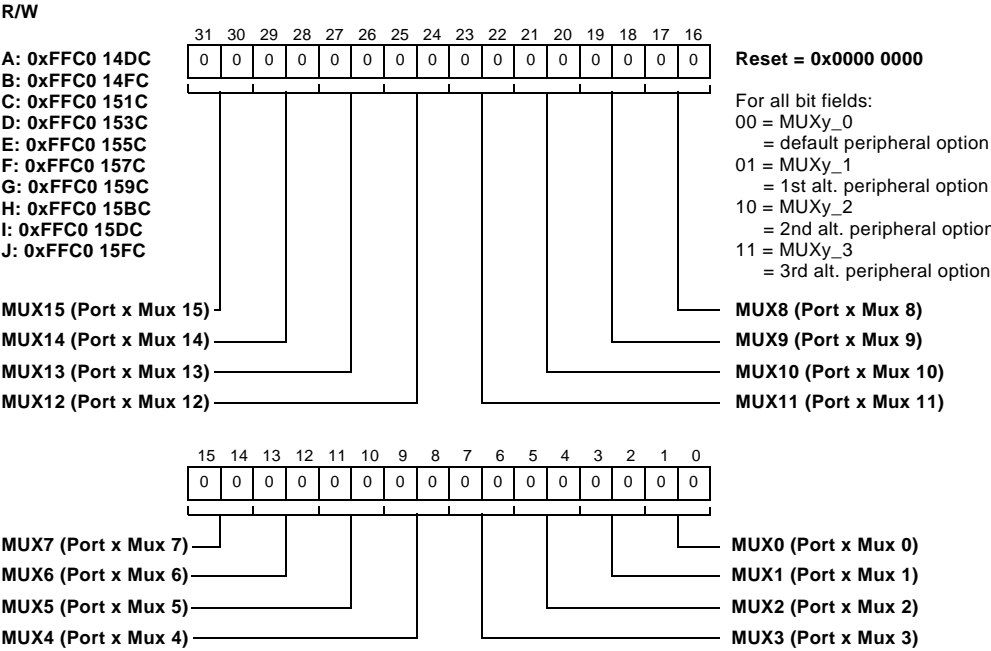


Figure 9-8. Port x Multiplexer Control Registers (PORTx_MUX)

GPIO Registers

The general-purpose I/O registers are described in the following sections.

- “Port x GPIO Direction Set (PORTx_DIR_SET/CLEAR) Register Pairs” on page 9-39
- “Port x GPIO Input Enable (PORTx_INEN) Registers” on page 9-40
- “Port x GPIO Data (PORTx/PORTx_SET/PORTx_CLEAR) Register Groups” on page 9-41

For information on using these registers, see “GPIO Functionality” on page 9-21.

Port x GPIO Direction Set (PORTx_DIR_SET/CLEAR) Register Pairs

The direction registers control the output drivers of the GPIOs (See [Figure 9-9](#) and [Figure 9-10](#)). If set, the output driver is enabled and the GPIO is in output mode. If cleared as by default, the output driver is disabled. Note that the input driver is not enabled by default.

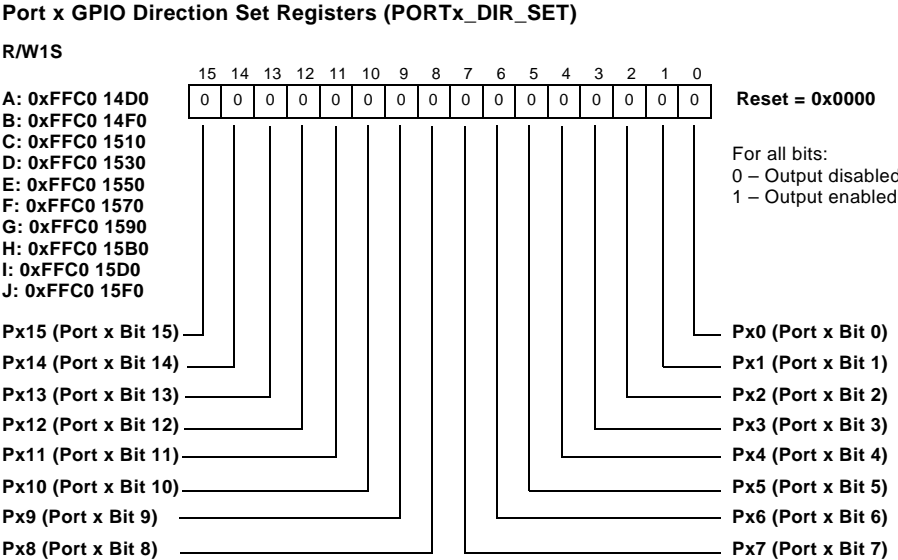


Figure 9-9. Port x GPIO Direction Set Registers (PORTx_DIR_SET)

Port Registers

Port x GPIO Direction Clear Registers (PORTx_DIR_CLEAR)

R/W1C

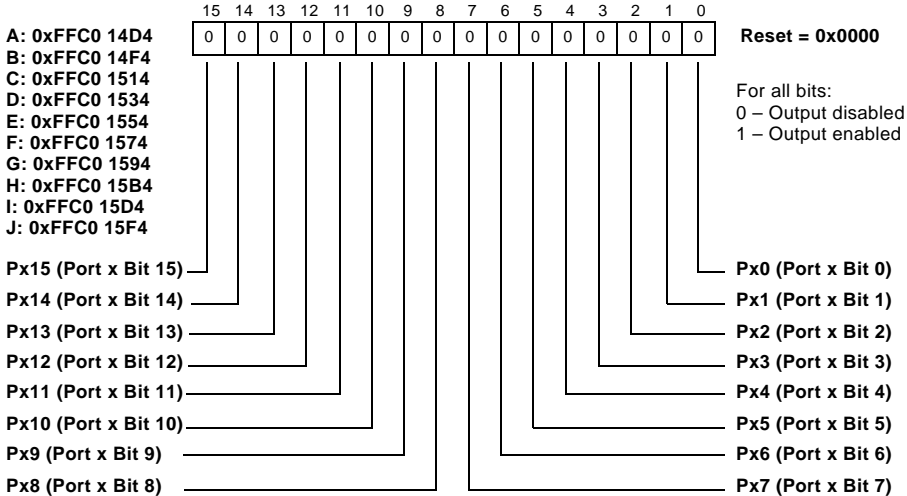


Figure 9-10. Port x GPIO Direction Clear Registers (PORTx_DIR_CLEAR)

Port x GPIO Input Enable (PORTx_INEN) Registers

By default, the input drivers are disabled after reset. To use a pin in GPIO input mode, the input driver must be enabled by writing a “1” to the PORTx_INEN register. If the input is enabled, reads from the PORTx/PORTx_SET/PORTx_CLEAR ports return the state of the pins.

However, the state of the output is not overwritten by the input (See [Figure 9-11](#)). It is altered by software writes only. Input and output drivers can be enabled at the same time. In this case, a read of the data register returns the true value of the data register and not the pin state.

Port x GPIO Input Enable Registers (PORTx_INEN)

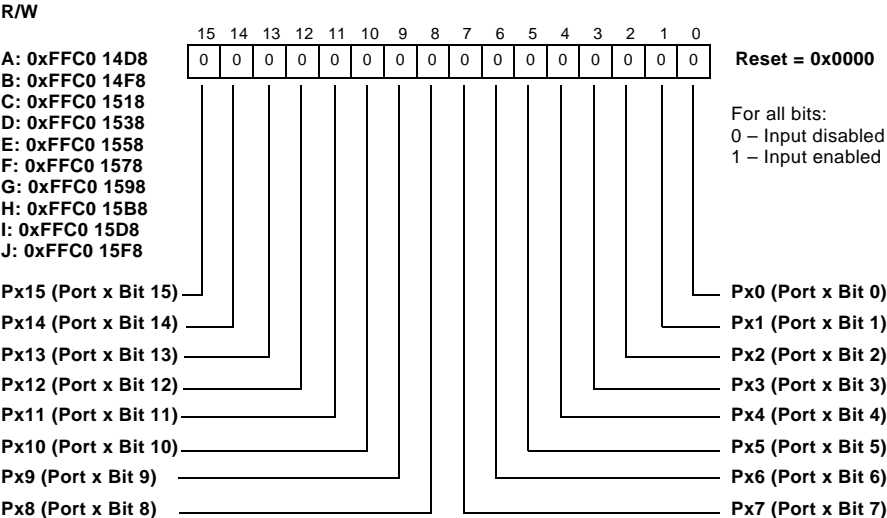


Figure 9-11. Port x GPIO Input Enable Registers (PORTx_INEN)

Port x GPIO Data (PORTx/PORTx_SET/PORTx_CLEAR) Register Groups

This group of registers controls the state of GPIO pins in output mode. Writes to the PORTx register impact the state of all pins of the port that are in output mode, for instance, that have their output driver enabled by the PORTx_DIR_SET and PORTx_DIR_CLEAR registers. The PORTx_SET and PORTx_CLEAR registers enable the software to set or clear specific pins without impacting other pins of the port.

Port Registers

When the input driver is enabled by the `PORTx_INEN` register, reads from any of the three registers return the state of the respective pins (See [Figure 9-12](#) through [Figure 9-14](#)). When the input driver is not enabled as by default, reads from any of the registers return the value previously written to the registers.

Port x GPIO Data Registers (PORTx)

R/W

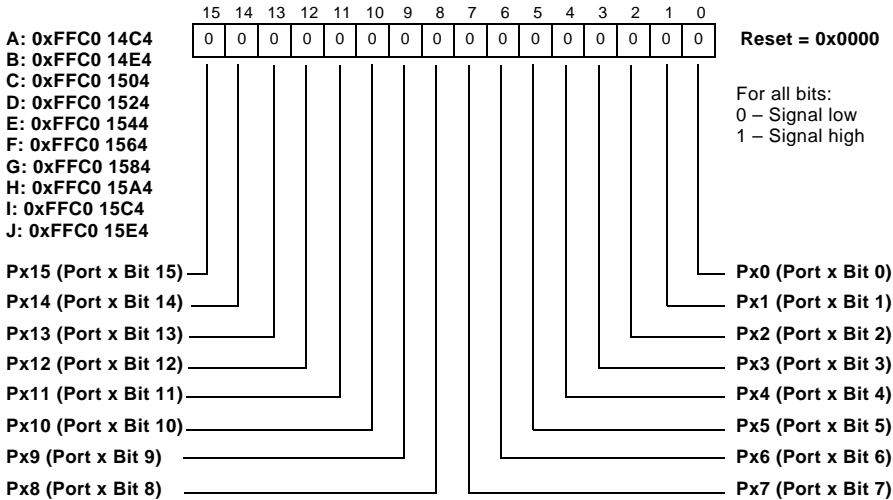


Figure 9-12. Port x GPIO Data Registers (PORTx)

Port x GPIO Data Set Registers (PORTx_SET)

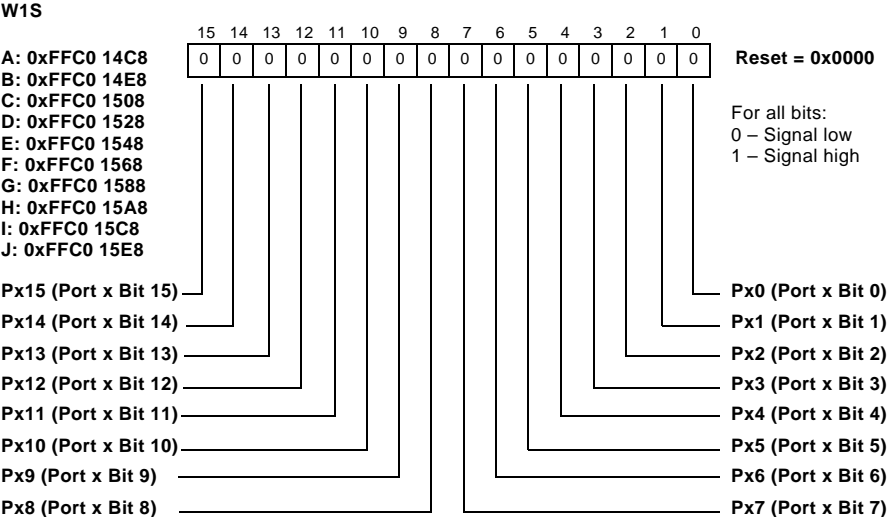


Figure 9-13. Port x GPIO Data Set Registers (PORTx_SET)

Port Registers

Port x GPIO Data Clear Registers (PORTx_CLEAR)

W1C

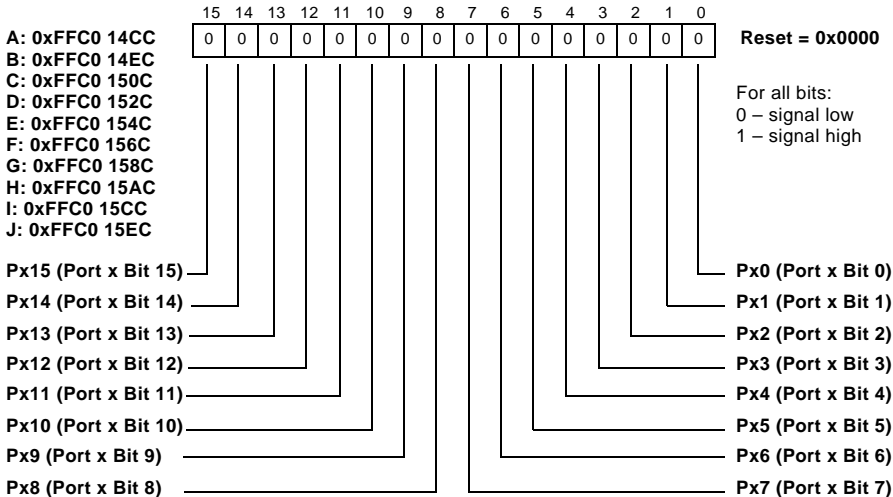


Figure 9-14. Port x GPIO Data Clear Registers (PORTx_CLEAR)

Pin Interrupt Registers

All PINTx registers are 32 bits wide and can be accessed by 32-bit load/store instructions. They also support 16-bit type of operation where the upper 16 bits are ignored and the application uses the lower 16 bits only. Consequently, all PINTx registers support 32-bit PAB accesses as well as 16-bit PAB accesses for the lower half words. Applications may use faster 16-bit accesses as long as they do not require functionality of upper register halves.

The pin interrupt registers are described in the following sections.

- “Pin Interrupt Mask (PINT_x_MASK_SET/PINT_x_MASK_CLEAR) Register Pairs” on page 9-46
- “Interrupt Request and Latch (PINT_x_REQUEST/PINT_x_LATCH) Registers” on page 9-48
- “Interrupt Edge (PINT_x_EDGE_SET/PINT_x_EDGE_CLEAR) Register Pairs” on page 9-51
- “Pin Interrupt Pin State (PINT_x_PINSTATE) Register” on page 9-53
- “Pin Interrupt Invert Set (PINT_x_INVERT_SET/PINT_x_INVERT_CLEAR) Register Pairs” on page 9-54
- “Pin Interrupt Assignment (PINT_x_ASSIGN) Registers” on page 9-56

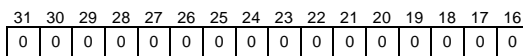
Pin Interrupt Mask (PINTx_MASK_SET/PINTx_MASK_CLEAR) Register Pairs

The pairs of WIS and WIC registers enable interrupt functionality on respective pins (See [Figure 9-15](#) and [Figure 9-16](#)). Setting a bit enables the interrupt. After reset, all bits are cleared. Note that the mask cannot be written directly by the PAB bus (no data register). Masks are controlled by WIS and WIC operations only.

Pin Interrupt Mask Set Registers (PINTx_MASK_SET)

WIS

- 0: 0xFFC01400
- 1: 0xFFC01430
- 2: 0xFFC01460
- 3: 0xFFC01490



Reset = 0x0000 0000

For all bits:
 0 – Interrupt disable
 1 – Interrupt enable

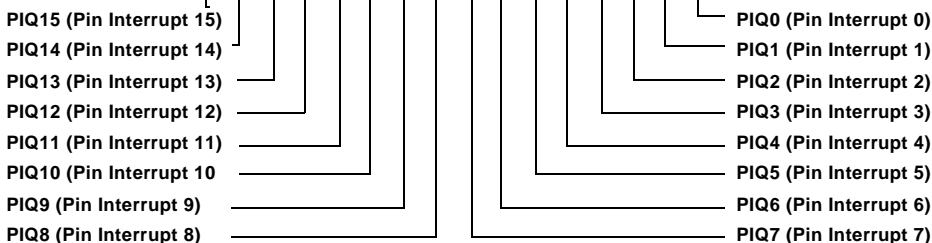
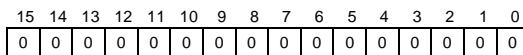
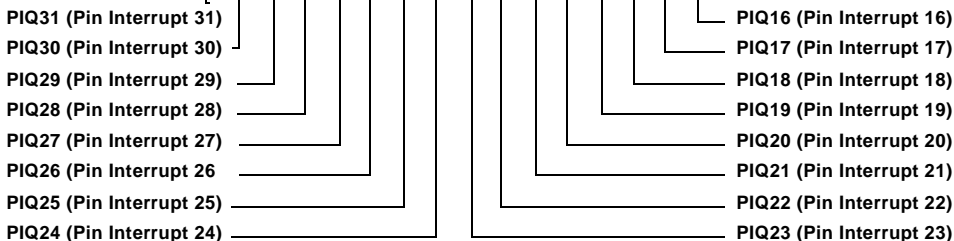
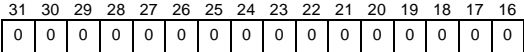


Figure 9-15. Pin Interrupt Mask Set Registers (PINTx_MASK_SET)

Pin Interrupt Mask Clear Registers (PINTx_MASK_CLEAR)

W1C

- 0: 0xFFC01404
- 1: 0xFFC01434
- 2: 0xFFC01464
- 3: 0xFFC01494

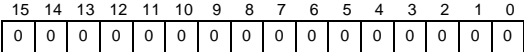


Reset = 0x0000 0000

For all bits:
0 – Interrupt disable
1 – Interrupt enable

- PIQ31 (Pin Interrupt 31)
- PIQ30 (Pin Interrupt 30)
- PIQ29 (Pin Interrupt 29)
- PIQ28 (Pin Interrupt 28)
- PIQ27 (Pin Interrupt 27)
- PIQ26 (Pin Interrupt 26)
- PIQ25 (Pin Interrupt 25)
- PIQ24 (Pin Interrupt 24)

- PIQ16 (Pin Interrupt 16)
- PIQ17 (Pin Interrupt 17)
- PIQ18 (Pin Interrupt 18)
- PIQ19 (Pin Interrupt 19)
- PIQ20 (Pin Interrupt 20)
- PIQ21 (Pin Interrupt 21)
- PIQ22 (Pin Interrupt 22)
- PIQ23 (Pin Interrupt 23)



- PIQ15 (Pin Interrupt 15)
- PIQ14 (Pin Interrupt 14)
- PIQ13 (Pin Interrupt 13)
- PIQ12 (Pin Interrupt 12)
- PIQ11 (Pin Interrupt 11)
- PIQ10 (Pin Interrupt 10)
- PIQ9 (Pin Interrupt 9)
- PIQ8 (Pin Interrupt 8)

- PIQ0 (Pin Interrupt 0)
- PIQ1 (Pin Interrupt 1)
- PIQ2 (Pin Interrupt 2)
- PIQ3 (Pin Interrupt 3)
- PIQ4 (Pin Interrupt 4)
- PIQ5 (Pin Interrupt 5)
- PIQ6 (Pin Interrupt 6)
- PIQ7 (Pin Interrupt 7)

Figure 9-16. Pin Interrupt Mask Clear Registers (PINTx_MASK_CLEAR)

Interrupt Request and Latch (PINTx_REQUEST/PINTx_LATCH) Registers

Both registers indicate whether an interrupt request is latched on the respective pin (See [Figure 9-17](#)). The PINTx_LATCH register is a latch that operates regardless of the interrupt masks. Bits of the PINTx_REQUEST register depend on the mask register. The PINTx_REQUEST register is a logical AND of the PINTx_LATCH register and the interrupt mask.

Pin Interrupt Request Registers (PINTx_REQUEST)

W1C

- 0: 0xFFC01408
- 1: 0xFFC01438
- 2: 0xFFC01468
- 3: 0xFFC01498

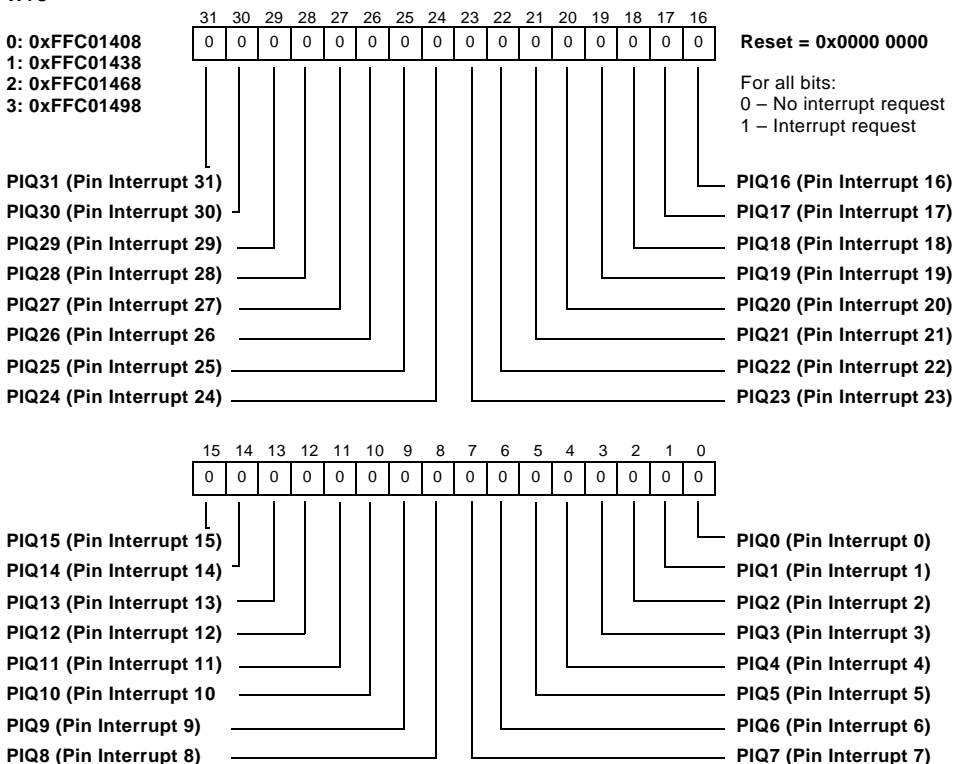


Figure 9-17. Pin Interrupt Request Registers (PINTx_REQUEST)

Having two separate registers here enables the user to interrogate certain pins in polling mode while others work in interrupt mode. The `PINTx_LATCH` registers can be used for edge detection or pin activity detection.

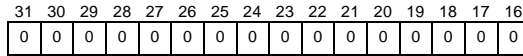
Both registers have W1C behavior (See [Figure 9-18](#)). Writing a 1 to either clears respective bits in both registers. For interrupt operation, the user may prefer to W1C the `PINTx_REQUEST` register (address still loaded in `Px` pointer). In polling mode it might be cleaner to W1C the `PINTx_LATCH` register.

Port Registers

Pin Interrupt Latch Registers (PINTx_LATCH)

W1C

- 0: 0xFFC01424
- 1: 0xFFC01454
- 2: 0xFFC01484
- 3: 0xFFC014B4



Reset = 0x0000 0000

For all bits:
 0 – No interrupt latched
 1 – Interrupt latched

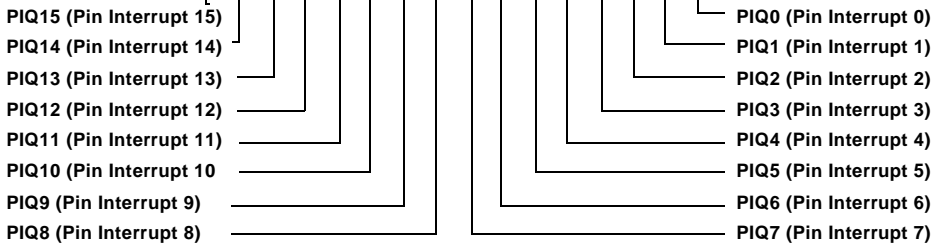
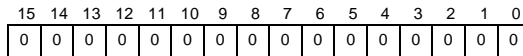
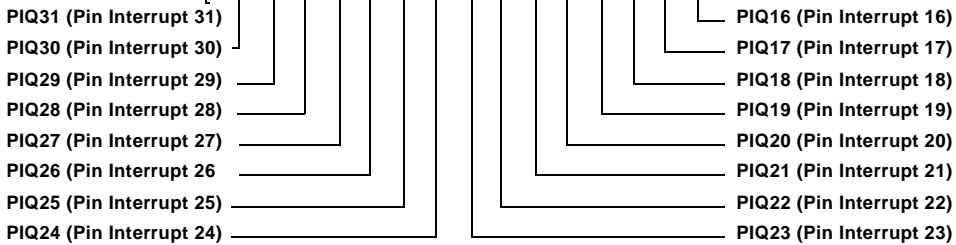


Figure 9-18. Pin Interrupt Latch Registers (PINTx_LATCH)

Regardless whether in edge-sensitive mode or level-sensitive mode, PINTx_LATCH bits are never cleared by hardware except at system reset. Even in level-sensitive mode, the PINTx_LATCH register functions as latch.

Interrupt Edge (PINTx_EDGE_SET/PINTx_EDGE_CLEAR) Register Pairs

This register pair controls whether the individual interrupts are edge-sensitive or level-sensitive (See Figure 9-19). Level sensitivity is default. After a W1S operation to the PINTx_EDGE_SET register, edge sensitivity for the interrupt is enabled.

Pin Interrupt Edge Set Registers (PINTx_EDGE_SET)

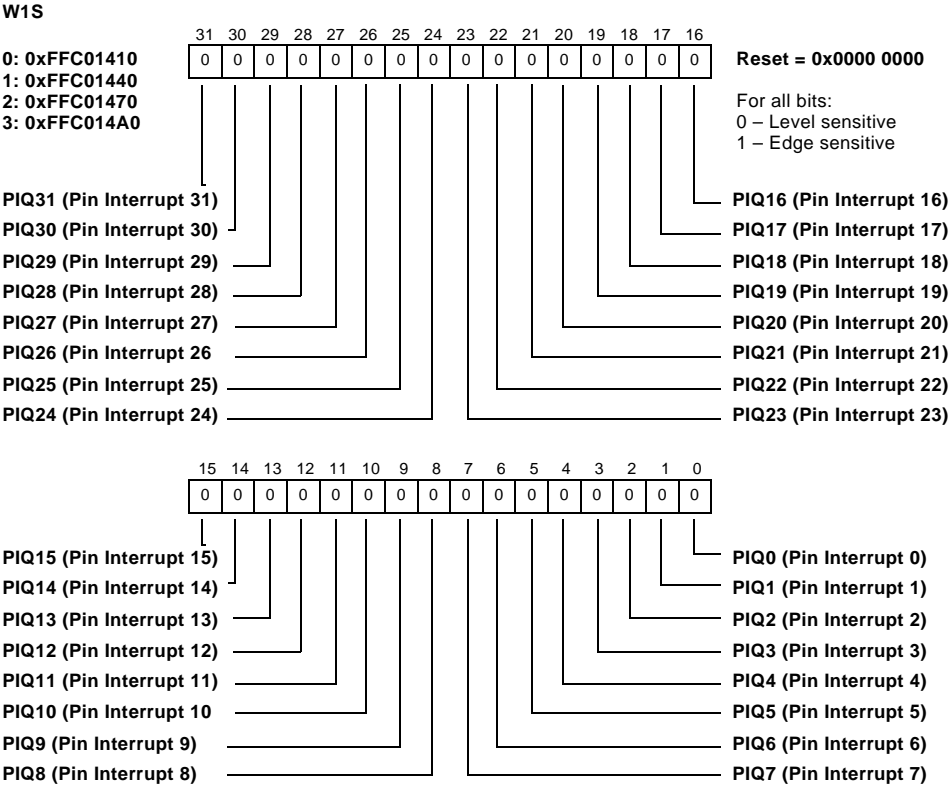


Figure 9-19. Pin Interrupt Edge Set Registers (PINTx_EDGE_SET)

Port Registers

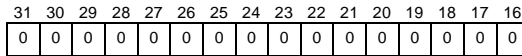
After the W1S operation, the `PINTx_PINSTATE` bits change from logical “0” to 1. (See [Figure 9-20](#))

After a W1C operation to the `PINTx_EDGE_CLEAR` register, edge sensitivity for the interrupt is disabled, and the interrupt returns to level sensitivity.

Pin Interrupt Edge Clear Registers (`PINTx_EDGE_CLEAR`)

W1C

- 0: `0xFFC01414`
- 1: `0xFFC01444`
- 2: `0xFFC01474`
- 3: `0xFFC014A4`



Reset = `0x0000 0000`

For all bits:
 0 – Level sensitive
 1 – Edge sensitive

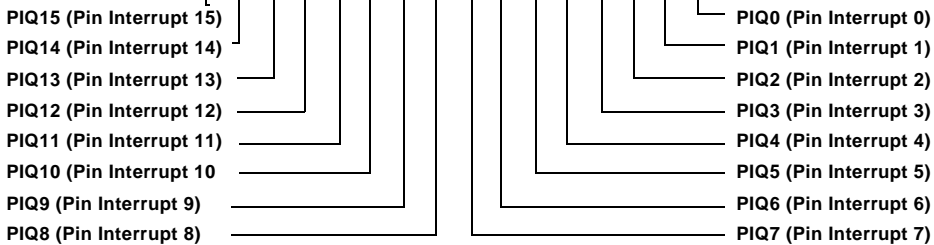
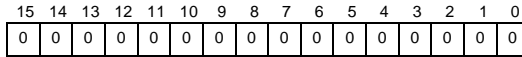
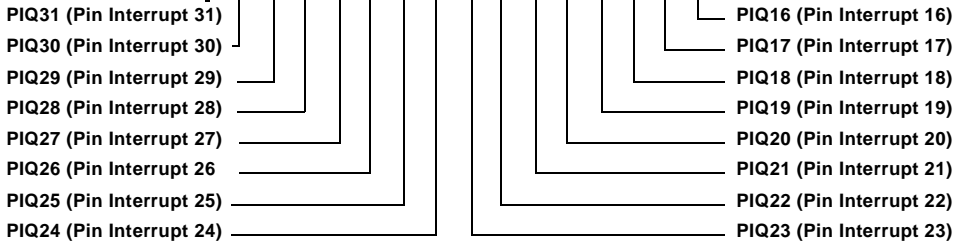


Figure 9-20. Pin Interrupt Edge Clear Registers (`PINTx_EDGE_CLEAR`)

Pin Interrupt Pin State (PINTx_PINSTATE) Register

The pin interrupt pin state registers enable the service routine to read the current state of the pin without reading from GPIO space (See [Figure 9-21](#)). If there was an edge-sensitive interrupt, the service routine can check whether the state of the pin is still high or turned low.

Pin Interrupt Pin State Registers (PINTx_PINSTATE)

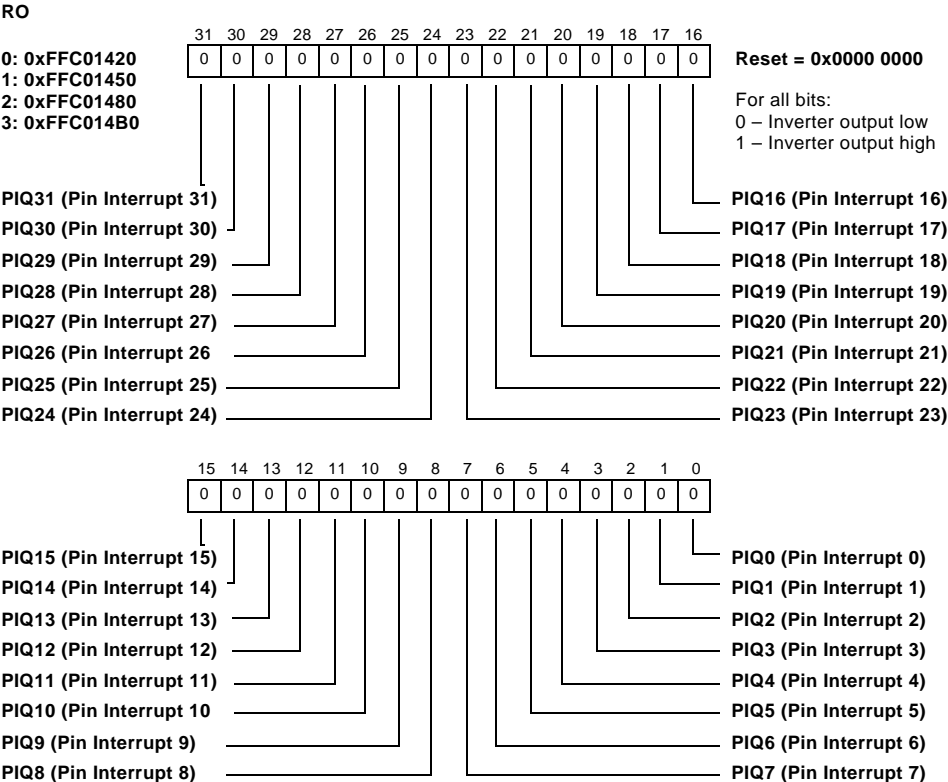


Figure 9-21. Pin Interrupt Pin State Registers (PINTx_PINSTATE)

Note that the content of the PINTx_PINSTATE register depends on the polarity setting of the PINTx_INVERT_SET/PINTx_INVERT_CLEAR registers.

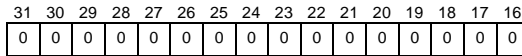
Pin Interrupt Invert Set (PINTx_INVERT_SET/PINTx_INVERT_CLEAR) Register Pairs

These register pairs control the inverters at the input of the module (See [Figure 9-22](#)). After reset, the inverters are cleared and the PINTx_PINSTATE bits contain an exact copy of the pin state. With the inverters on, PINTx_PINSTATE register reads the inverted/negated pin state.

Pin Interrupt Invert Set Registers (PINTx_INVERT_SET)

W1S

- 0: 0xFFC01418
- 1: 0xFFC01448
- 2: 0xFFC01478
- 3: 0xFFC014A8



Reset = 0x0000 0000

For all bits:
0 – Input not inverted
1 – Input inverted

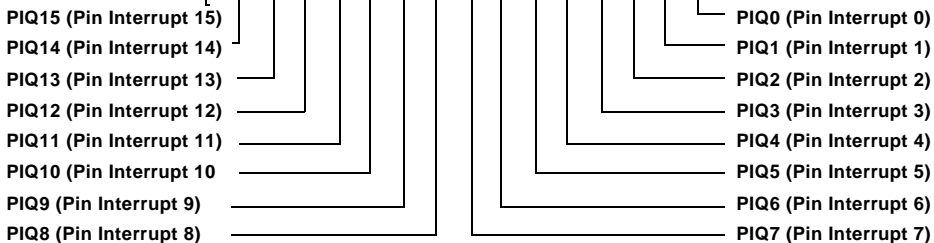
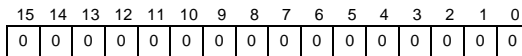
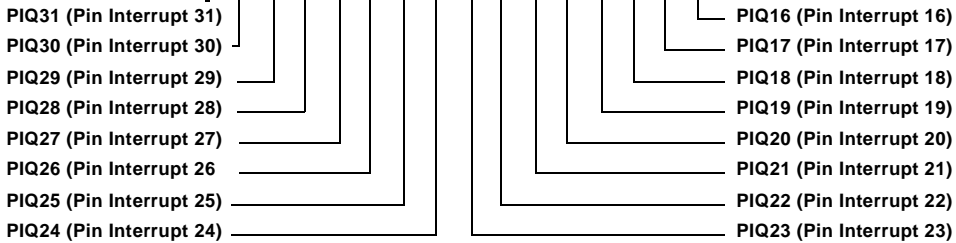


Figure 9-22. Pin Interrupt Invert Set Registers (PINTx_INVERT_SET)

In level-sensitive mode, the interrupt is active when `PINTx_PINSTATE` is logical “1”. For instance, when the pin is high and the inverter is off, or when the pin is low and the inverter is on.

In edge-sensitive mode, the rising edges are latched when the inverter is off (See [Figure 9-23](#)). With the inverter on, falling edges generate the interrupt.

Pin Interrupt Invert Clear Registers (PINTx_INVERT_CLEAR)

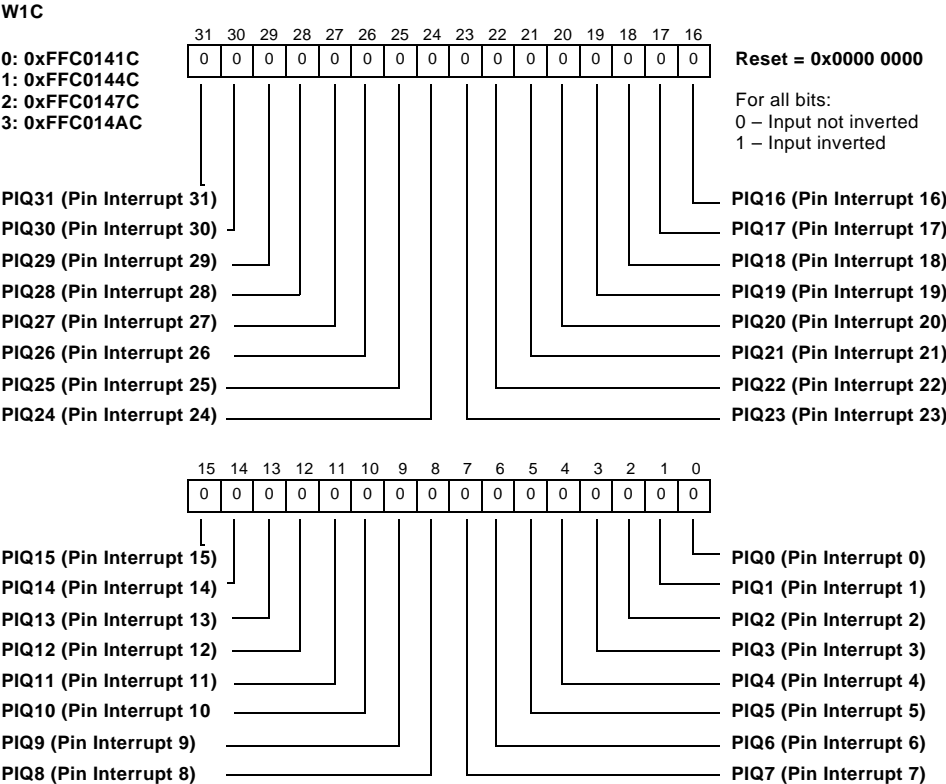


Figure 9-23. Pin Interrupt Invert Clear Registers (PINTx_INVERT_CLEAR)

Pin Interrupt Assignment (PINTx_ASSIGN) Registers

The 32-bit pin interrupt assignment registers control the pin-to-interrupt assignment in a byte-wide manner. Unlike the other pin interrupt registers, the pin interrupt assignment registers do not consist of 32 individual bits. They consist of four control bytes each that function as a multiplexer control.

On ADSP-BF54x Blackfin processors, only three bits of each byte are populated. The other bits are reserved. Both PINT0 and PINT1 blocks can sense to signals of port A and port B. The lower eight pins of port A or port B for example, can be forwarded to either the byte 0 or byte 2 of the pin interrupt registers. Similarly, the upper eight pins can be forwarded to byte 1 or byte 3 of the pin interrupt registers. Both PINT2 and PINT3 blocks can sense to signals of port C to port J. The lower eight pins of any of those ports can be mapped to byte 0 or byte 2 of the pin interrupt registers. Similarly, the upper eight pins of any two ports can be mapped to byte 1 and byte 3.

Figure 9-24 shows the PINT0_ASSIGN register.

Pin Interrupt Assignment Register 0 (PINT0_ASSIGN)

R/W

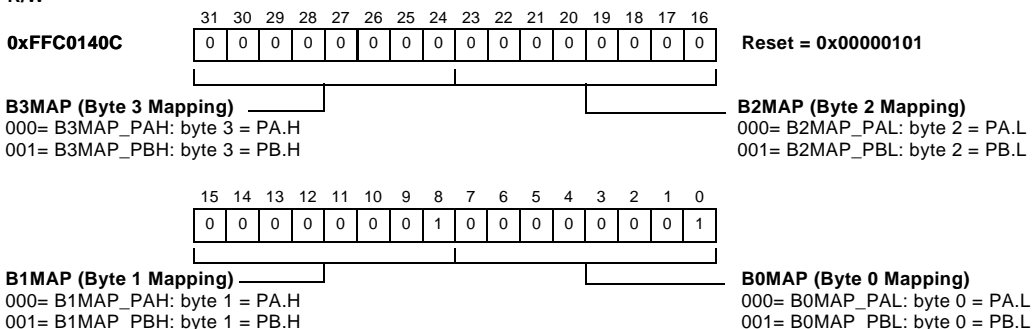


Figure 9-24. Pin Interrupt Assignment Register 0 (PINT0_ASSIGN)

Figure 9-25 shows the PINT1_ASSIGN register.

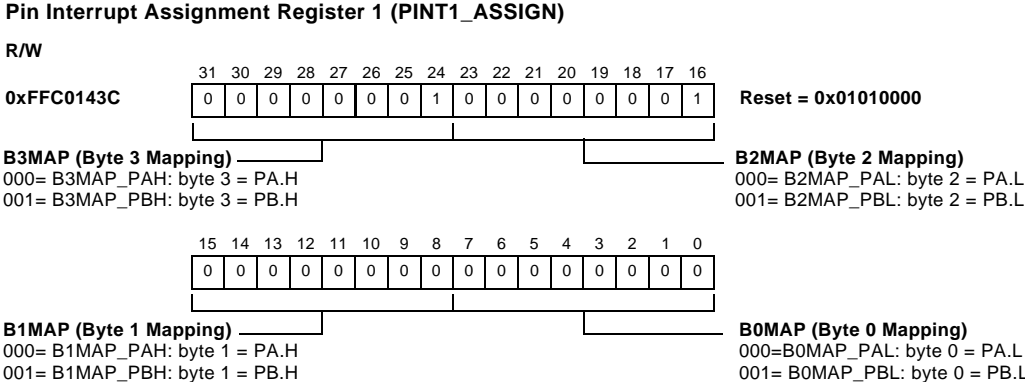


Figure 9-25. Pin Interrupt Assignment Register 1 (PINT1_ASSIGN)

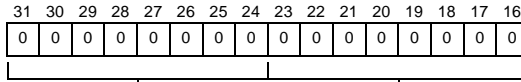
Port Registers

Figure 9-26 shows the PINT2_ASSIGN register.

Pin Interrupt Assignment Register 2 (PINT2_ASSIGN)

R/W

0xFFC0146C



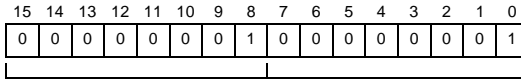
Reset = 0x00000101

B3MAP (Byte 3 Mapping)

- 000= B3MAP_PCH: byte 3 = PC.H
- 001= B3MAP_PDH: byte 3 = PD.H
- 010= B3MAP_PEH: byte 3 = PE.H
- 011= B3MAP_PFH: byte 3 = PF.H
- 100= B3MAP_PGH: byte 3 = PG.H
- 101= B3MAP_PHH: byte 3 = PH.H
- 110= B3MAP_PIH: byte 3 = PI.H
- 111= B3MAP_PJH: byte 3 = PJ.H

B2MAP (Byte 2 Mapping)

- 000= B2MAP_PCL: byte 2 = PC.L
- 001= B2MAP_PDL: byte 2 = PD.L
- 010= B2MAP_PEL: byte 2 = PE.L
- 011= B2MAP_PFL: byte 2 = PF.L
- 100= B2MAP_PGL: byte 2 = PG.L
- 101= B2MAP_PHL: byte 2 = PH.L
- 110= B2MAP_PIL: byte 2 = PI.L
- 111= B2MAP_PJL: byte 2 = PJ.L



B1MAP (Byte 1 Mapping)

- 000= B1MAP_PCH: byte 1 = PC.H
- 001= B1MAP_PDH: byte 1 = PD.H
- 010= B1MAP_PEH: byte 1 = PE.H
- 011= B1MAP_PFH: byte 1 = PF.H
- 100= B1MAP_PGH: byte 1 = PG.H
- 101= B1MAP_PHH: byte 1 = PH.H
- 110= B1MAP_PIH: byte 1 = PI.H
- 111= B1MAP_PJH: byte 1 = PJ.H

B0MAP (Byte 0 Mapping)

- 000= B0MAP_PCL: byte 0 = PC.L
- 001= B0MAP_PDL: byte 0 = PD.L
- 010= B0MAP_PEL: byte 0 = PE.L
- 011= B0MAP_PFL: byte 0 = PF.L
- 100= B0MAP_PGL: byte 0 = PG.L
- 101= B0MAP_PHL: byte 0 = PH.L
- 110= B0MAP_PIL: byte 0 = PI.L
- 111= B0MAP_PJL: byte 0 = PJ.L

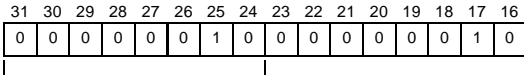
Figure 9-26. Pin Interrupt Assignment Register 2 (PINT2_ASSIGN)

Figure 9-27 shows the PINT3_ASSIGN register.

Pin Interrupt Assignment Register 3 (PINT3_ASSIGN)

R/W

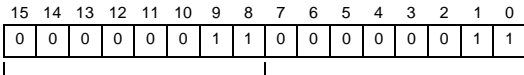
0xFFC0149C



Reset = 0x02020303

- B3MAP (Byte 3 Mapping)**
- 000= B3MAP_PCH: byte 3 = PC.H
 - 001= B3MAP_PDH: byte 3 = PD.H
 - 010= B3MAP_PEH: byte 3 = PE.H
 - 011= B3MAP_PFH: byte 3 = PF.H
 - 100= B3MAP_PGH: byte 3 = PG.H
 - 101= B3MAP_PHH: byte 3 = PH.H
 - 110= B3MAP_PIH: byte 3 = PI.H
 - 111= B3MAP_PJH: byte 3 = PJ.H

- B2MAP (Byte 2 Mapping)**
- 000= B2MAP_PCL: byte 2 = PC.L
 - 001= B2MAP_PDL: byte 2 = PD.L
 - 010= B2MAP_PEL: byte 2 = PE.L
 - 011= B2MAP_PFL: byte 2 = PF.L
 - 100= B2MAP_PGL: byte 2 = PG.L
 - 101= B2MAP_PHL: byte 2 = PH.L
 - 110= B2MAP_PIL: byte 2 = PI.L
 - 111= B2MAP_PJL: byte 2 = PJ.L



- B1MAP (Byte 1 Mapping)**
- 000= B1MAP_PCH: byte 1 = PC.H
 - 001= B1MAP_PDH: byte 1 = PD.H
 - 010= B1MAP_PEH: byte 1 = PE.H
 - 011= B1MAP_PFH: byte 1 = PF.H
 - 100= B1MAP_PGH: byte 1 = PG.H
 - 101= B1MAP_PHH: byte 1 = PH.H
 - 110= B1MAP_PIH: byte 1 = PI.H
 - 111= B1MAP_PJH: byte 1 = PJ.H

- B0MAP (Byte 0 Mapping)**
- 000= B0MAP_PCL: byte 0 = PC.L
 - 001= B0MAP_PDL: byte 0 = PD.L
 - 010= B0MAP_PEL: byte 0 = PE.L
 - 011= B0MAP_PFL: byte 0 = PF.L
 - 100= B0MAP_PGL: byte 0 = PG.L
 - 101= B0MAP_PHL: byte 0 = PH.L
 - 110= B0MAP_PIL: byte 0 = PI.L
 - 111= B0MAP_PJL: byte 0 = PJ.L

Figure 9-27. Pin Interrupt Assignment Register 3 (PINT3_ASSIGN)

Programming Examples

Listing 9-1 illustrates how to enable the output drivers of the port pins PG6 and PG7 on port G. The pins are toggled afterward.

Listing 9-1. Output Driver Enable

```
/* enable GPIO mode. */
/* This is optional as all PORTx_FER register
   are cleared by default after reset */
P5.H = hi(PORTG_FER);
P5.L = lo(PORTG_FER);
R7 = PG7 | PG6 (z);
R6 = ~R7;
R5 = w[P5](z);
R5 = R5 & R6;
w[P5] = R5;

/* start with PG7=0 and PG6=1 */
P5.L = lo(PORTG);
R5 = w[P5] (z);
R5 = R5 & R6;
bitset(R5, bitpos(PG6));
w[P5] = R5;

/* enable output drivers */
P5.L = lo(PORTG_DIR_SET);
w[P5] = R7;

...

/* clear PG6 */
```

```

P5.L = 1o(PORTG_CLEAR);
R5 = PG6;
w[P5] = R5;

/* set PG7 */
P5.L = 1o(PORTG_CLEAR);
R5 = PG7;
w[P5] = R5;

```

Note that the level of the GPIO flags can be defined before the output is enabled. With the separate set and clear ports of the data and direction registers multiple software threads can control their own pins individually.

Listing 9-2 programs the port pin PG8 on port G in open-drain mode. It assumes an external pull-up resistor. Once the PG8 bit is also set in the PORTG_INEN register reads from PORTG register return the actual state of the pin.

Listing 9-2. Open-Drain Mode Programming

```

/* set the internal flag to zero */
P5.H = hi(PORTG_CLEAR);
P5.L = 1o(PORTG_CLEAR);
R5 = PG8 (z);
w[P5] = R5;

/* enable input driver */
P5.L = 1o(PORTG_INEN);
P5.H = hi(PORTG_INEN);
R6 = w[P5] (z);
R6 = R5 | R6;
w[P5] = R6;

/* drive the PG8 pin low */
P5.L = 1o(PORTG_DIR_SET);

```

Programming Examples

```
w[P5] = R5;

...

/* three-state the PG8 pin again */
P5.L = lo(PORTG_DIR_CLEAR);
w[P5] = R5;
```

Listing 9-3 illustrates the pin interrupt functionality. The input pin PB8 in configured to request an IVG6 interrupt through the pin interrupt block PINT0 every time a raising edge is detected.

Listing 9-3. Pin Interrupt Functionality

```
#include <blackfin.h>

.section program;
.global _main;
_main:
    /* register interrupt service routines */
    R7.L = lo(_isr_PB8);
    R7.H = hi(_isr_PB8);
    P5.L = lo(EVT7);
    P5.H = hi(EVT7);
    [P5] = R7;

    /* interrupt assignment PINT0 => IVG7 */
    R7.L = lo(0xFFFF0FFF);
    R7.H = hi(0xFFFF0FFF);
    P5.L = lo(SIC_IAR2);
    P5.H = hi(SIC_IAR2);
    [P5] = R7;

    /* interrupt unmasking */
    R7.L = lo(IRQ_PINT0);
```

```
R7.H = hi(IRQ_PINT0);
P5.L = lo(SIC_IMASK0);
P5.H = hi(SIC_IMASK0);
[P5] = R7;
R7 = EVT_IVG7;
P5.L = lo(IMASK);
P5.H = hi(IMASK);
[P5] = R7;

/* enable input drivers for push-button on Port B */
/* pin can be also output or input enabled by other functions
*/
P5.L = lo(PORTB_INEN);
P5.H = hi(PORTB_INEN);
R6 = w[P5] (z);
R7 = PB8 (z);
R6 = R6 | R7;
w[P5] = R6;

/* assign PB8 to PINT0 byte 1 */
P5.L = lo(PINT0_ASSIGN);
P5.H = hi(PINT0_ASSIGN);
R7.L = lo(B1MAP_PBH);
R7.H = hi(B1MAP_PBH);
[P5] = R7;

/* set to raising edge sensitivity */
R7.L = lo(PB8);
R7.H = hi(PB8);
P5.L = lo(PINT0_INVERT_CLEAR);
P5.H = hi(PINT0_INVERT_CLEAR);
[P5] = R7;
P5.L = lo(PINT0_EDGE_SET);
P5.H = hi(PINT0_EDGE_SET);
```

Programming Examples

```
[P5] = R7;

/* W1C potential latches due to history */
P5.L = lo(PINT0_LATCH);
P5.H = hi(PINT0_LATCH);
[P5] = R7;

/* unmask interrupts */
P5.L = lo(PINT0_MASK_SET);
P5.H = hi(PINT0_MASK_SET);
[P5] = R7;

JUMP 0;
_main.end;
```

Listing 9-4 shows the fragments of an interrupt service routine that matches for **Listing 9-3**. The interrupt request can be cleared by WIC operation to either the `PINTO_REQUEST` or the `PINTO_LATCH` register.

Listing 9-4. Interrupt Service Routine Programming

```

_isr_PB8:
    [--SP] = ASTAT;
    [--SP] = (R7:5, P5:4);

    /* clear interrupt request early in the ISR*/
    P5.L = lo(PINTO_REQUEST);
    P5.H = hi(PINTO_REQUEST);
    R7 = PB8 (z);
    [P5] = R7;

    /* more service code goes to here */

    SSYNC;
    (R7:5, P5:4) = [SP++];
    ASTAT = [SP++];
    RTI;
_isr_PB8.end:

```

Programming Examples

[Listing 9-5](#) provides a C version of [Listing 9-3](#) and [Listing 9-4](#). Additionally, every interrupt event toggles the output on the PF6 GPIO pin.

Listing 9-5. Pin Interrupts and Interrupt Service in C

```
#include <blackfin.h>
#include <ccb1kfn.h>
#include <sys/exception.h>

short dPattern;

/* interrupt service routine */
EX_INTERRUPT_HANDLER(IsrPB8)
{
    /* clear interrupt request */
    *pPINTO_REQUEST = PB8;

    /* toggle output on PG6 */
    if (dPattern & PG6)
    {
        *pPORTG_CLEAR = PG6;
    }
    else
    {
        *pPORTG_SET = PG6;
    }
    dPattern ^= PG6;
}

void main (void)
{
    /* register interrupt routine */
    register_handler(ik_ivg7, IsrPB8);
    /* assign PINTO interrupt to IVG7 */
```



```
*pSIC_IAR2 = 0xFFFF0FFFL;
*pSIC_IMASK0 = IRQ_PINT0;

/* enable the PB8 input driver */
*pPORTB_INEN = PB8;

/* assign PB8 to PINT0 byte 1 */
*pPINT0_ASSIGN = BMAP_PBH;

/* set to raising edge sensitivity */
*pPINT0_INVERT_CLEAR = PB8;
*pPINT0_EDGE_SET = PB8;

/* W1C potential latches due to history */
*pPINT0_LATCH = PB8;

/* unmask interrupts */
*pPINT0_MASK_SET = PB8;

/* initialize PG6 to high */
*pPORTG_SET = PG6;
*pPORTG_DIR_SET = PG6;
dPattern = PG6;

while (1);
}
```

Programming Examples

Listing 9-6 illustrates how to control the port multiplexing. In the example, Port H is configured to provide the following signals: UART1 TX and RX, TMR8, CDG and CUD, DMAR0 and DMAR1 and A4 to A9. PH7 operates in GPIO mode.

Listing 9-6. Port Multiplexing Example

```
P5.H = hi(PORTH_FER);
P5.L = lo(PORTH_FER);
R5.L = PH15 | PH14 | PH13 | PH12 | PH11 | PH10 | PH9 | PH8
      | nPH7 | PH6 | PH5 | PH4 | PH3 | PH2 | PH1 | PH0;
w[P5] = R5;

P5.H = hi(PORTH_MUX);
P5.L = lo(PORTH_MUX);
R5.H = MUX15_0 | MUX14_0 | MUX13_0 | MUX12_0
      | MUX11_0 | MUX10_0 | MUX9_0 | MUX8_0;
R5.L = MUX7_0 | MUX6_1 | MUX5_1 | MUX4_2
      | MUX3_2 | MUX2_1 | MUX1_0 | MUX0_0;
[P5] = R5;

/*For the second part where the PORTH_MUX register is configured,
a more compact syntax can be used as shown below.*/

P5.H=hi(PORTH_MUX);
P5.L=lo(PORTH_MUX);
R5.H=hi(MUX(0,0, 0,0,0,0,0,0, 0, 1,1, 2,2, 1, 0,0));
R5.L=lo(MUX(0,0, 0,0,0,0,0,0, 0, 1,1, 2,2, 1, 0,0));
[P5] = R5;
```

10 GENERAL-PURPOSE TIMERS

This chapter describes the general-purpose timer modules and includes the following sections:

- “Overview and Features” on page 10-1
- “Interface Overview” on page 10-3
- “Description of Operation” on page 10-7
- “Modes of Operation” on page 10-14
- “Programming Model” on page 10-37
- “Timer Registers” on page 10-39
- “Programming Examples” on page 10-58

Overview and Features

The ADSP-BF544, ADSP-BF547, ADSP-BF548, and ADSP-BF549 Blackfin processors feature two general-purpose timer modules that contain eleven identical 32-bit timers. The ADSP-BF542 processors feature only one timer module with eight timers. Every timer can operate in various operating modes on individual configuration. Although the timers operate completely independent from each other, all of them can be started and stopped simultaneously for synchronous operation.

Overview and Features

Features

The general-purpose timers support the following operating modes:

- Singleshot mode for interval timing and single pulse generation
- Pulse-width modulation (PWM) generation with consistent update of period and pulse width values
- External signal capture mode with consistent update of period and pulse width values
- External event counter mode

Feature highlights include:

- Synchronous operation of all timers
- Consistent management of period and pulse width values
- Autobaud detection for CAN and both UART modules
- Period measurement for the GP counter module
- Graceful bit pattern termination when stopping
- Support for center-aligned PWM patterns
- Error detection on implausible pattern values
- All read and write accesses to 32-bit registers are atomic
- Every timer has its dedicated interrupt request output
- Unused timers can function as edge-sensitive pin interrupts

Interface Overview

[Figure 10-1](#) shows the derivative-specific block diagram of the general-purpose timer module.

Interface Overview

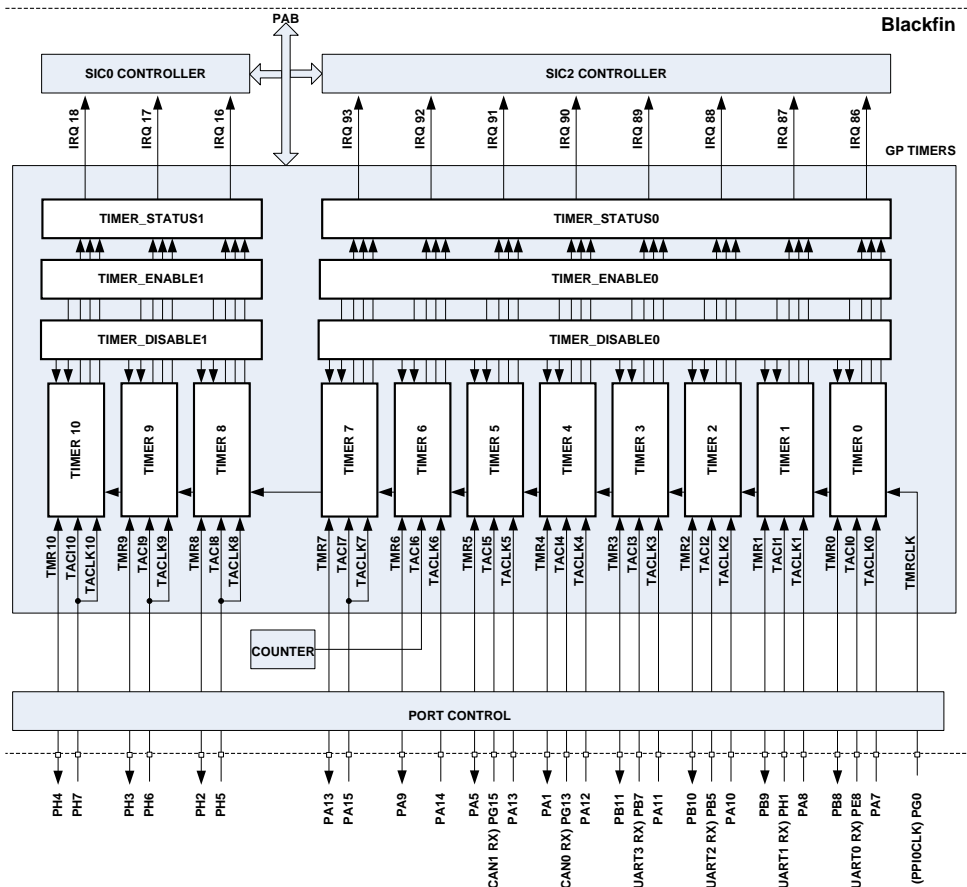


Figure 10-1. Timer Block Diagram

The timer module features a global infrastructure to control synchronous operation of all timers if required. The internal structure of the individual timers is illustrated by [Figure 10-2](#), which shows the details of timer 0 representatively. The other timers have identical structure.

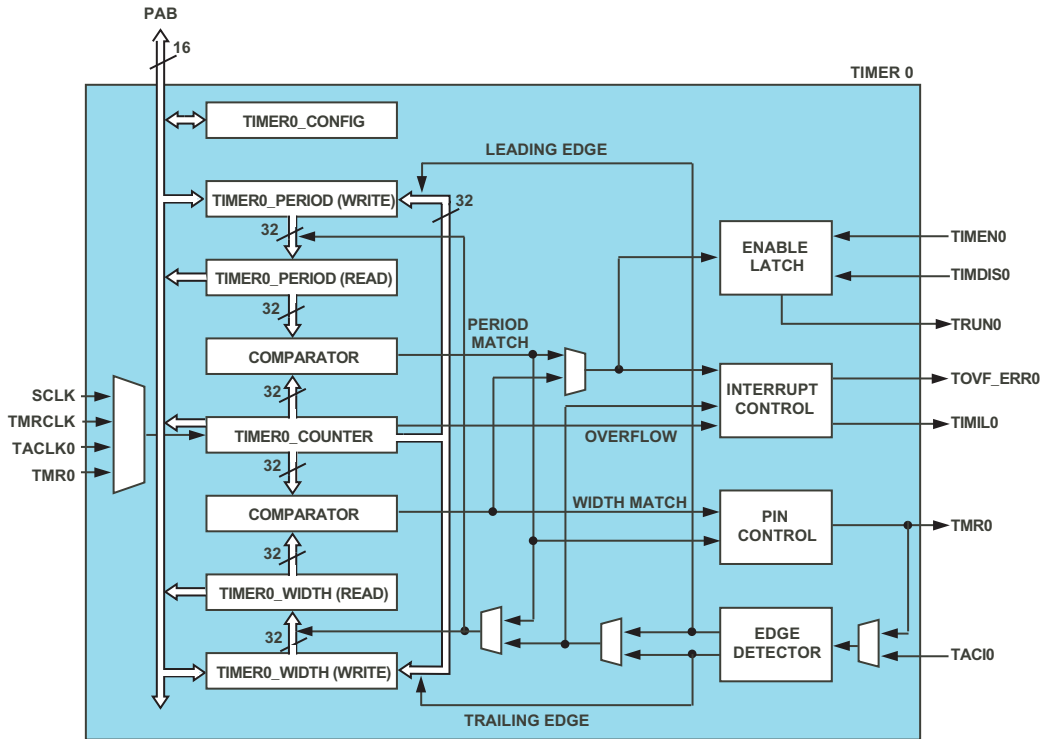


Figure 10-2. Internal Timer Structure

External Interface

Every timer has a dedicated TMR_x pin that can be found on ports A, B, and H. If enabled, the TMR_x pins output the single pulse or PWM signals generated by the timer. They function as input in capture and counter modes. Polarity of the signals is programmable.

Interface Overview

Alternate clock ($TACLK_x$) and capture ($TACI_x$) inputs are found on ports A, B, E, G, and H. The $TACLK_x$ pins can alternatively clock the timers in `PWM_OUT` mode.

In `WDTH_CAP` mode, timers 0-5 feature $TACI_x$ inputs that can be used for bit rate detection on CAN and UART inputs. The $TACI_0$ - $TACI_3$ pins connect, respectively, to the UART0-UART3 RX inputs. Additionally, the $TACI_4$ input connects to the CAN0 RX input, and the $TACI_5$ input connects to the CAN1 RX input. The $TACI_6$ input senses to an output of the general purpose counter module and supports capturing of the event timing this way. The $TACI_7$, $TACI_8$, $TACI_9$ and $TACI_{10}$ inputs are available on pins for various purposes. $TACI_x$ inputs can be used with or without the respective UART or CAN peripheral enabled. If the peripheral is not enabled, the input drivers of the $TACI_x$ inputs must be explicitly enabled.

The $TMRCLK$ input is another clock input common to all 11 timers. The EPPIO unit is clocked by the same pin; therefore any of the timers can be clocked by `EPPIO_CLK`.

In order to enable $TMRCLK$, the `PORTG_FER` bit 0 must be set and input enable for GPIO bit 0 needs to be set in the `PORTG_INEN` register.

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 $((2^{32}-1) / 133 \text{ MHz}) = 32.2$ seconds.

Clock and capture input pins are sampled every `SCLK` cycle. The duration of every low or high state must be one `SCLK` minimum. The maximum allowed frequency of timer input signals is $SCLK/2$.

Internal Interface

Timer registers are always accessed by the core through the 16-bit PAB bus. Hardware ensures that all read and write operations from and to 32-bit timer registers are atomic.

Every timer has its dedicated interrupt request output that connects to the SIC controller, for a total of 11 interrupt outputs.

Description of Operation

The core of every timer is a 32-bit counter, that can be interrogated through the read-only `TIMERx_COUNTER` register. Depending on operation mode, the counter is reset to either `0x0000 0000` or `0x0000 0001` when the timer is enabled. The counter always counts upward. Usually, it is clocked by `SCLK`. In PWM mode it can be clocked by the alternate clock input `TACLKx`, or the common timer clock input `TMRCLK` alternatively. In counter mode, the counter is clocked by edges on the `TMRx` input. The significant edge is programmable.

After $2^{32}-1$ clocks, the counter overflows. In this case, this is reported by the overflow/error bit `TOVF_ERRx` in the `TIMER_STATUSx` registers. In PWM and counter mode, the counter is reset by hardware when its content reaches the values stored in the `TIMERx_PERIOD` register. In capture mode the counter is reset by leading edges on the input pin `TMRx` or `TACIx`. If enabled, these events cause the interrupt latch `TIMILx` in the `TIMER_STATUSx` registers to be set and issue a system interrupt request. The `TOVF_ERRx` and `TIMILx` latches are sticky and should be cleared by software using `WIC` operations to clear the interrupt request. Each global `TIMER_STATUSx` register is 32 bits wide. A single atomic 32-bit read can consistently report the status of all timers within a given `TIMER_STATUSx` register.

Before a timer can be enabled, its mode of operation is programmed in its timer-specific `TIMERx_CONFIG` register. Then, one or more timers are started by writing a 1 to the representative bits in one or more of the `TIMER_ENABLEx` registers.

The `TIMER_ENABLEx` registers can be used to enable some or all timers within a block simultaneously, through “write-1-to-set” control bits, one for each timer. Likewise, the `TIMER_DISABLEx` registers can be used to dis-

Description of Operation

able some or all timers within a block at the same time, through “write-1-to-clear” control bits. The `TIMER_ENABLE0` and `TIMER_DISABLE0` registers control timers 0-7, while the `TIMER_ENABLE1` and `TIMER_DISABLE1` registers control timers 8-10. Either the `TIMER_ENABLEx` or `TIMER_DISABLEx` register for a given timer block 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 `SCLK` cycles after the `TIMENx` bit is set.

While the PWM mode is used to generate PWM patterns, the capture mode (`WDTH_CAP`) is designed to “receive” PWM signals. A PWM pattern is represented by a pulse width and a signal period. This is described by the `TIMERx_WIDTH` and `TIMERx_PERIOD` register pair. In capture mode these registers are read-only. Hardware always captures both values. Regardless of whether in PWM or capture mode, shadow buffers always ensure consistency between the `TIMERx_WIDTH` and `TIMERx_PERIOD` values. In PWM mode, hardware performs a plausibility check by the time the timer is enabled. In this case the error type is reported by the `TIMERx_CONFIG` register and signalled by the `TOVF_ERRx` bit.

Interrupt Processing

Each of the 11 timers can generate a single interrupt. The 11 resulting interrupt signals are routed to the system interrupt controller block for prioritization and masking. The `TIMER_STATUSx` registers latch the timer interrupts to provide a means for software to determine the interrupt source.

To enable interrupt generation, set the `IRQ_ENA` bit and unmask the interrupt source in the `IMASK` and `SIC_IMASKx` registers. To poll the `TIMILx` bit without interrupt generation, set `IRQ_ENA` but leave the interrupt masked at the system level. If enabled by `IRQ_ENA`, interrupt requests are also generated by error conditions as reported by the `TOVF_ERRx` bits.

The system interrupt controller enables flexible interrupt handling. All timers may or may not share the same CEC interrupt channel, so that a single interrupt routine services more than one timer. In PWM mode, multiple 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 (for timers 0-7) by writing `0x000F 000F` to the `TIMER_STATUS0` register.

If interrupts are enabled, make sure that the interrupt service routine (ISR) clears the `TIMILx` bit in the `TIMER_STATUSx` 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 need to be inserted. In `EXT_CLK` mode, reset the `TIMILx` bit in the `TIMER_STATUSx` register at the very beginning of the interrupt service routine to avoid missing any timer events. [Figure 10-3](#) shows the timers interrupt structure.

Description of Operation

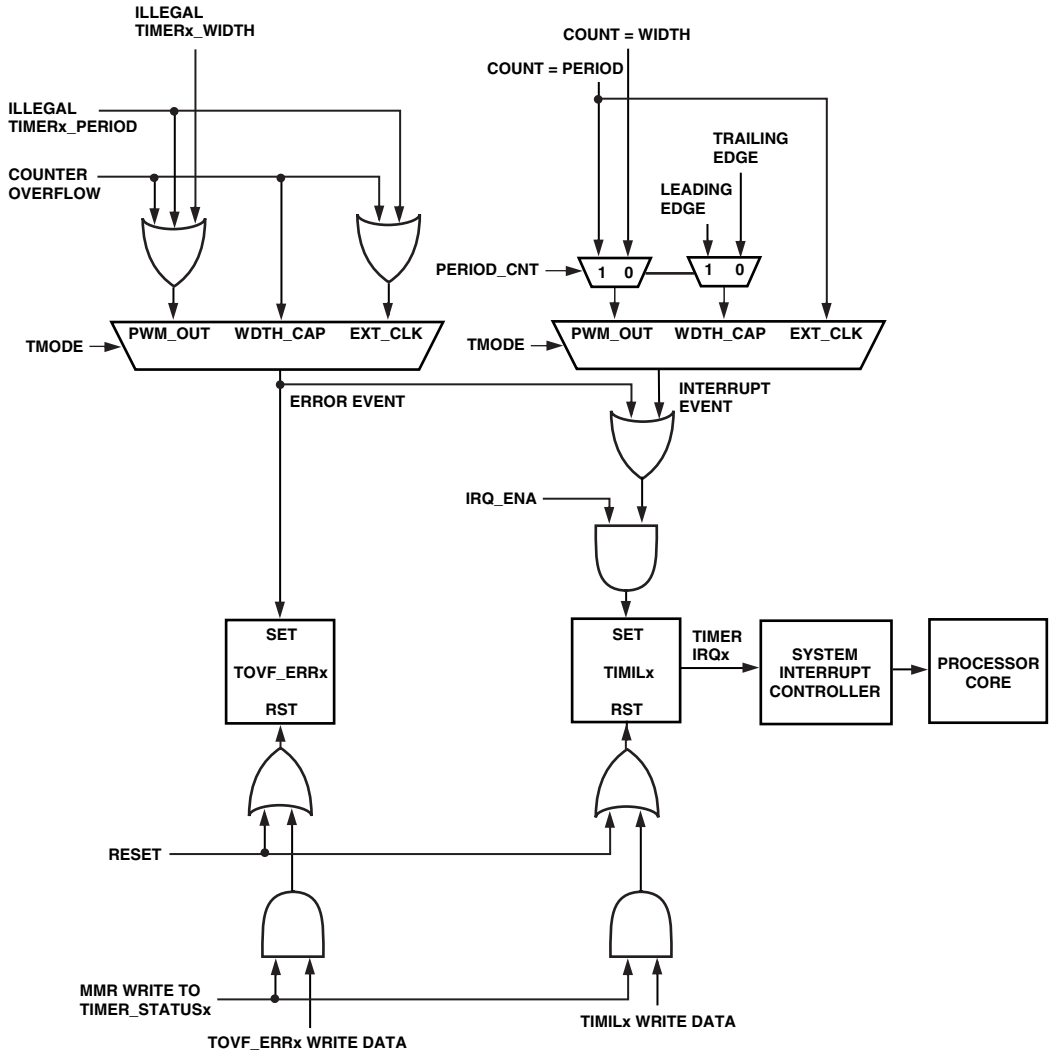


Figure 10-3. Timers Interrupt Structure

Illegal States

Every timer features an error detection circuit. It handles overflow situations but also performs pulse width versus period plausibility checks. Errors are reported by the `TOVF_ERRx` bits in the `TIMER_STATUSx` register and the `ERR_TYP` bit field in the individual `TIMERx_CONFIG` registers.

[Table 10-1](#) provides a summary of error conditions, by using these terms:

- **Startup.** The first clock period when the timer counter is running after the timer is enabled by writing `TIMER_ENABLEx` register.
- **Rollover.** The time when the current count matches the value in `TIMERx_PERIOD` register 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 is no error since this timer was enabled.
 - When `TOVF_ERR` is unchanged, it reads 0 if there is 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.

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.

Description of Operation

Table 10-1 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 10-1. Overview of Illegal States

Mode	Event	<code>TIMERx_PERIOD</code>	<code>TIMERx_WIDTH</code>	<code>ERR_TYP</code>	<code>TOVF_ERR</code>	
PWM_OUT, PERIOD_ CNT = 1	Startup (No boundary condition tests performed on <code>TIMERx_WIDTH</code>)	== 0	Anything	b#10	Set	
		== 1	Anything	b#10	Set	
		>= 2	Anything	Unchanged	Unchanged	
	Rollover		== 0	Anything	b#10	Set
			== 1	Anything	b#11	Set
			>= 2	== 0	b#11	Set
			>= 2	< <code>TIMERx_PERIOD</code>	Unchanged	Unchanged
			>= 2	>= <code>TIMERx_PERIOD</code>	b#11	Set
	Overflow, not possible unless there is also another error, such as <code>TIMERx_PERIOD</code> == 0.	Anything	Anything	b#01	Set	

Table 10-1. Overview of Illegal States (Cont'd)

Mode	Event	TIMERx_ PERIOD	TIMERx_ WIDTH	ERR_TYP	TOVF_ERR
PWM_OUT, PERIOD_ CNT = 0	Startup	Anything	== 0	b#01	Set
		Anything	>= 1	Unchanged	Unchanged
	Rollover	Rollover is not possible in this mode.			
	Overflow, not possible unless there is also another error, such as TIMERx_WIDTH == 0.	Anything	Anything	b#01	Set
WIDTH_CAP	Startup	TIMERx_PERIOD and TIMERx_WIDTH are read-only in this mode, no error possible.			
	Rollover	TIMERx_PERIOD and TIMERx_WIDTH are read-only in this mode, no error possible.			
	Overflow	Anything	Anything	b#01	Set
EXT_CLK	Startup	== 0	Anything	b#10	Set
		>= 1	Anything	Unchanged	Unchanged
	Rollover	== 0	Anything	b#10	Set
		>= 1	Anything	Unchanged	Unchanged
	Overflow, not possible unless there is also another error, such as TIMERx_PERIOD == 0.	Anything	Anything	b#01	Set

Modes of Operation

The following sections provide a functional description of the general-purpose timers in various operating modes.

Pulse Width Modulation (PWM_OUT) Mode

Use the `PWM_OUT` mode for PWM signal or single-pulse generation, for interval timing or for periodic interrupt generation. [Figure 10-4](#) illustrates `PWM_OUT` mode.

Setting the `TMODE` field to `b#01` in the timer configuration (`TIMERx_CONFIG`) register enables `PWM_OUT` mode. Here, the timer `TMRx` pin is an output, but it can be disabled by setting the `OUT_DIS` bit in the `TIMERx_CONFIG` 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`).

Once a timer is enabled, `TIMERx_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

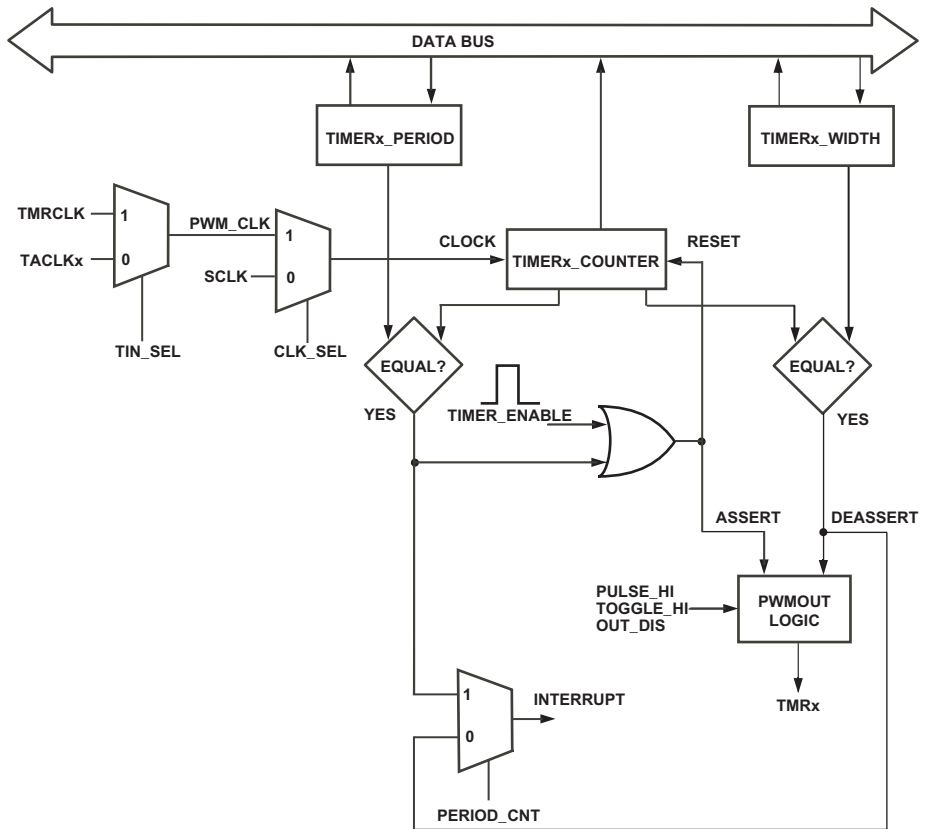


Figure 10-4. Timer Flow Diagram, PWM_OUT Mode

interrupt (if enabled) at the end of each period and stops only after it is disabled. A setting of `PERIOD_CNT = 0` counts to the end of the width; a setting of `PERIOD_CNT = 1` counts to the end of the period.

- i** The `TIMERx_PERIOD` and `TIMERx_WIDTH` registers are read-only in some operation modes. Be sure to set the `TMODE` field in the `TIMERx_CONFIG` register to `b#01` before writing to these registers.

Modes of Operation

Output Pad Disable

The output pin can be disabled in PWM_OUT mode by setting the OUT_DIS bit in the TIMERx_CONFIG register. The TMRx pin is then three-stated regardless of the setting of PULSE_HI and TOGGLE_HI. This can reduce power consumption when the output signal is not being used. The TMRx pin can also be disabled by the PORTx_FER and the PORTx_MUX registers.

Single Pulse Generation

If the PERIOD_CNT bit is cleared, the PWM_OUT mode generates a single pulse on the TMRx pin. This mode can also be used to implement a precise delay. The pulse width is defined by the TIMERx_WIDTH register, and the TIMERx_PERIOD register is not used. See [Figure 10-5](#).

At the end of the pulse, the timer interrupt latch bit TIMILx is set, and the timer is stopped automatically. No writes to the TIMER_DISABLEx register are required in this mode. If the PULSE_HI bit is set, an active high pulse is generated on the TMRx pin. If the PULSE_HI bit is not set, the pulse is active low.

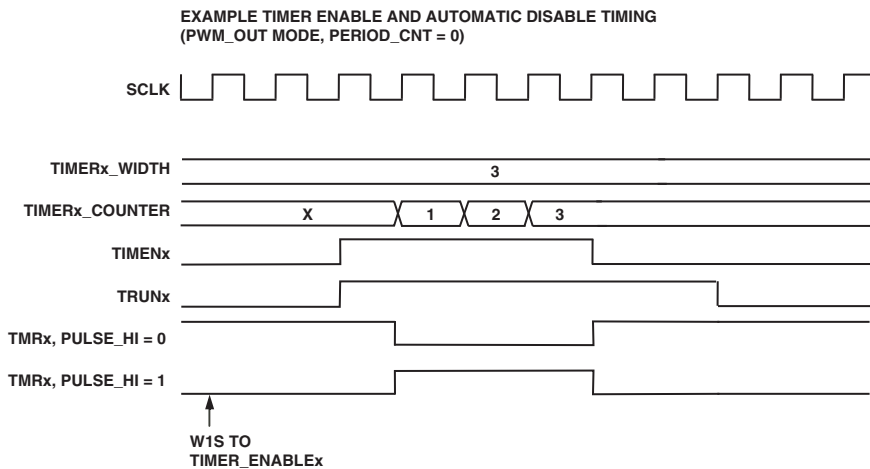


Figure 10-5. Timer Enable and Automatic Disable Timing

The pulse width may be programmed to any value from 1 to $(2^{32}-1)$, inclusive.

Pulse-Width Modulation Waveform Generation

If the `PERIOD_CNT` bit is set, the internally-clocked timer generates rectangular signals with well-defined period and duty cycles (PWM patterns). This mode also generates periodic interrupts for real-time signal processing.

The 32-bit timer period (`TIMERx_PERIOD`) and timer pulse width (`TIMERx_WIDTH`) registers are programmed with the values required by the PWM signal.

When the timer is enabled in this mode, the `TMRx` pin is pulled to a deasserted state each time the counter equals the value of the `TIMERx_WIDTH` register. The pin is asserted again when the period expires (or when the timer gets started).

To control the assertion sense of the `TMRx` pin, the `PULSE_HI` bit in the corresponding `TIMERx_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 `PWM_OUT` mode, the `TMRx` pin is driven to the deasserted level.

Figure 10-6 shows timing details.

Modes of Operation

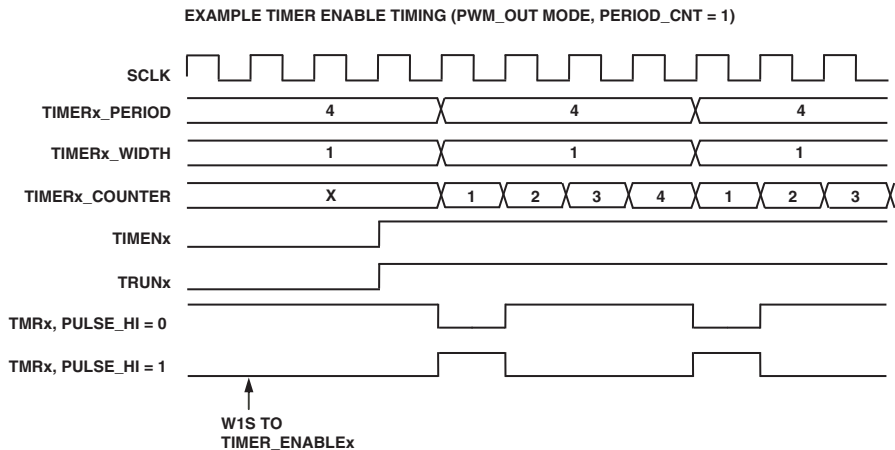


Figure 10-6. Timer Enable Timing

If enabled, a timer interrupt is generated at the end of each period. An interrupt service routine (ISR) must clear the interrupt latch bit (`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. Reads from `TIMERx_PERIOD` and `TIMERx_WIDTH` return the old values until the period expires.

The `TOVF_ERRx` status bit signifies an error condition in `PWM_OUT` mode. The `TOVF_ERRx` bit is set if `TIMERx_PERIOD = 0` or `TIMERx_PERIOD = 1` at startup, or when `TIMERx_COUNTER` rolls over. It is also set if the `TIMERx_WIDTH` register value is greater than or equal to the `TIMERx_PERIOD` register value by the time the counter rolls over. The `ERR_TYP` bits are set when the `TOVF_ERRx` bit is set.

Although the hardware reports an error if the `TIMERx_WIDTH` value equals the `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 `TOVL_ERRx` flags. Pulse width values greater than the period value are not recommended. Similarly, `TIMERx_WIDTH = 0` is not a valid operation. Duty cycles of 0% are not supported.

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.

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 (through 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 10-7](#).

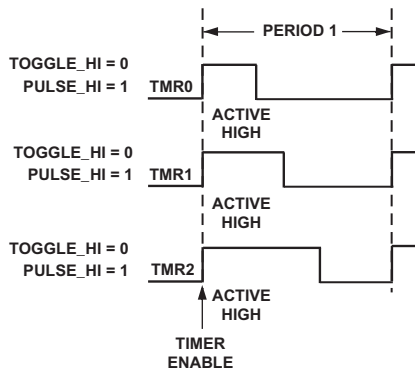


Figure 10-7. Timers With Pulses Aligned to Asserting Edge

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

Modes of Operation

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 the `TOGGLE_HI` bit 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.

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 `Count = 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 10-8 shows an example with three timers running with the same period settings. When software does not alter the PWM settings at run-time, the duty cycle is 50%. The values of the `TIMERx_WIDTH` registers control the phase between the signals.

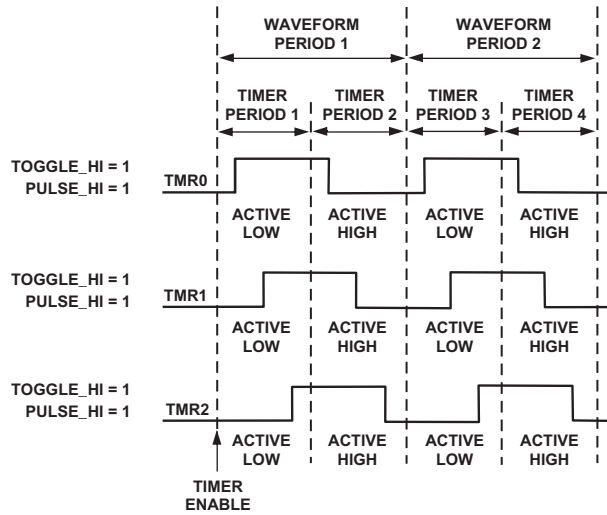


Figure 10-8. Three Timers With Same Period Settings

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 10-9](#)).

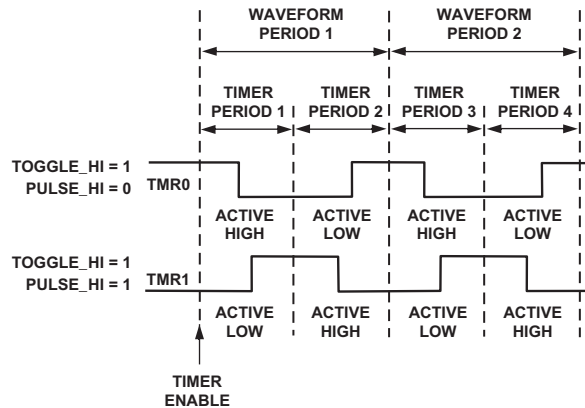


Figure 10-9. Two Timers With Non-Overlapping Clocks

Modes of Operation

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 values are half as large. In odd-numbered periods, write $(\text{Period} - \text{Width})$ instead of `Width` to `TIMERx_WIDTH` in order to obtain center-aligned pulses.

For example, if the pseudo-code when `TOGGLE_HI = 0` is:

```
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:

```
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 ;

    waitfor (interrupt) ;
}
```



```
write(TIMERx_PERIOD, per1) ;
write(TIMERx_WIDTH, per1 - wid1) ;

waitfor (interrupt) ;

write(TIMERx_PERIOD, per2) ;
write(TIMERx_WIDTH, wid2) ;

}
```

As shown in this example, the pulses produced do not need to be symmetric (*wid1* does not need to equal *wid2*). The period can be offset to adjust the phase of the pulses produced (*per1* does not need to equal *per2*).

The timer enable latch (TRUN_x bit in the TIMER_STATUS_x register) is updated only at the end of even-numbered periods in TOGGLE_HI mode. When TIMER_DISABLE_x is written to 1, the current pair of counter periods (one waveform period) completes before the timer is disabled.

As when TOGGLE_HI = 0, errors are reported if the TIMER_x_PERIOD register is either set to 0 or 1, or when the width value is greater than or equal to the period value.

Externally-Clocked PWM_OUT

By default, the timer is clocked internally by SCLK. Alternatively, if the CLK_SEL bit in the timer configuration (TIMER_x_CONFIG) register is set, the timer is clocked by PWM_CLK. The PWM_CLK is normally input from the TACLK_x pin, but may also be taken from the common TMRCLK pin. 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 TIMER_x_WIDTH register.

Modes of Operation

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 alternate timer clock inputs (`TACLKx`) are enabled when a timer is in `PWM_OUT` mode with `CLK_SEL = 1` and `TIN_SEL = 0`, without regard to the content of the `PORTx_MUX` and `PORTx_FER` registers.

Stopping the Timer in `PWM_OUT` Mode

In all `PWM_OUT` mode variants, the timer treats a disable operation (`W1C` to `TIMER_DISABLEx`) 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 `TIMER_STATUSx` register to read `0` or by waiting for the last interrupt (if enabled). Note the timer cannot be reconfigured (`TIMERx_CONFIG` cannot be written to a new value) until after the timer stops and `TRUNx` reads `0`.

In `PWM_OUT` single pulse mode (`PERIOD_CNT = 0`), it is not necessary to write `TIMER_DISABLEx` to stop the timer. At the end of the pulse, the timer stops automatically, the corresponding bit in `TIMER_ENABLEx` (and `TIMER_DISABLEx`) are cleared, and the corresponding `TRUNx` bit is cleared

(See [Figure 10-5 on page 10-16](#)). To generate multiple pulses, write a 1 to `TIMER_ENABLEx`, wait for the timer to stop, then write another 1 to `TIMER_ENABLEx`.

In continuous PWM generation mode (`PWM_OUT`, `PERIOD_CNT = 1`) software can stop the timer by writing to the `TIMER_DISABLEx` register. To prevent the ongoing PWM pattern from being spoiled in unpredictable fashion, the timer does not stop immediately when the corresponding 1 is written to the `TIMER_DISABLEx` register. Rather, the write simply clears the enable latch and the timer still completes the ongoing PWM patterns gracefully. It stops cleanly at the end of the first period when the enable latch is cleared. During this final period the `TIMENx` bit returns 0, but the `TRUNx` bit still reads as a 1.

If the `TRUNx` bit is not cleared explicitly, and the enable latch can be cleared and re-enabled all before the end of the current period, the `TRUNx` bit will continue to run as if nothing happened. Typically, software should disable a `PWM_OUT` timer and then wait for it to stop itself.

[Figure 10-10](#) shows detailed timing.

Modes of Operation

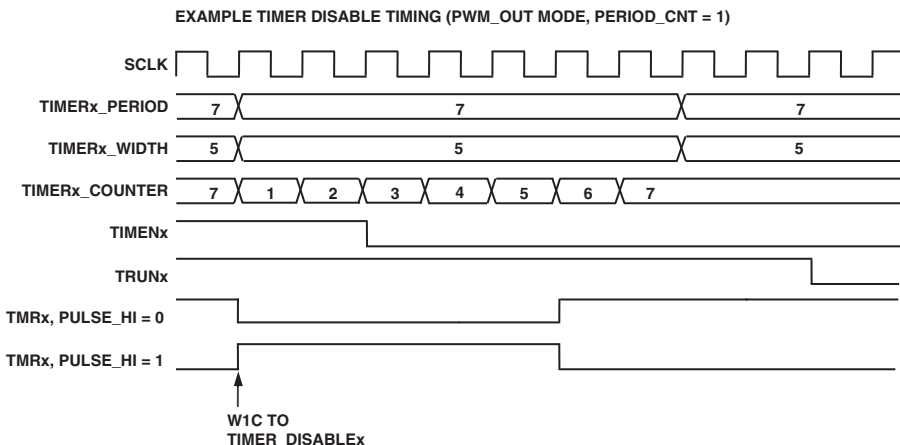


Figure 10-10. Timer Disable Timing

If necessary, the processor can force a timer in PWM_OUT mode to abort immediately. Do this by first writing a 1 to the corresponding bit in `TIMER_DISABLEx`, and then writing a 1 to the corresponding `TRUNx` bit in `TIMER_STATUSx`. 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.

When timers are disabled, the `TIMERx_COUNTER` registers retain their state; when a timer is re-enabled, the timer counter is reinitialized based on the operating mode. The `TIMERx_COUNTER` registers are read-only. Software cannot overwrite or preset the timer counter value directly.

Pulse-Width Count and Capture (WDTH_CAP) Mode

Use the `WDTH_CAP` mode, often simply called “capture mode,” to measure pulse widths on the `TMRx` or `TACIx` input pins, or to “receive” PWM signals. [Figure 10-11](#) shows a flow diagram for `WDTH_CAP` mode.

In `WDTH_CAP` mode, the `TMRx` pin is an input pin. 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 `TIMERx_CONFIG` register enables this mode.

When enabled in this mode, the timer resets the count in the `TIMERx_COUNTER` register to `0x0000 0001` and does not start counting until it detects a leading edge on the `TMRx` pin.

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 `TIMERx_COUNTER` register into the width buffer register. At the next leading edge, the timer transfers the current 32-bit value of the `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 `TMRx` pin, the `PULSE_HI` bit in the `TIMERx_CONFIG` register is set or cleared. If the `PULSE_HI` bit is cleared, the measurement is initiated by a falling edge, the content of the `TIMERx_COUNTER` is captured to the pulse width buffer on the rising edge, and to the period buffer on the next falling edge. When the `PULSE_HI` bit is set, the measurement is initiated by a rising edge, the counter value is captured to the pulse width buffer on the falling edge, and to the period buffer on the next rising edge.

Modes of Operation

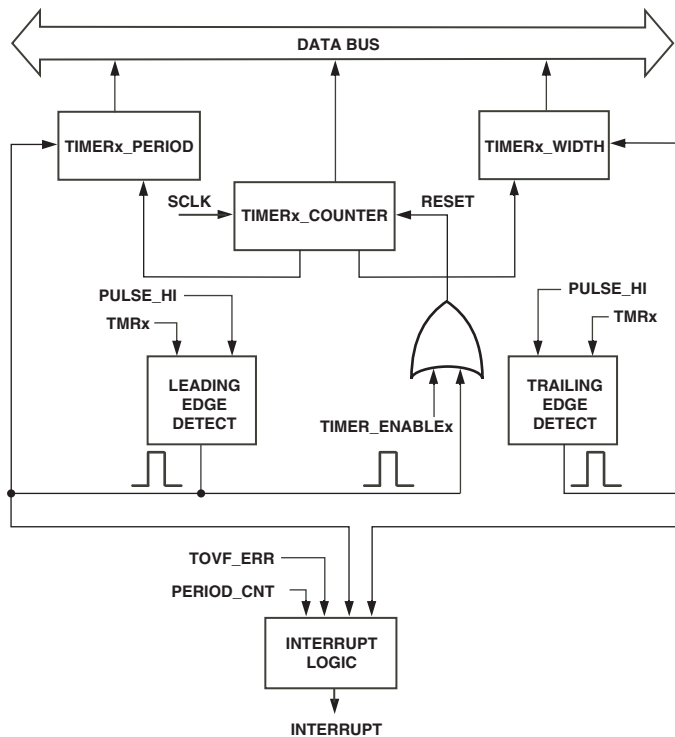


Figure 10-11. Timer Flow Diagram, WIDTH_CAP Mode

In WIDTH_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 `TIMILx` bit gets set (if enabled) but does not generate an error.

The `PERIOD_CNT` bit in the `TIMERx_CONFIG` register controls the point in time when this set of transactions is executed. Taken together, these three events are called a measurement report. The `TOVF_ERRx` bit does not get set at a measurement report. A measurement report occurs once per input signal period (at most).

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).

If the `PERIOD_CNT` bit is set and a leading edge occurred (see [Figure 10-12](#)), 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 10-13](#)), 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 10-13](#). 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

Modes of Operation

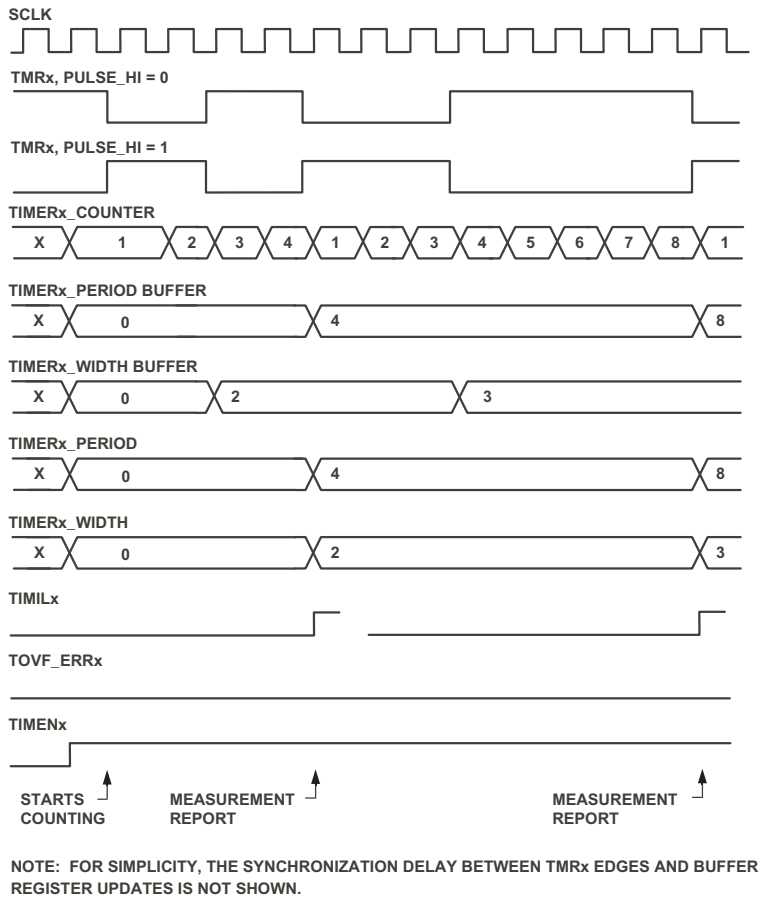
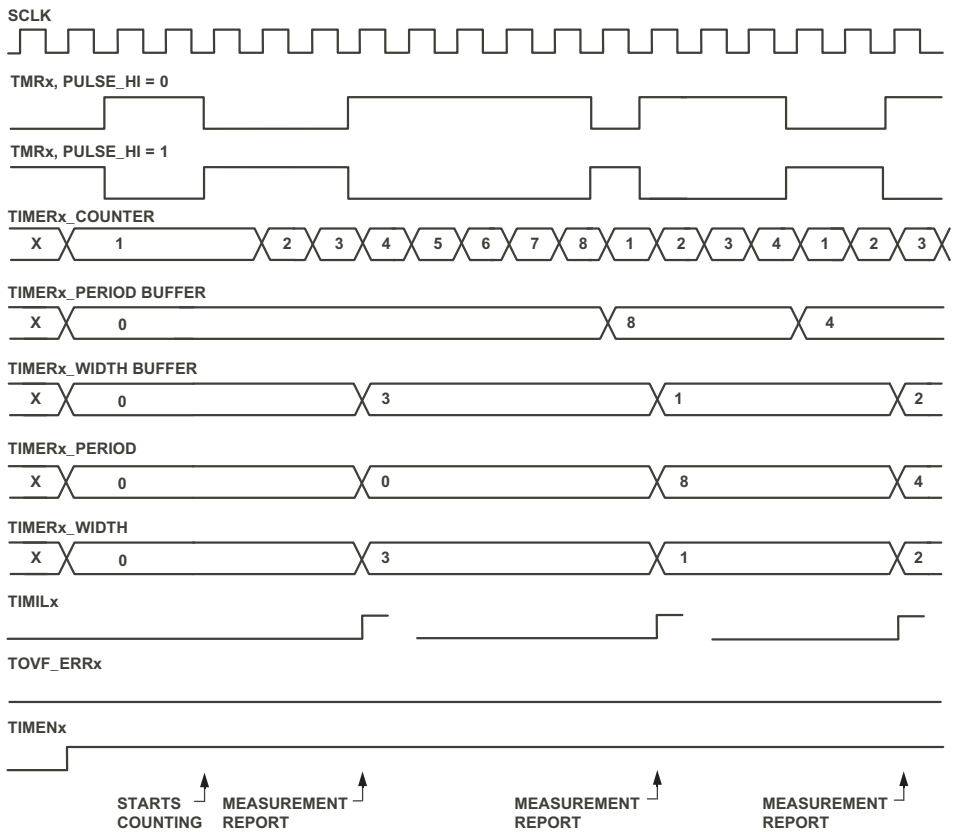


Figure 10-12. Example of Period Capture Measurement Report Timing (WDTM Mode, PERIOD_CNT = 1)



NOTE: FOR SIMPLICITY, THE SYNCHRONIZATION DELAY BETWEEN TMRx EDGES AND BUFFER REGISTER UPDATES IS NOT SHOWN.

Figure 10-13. Example of Width Capture Measurement Report Timing (WIDTH_CAP Mode, PERIOD_CNT = 0)

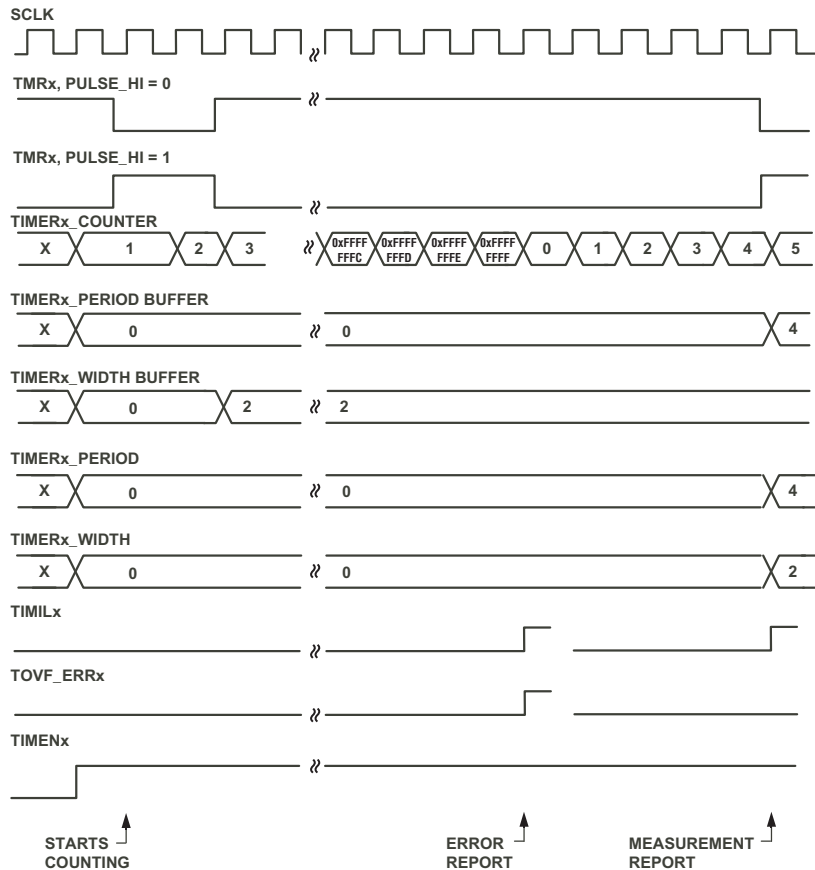
Modes of Operation

(because no measurement report occurred to copy the value captured in the width buffer register to `TIMERx_WIDTH`). See the first interrupt in [Figure 10-14](#).



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 reenabled 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.

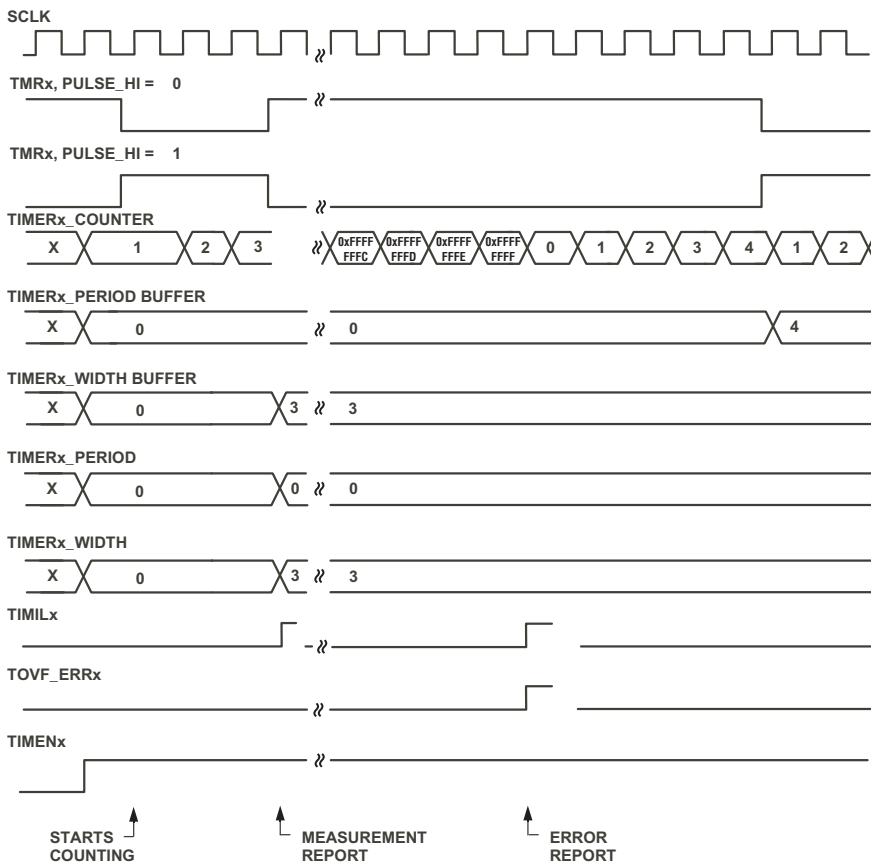
A timer interrupt (if enabled) is generated if `TIMERx_COUNTER` wraps around from `0xFFFF FFFF` to `0` in the absence of a leading edge. At that point, the `TOVF_ERRx` bit in the `TIMER_STATUSx` register and the `ERR_TYP` bits in the `TIMERx_CONFIG` register are set, indicating a count overflow due to 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 10-14](#) and [Figure 10-15](#) for more information.



NOTE: FOR SIMPLICITY, THE SYNCHRONIZATION DELAY BETWEEN TMRx EDGES AND BUFFER REGISTER UPDATES IS NOT SHOWN.

Figure 10-14. Example Timing for Period Overflow Followed by Period Capture (WDTM_CAP Mode, PERIOD_CNT = 1)

Modes of Operation



NOTE: FOR SIMPLICITY, THE SYNCHRONIZATION DELAY BETWEEN TMRx EDGES AND BUFFER REGISTER UPDATES IS NOT SHOWN.

Figure 10-15. Example Timing for Width Capture Followed by Period Overflow (WDTH_CAP Mode, PERIOD_CNT = 0)

Both TIMILx and TOVF_ERRx are sticky bits, and software has to explicitly clear them. If the timer overflowed and PERIOD_CNT = 1, neither the TIMERx_PERIOD nor the TIMERx_WIDTH register were updated. If the timer

overflowed and `PERIOD_CNT = 0`, the `TIMERx_PERIOD` and `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} `SCLK` counts to the total for the period, but the width is ambiguous. For example, in [Figure 10-14](#) 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 `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` with a 50% duty cycle. Under these conditions, the `WDTH_CAP` mode timer would measure `Period = 2` and `Pulse Width = 1`.

Autobaud Mode

In `WDTH_CAP` mode, some of the timers can provide autobaud detection for the universal asynchronous receiver/transmitter (UART) and controller area network (CAN) interfaces. The timer input select (`TIN_SEL`) bit in the `TIMERx_CONFIG` register causes the timer to sample the `TACIx` pin instead of the `TMRx` pin, when enabled for `WDTH_CAP` mode. Autobaud detection can be used for initial bit rate negotiations as well as for detection of bit rate drifts while the interface is in operation. For details with the UART interface, see the “UART Port Controllers” chapter in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 2 of 2)*. For details with the CAN interface, see the “CAN Module” chapter in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 2 of 2)*.

Modes of Operation

Capturing Timings from the GP Counter Module

In `WDTH_CAP` mode, one of the timers can sense to an internal signal of the GP counter module through the `TACI6` input. This enables the timer to capture the period between counter events. For details, see [“Capturing Timing Information \(Using the General-Purpose Timer\)”](#) on page 13-18.

External Event (`EXT_CLK`) Mode

Use the `EXT_CLK` mode, sometimes referred to as the “counter mode,” to count external events, that is, signal edges on the `TMRx` pin which is an input in this mode. [Figure 10-16](#) shows a flow diagram for `EXT_CLK` mode.

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.

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.

After the timer is enabled, it resets `TIMERx_COUNTER` to `0x0` and then waits for the first leading edge on the `TMRx` pin. This edge causes `TIMERx_COUNTER` to be incremented 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 `TIMERx_COUNTER` 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).

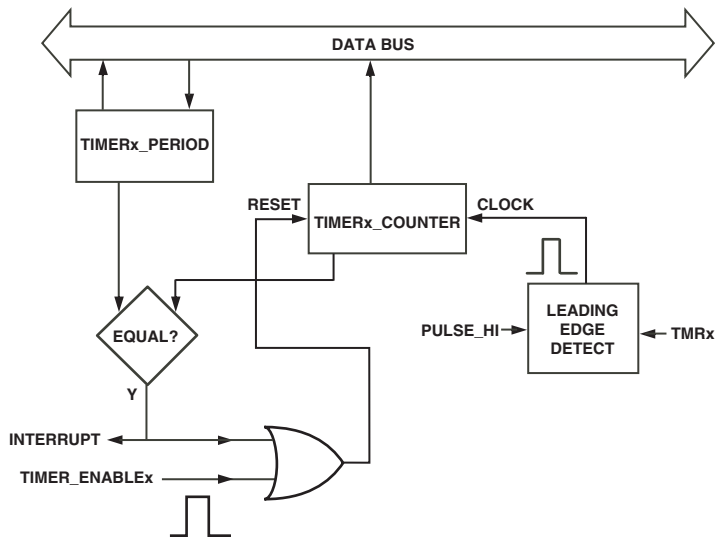


Figure 10-16. Timer Flow Diagram, EXT_CLK Mode

The configuration bits, `TIN_SEL` and `PERIOD_CNT`, have no effect in this mode. The `TOVF_ERRx` and `ERR_TYP` bits are set if `TIMERx_COUNTER` wraps around from `0xFFFF FFFF` to `0` or if `Period = 0` at startup, or when `TIMERx_COUNTER` rolls over (from `Count = Period` to `Count = 0x1`). `TIMERx_WIDTH` is unused.

Programming Model

The architecture of the timer block enables any timer to work individually or synchronously along with others in its group. That is, timers 0-7 are members of the same group, and timers 8-10 are members of a separate

Programming Model

group. Regardless of the operation mode, the timers' programming model is always straightforward. Because of the error-checking mechanism, always follow this order when enabling timers:

1. Set timer mode.
2. Write `TIMERx_WIDTH` and `TIMERx_PERIOD` registers as applicable.
3. Enable timer.

If this order is not followed, the plausibility check may fail because of undefined width and period values, or writes to `TIMERx_WIDTH` and `TIMERx_PERIOD` may result in an error condition, because the registers are read-only in some modes. Accordingly, the timer may not start as expected.

If in `PWM_OUT` mode the PWM patterns of the second period differ from the patterns of the first one, the initialization sequence above might become:

1. Set timer mode to `PWM_OUT`.
2. Write first `TIMERx_WIDTH` and `TIMERx_PERIOD` value pair.
3. Enable timer.
4. Immediately write second `TIMERx_WIDTH` and `TIMERx_PERIOD` value pair.

Hardware ensures that the buffered width and period values become active when the first period expires.

Once started, timers require minimal interaction with software, which is usually performed by an interrupt service routine. In `PWM_OUT` mode software must update the pulse width and/or settings as required. In `WIDTH_CAP` mode it must store captured values for further processing. In any case, the service routine should clear the `TIMILx` bits of the timers it controls.

Timer Registers

The timer peripheral module provides general-purpose timer functionality. It consists of 11 identical timer units.

Each timer provides four registers:

- `TIMERx_CONFIG[15:0]` – timer configuration register (on page 10-47)
- `TIMERx_WIDTH[31:0]` – timer pulse width register (on page 10-52)
- `TIMERx_PERIOD[31:0]` – timer period register (on page 10-52)
- `TIMERx_COUNTER[31:0]` – timer counter register (on page 10-49)

Additionally, three register sets are shared between the 11 timers:

- `TIMER_ENABLEx[15:0]` – timer enable registers (on page 10-40)
- `TIMER_DISABLEx[15:0]` – timer disable registers (on page 10-41)
- `TIMER_STATUSx[31:0]` – timer status registers (on page 10-43)

`TIMER_ENABLE0`, `TIMER_DISABLE0`, and `TIMER_STATUS0` control timers 0-7. `TIMER_ENABLE1`, `TIMER_DISABLE1`, and `TIMER_STATUS1` control timers 8-10.

The size of accesses is enforced. A 32-bit access to a `TIMERx_CONFIG` register or a 16-bit access to a `TIMERx_WIDTH`, `TIMERx_PERIOD`, or `TIMERx_COUNTER` register results in a memory-mapped register (MMR) error. Both 16- and 32-bit accesses are allowed for the `TIMER_ENABLEx`, `TIMER_DISABLEx`, and `TIMER_STATUSx` registers. On a 32-bit read of one of the 16-bit registers, the upper word returns all 0s.

Table 10-6 on page 10-56 summarizes control bit and register usage in each timer mode.

Timer Enable (TIMER_ENABLEx) Registers

The `TIMER_ENABLEx` registers, shown in [Figure 10-17](#) and [Figure 10-18](#), allow all timers within a group to be enabled simultaneously in order to make them run completely synchronously. For each timer there is a single `WIS` control bit. Writing a 1 enables the corresponding timer; writing a 0 has no effect. The bits can be set individually or in any combination. A read of the `TIMER_ENABLEx` register shows the status of the enable for the corresponding timers within a group. A 1 indicates that the timer is enabled. All unused bits return 0 when read.

Timer Enable 0 Register (TIMER_ENABLE0)

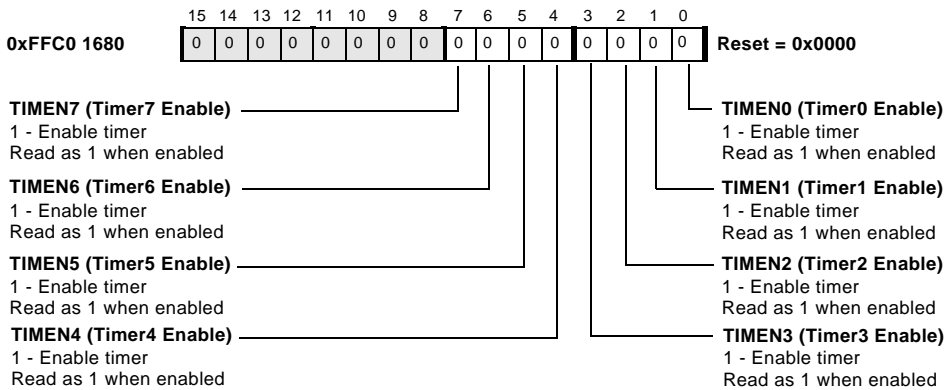


Figure 10-17. Timer Enable 0 (TIMER_ENABLE0) Register

Timer Enable 1 Register (TIMER_ENABLE1)

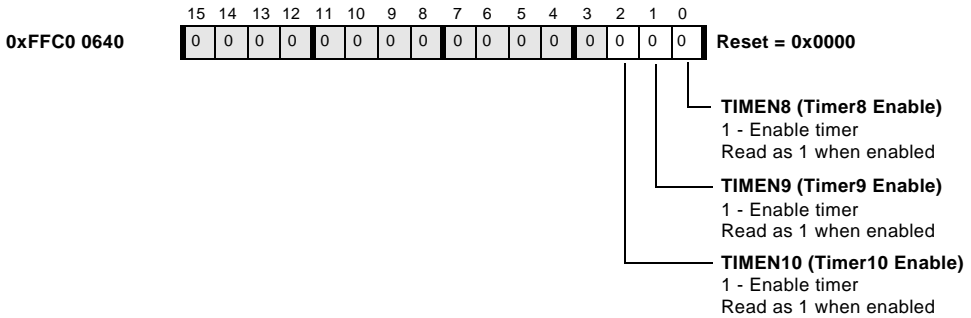


Figure 10-18. Timer Enable 1 (TIMER_ENABLE1) Register

Timer Disable (TIMER_DISABLEx) Registers

The `TIMER_DISABLEx` registers, shown in [Figure 10-19](#) and [Figure 10-20](#) allow all timers within a group to be disabled simultaneously. For each timer there is a single W1C control bit. Writing a 1 disables the corresponding timer; writing a 0 has no effect. The bits within a disable register can be cleared individually or in any combination. A read of the `TIMER_DISABLEx` register returns a value identical to a read of the corresponding `TIMER_ENABLEx` register. A 1 indicates that the timer is enabled. All unused bits return 0 when read.

Timer Registers

Timer Disable 0 Register (TIMER_DISABLE0)

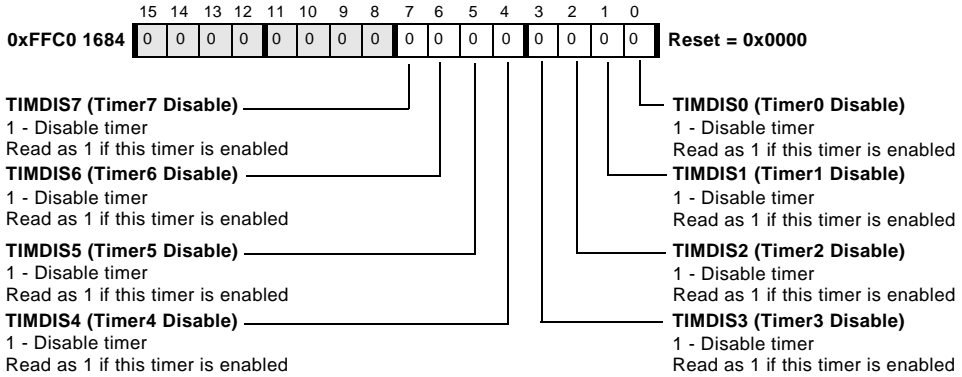


Figure 10-19. Timer Disable 0 (TIMER_DISABLE0) Register

Timer Disable 1 Register (TIMER_DISABLE1)

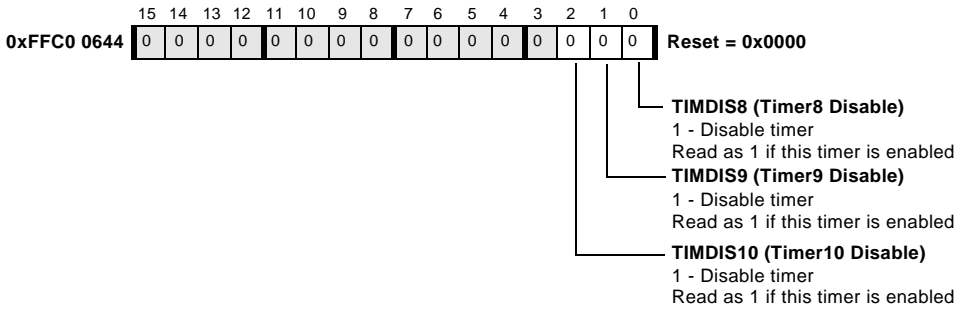


Figure 10-20. Timer Disable 1 (TIMER_DISABLE1) Register

In PWM_OUT mode, a write of a 1 to `TIMER_DISABLEx` does not stop the corresponding 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 corresponding bit in `TIMER_DISABLEx`, and then writing a 1 to the corresponding `TRUNx` bit in `TIMER_STATUSx`. See “Stopping the Timer in PWM_OUT Mode” on page 10-24.

In `WDTH_CAP` and `EXT_CLK` modes, a write of a 1 to `TIMER_DISABLEx` stops the corresponding timer immediately.

Timer Status (TIMER_STATUSx) Registers

The `TIMER_STATUSx` registers are used to check the status of all timers within a group with a single read (see Figure 10-21 and Figure 10-22). 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 `TIMER_STATUSx` register read access, all reserved or unused bits return a 0.

Timer Registers

For detailed behavior and usage of the `TRUNx` bit see [“Stopping the Timer in PWM_OUT Mode” on page 10-24](#). 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.

Error conditions are explained in [“Illegal States” on page 10-11](#).

Timer Status Register 0 (TIMER_STATUS0)

All bits are W1C

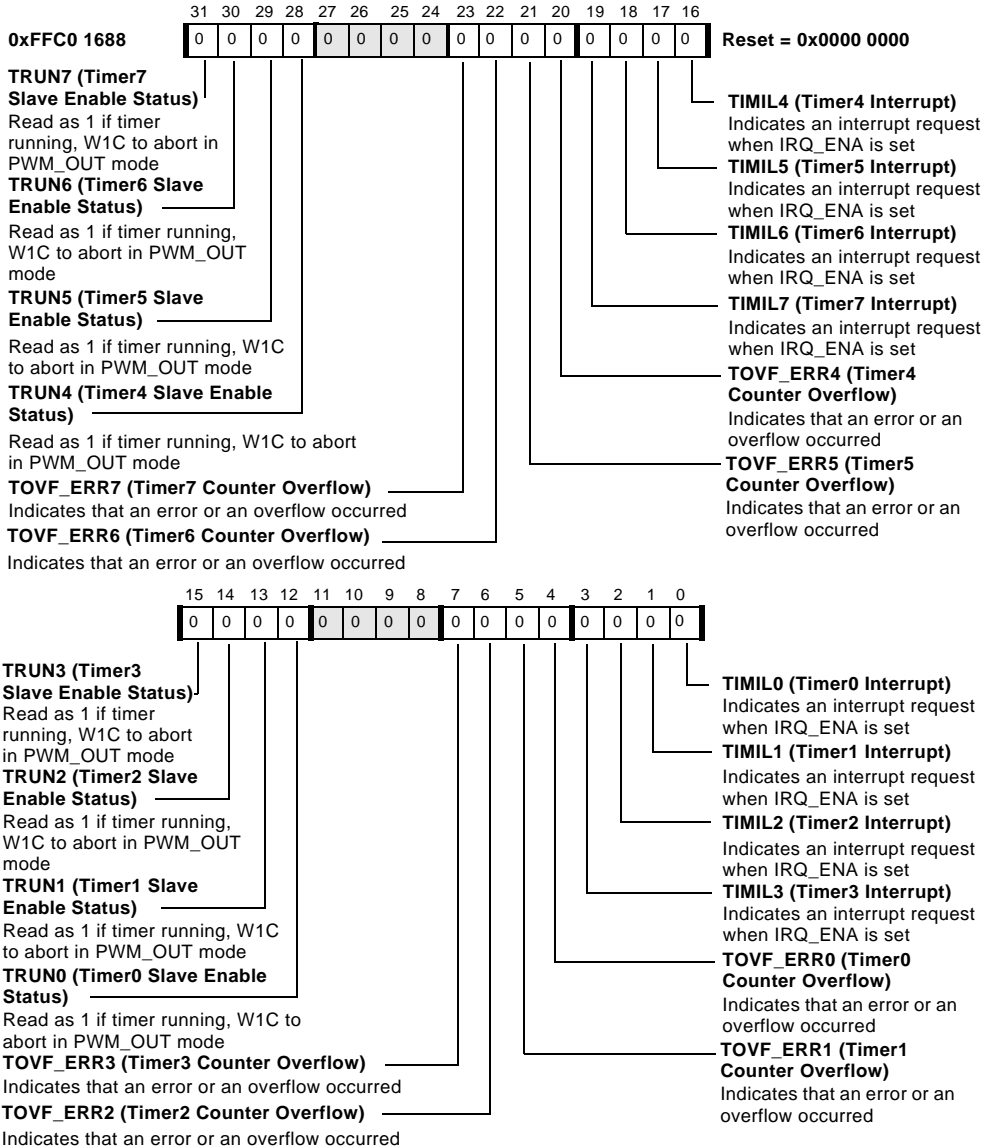


Figure 10-21. Timer Status 0 (TIMER_STATUS0) Register

Timer Registers

Timer Status Register 1 (TIMER_STATUS1)

All bits are W1C

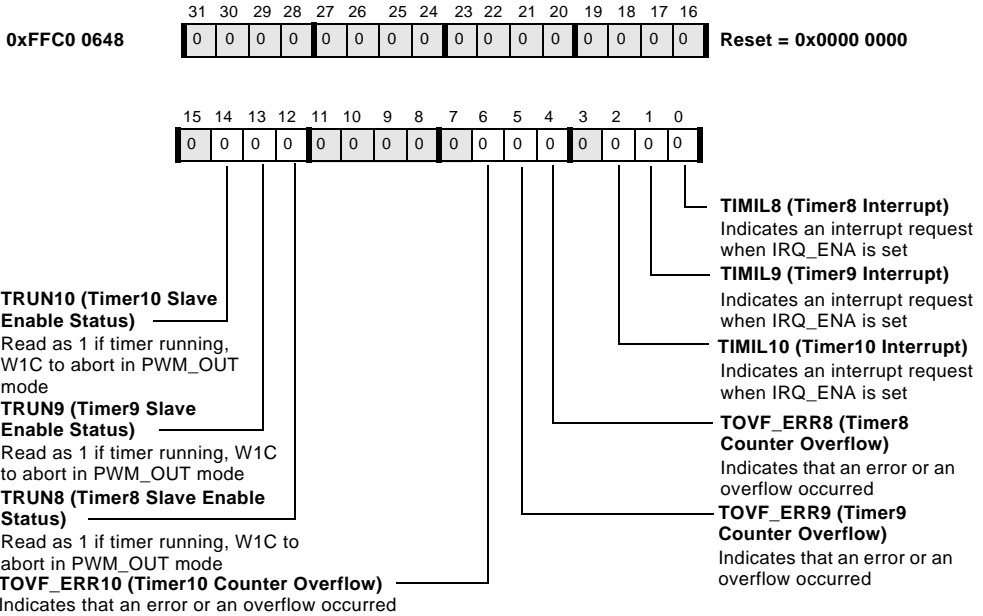


Figure 10-22. Timer Status 1 (TIMER_STATUS1) Register

Timer Configuration (TIMERx_CONFIG) Registers

Each timer's operating mode is specified by its `TIMERx_CONFIG` register (Figure 10-23 and Table 10-2), which 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_STATUSx` 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 10-1 on page 10-12. 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.

Timer Registers

Timer Configuration Registers (TIMERx_CONFIG)

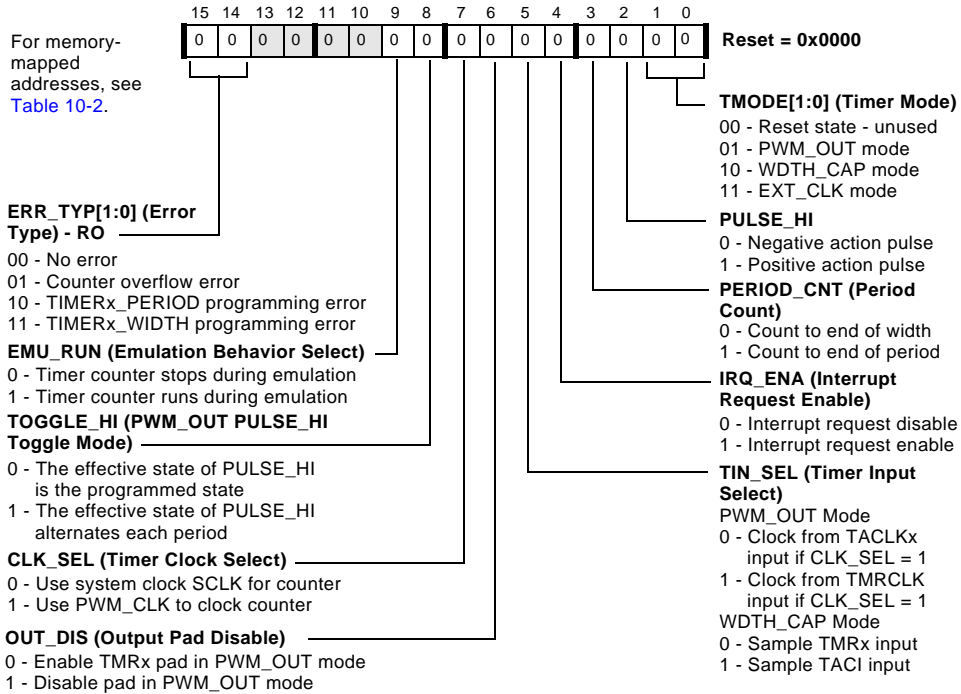


Figure 10-23. Timer Configuration (TIMERx_CONFIG) Registers

Table 10-2. Timer Configuration Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
TIMER0_CONFIG	0xFFC0 1600
TIMER1_CONFIG	0xFFC0 1610
TIMER2_CONFIG	0xFFC0 1620
TIMER3_CONFIG	0xFFC0 1630

Table 10-2. Timer Configuration Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
TIMER4_CONFIG	0xFFC0 1640
TIMER5_CONFIG	0xFFC0 1650
TIMER6_CONFIG	0xFFC0 1660
TIMER7_CONFIG	0xFFC0 1670
TIMER8_CONFIG	0xFFC0 0600
TIMER9_CONFIG	0xFFC0 0610
TIMER10_CONFIG	0xFFC0 0620

Timer Counter (TIMERx_COUNTER) Registers

These read-only registers retain their state when disabled. When enabled, the `TIMERx_COUNTER` register is reinitialized by hardware based on configuration and mode. The `TIMERx_COUNTER` register, shown in [Figure 10-24](#) and [Table 10-3](#), may be read at any time (whether the timer is running or stopped), and it returns an atomic 32-bit value. Depending on the operation mode, the incrementing counter can be clocked by four different sources: `SCLK`, the `TMRx` pin, the alternative timer clock pin `TACLKx`, or the common `TMCLK` pin, which is most likely used as the `EPPIO` clock (`EPPIO_CLK`).

When 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, inter-

Timer Registers

rupts 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)

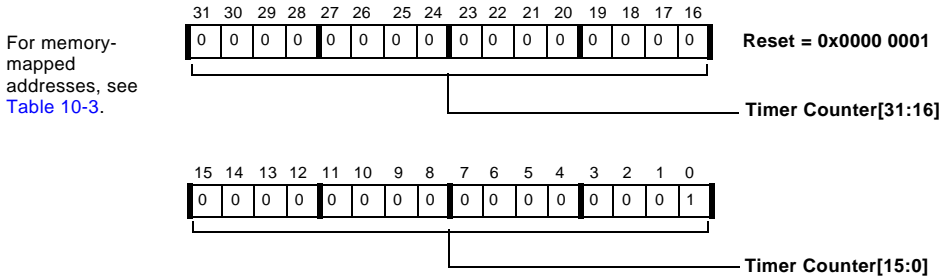


Figure 10-24. Timer Counter (TIMERx_COUNTER) Registers


Table 10-3. Timer Counter Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
TIMER0_COUNTER	0xFFC0 1604
TIMER1_COUNTER	0xFFC0 1614
TIMER2_COUNTER	0xFFC0 1624
TIMER3_COUNTER	0xFFC0 1634
TIMER4_COUNTER	0xFFC0 1644
TIMER5_COUNTER	0xFFC0 1654
TIMER6_COUNTER	0xFFC0 1664
TIMER7_COUNTER	0xFFC0 1674
TIMER8_COUNTER	0xFFC0 0604
TIMER9_COUNTER	0xFFC0 0614
TIMER10_COUNTER	0xFFC0 0624

TIMERx_PERIOD and TIMERx_WIDTH Registers

Usage of the `TIMERx_PERIOD` register, shown in [Figure 10-25](#) and [Table 10-3](#), and the `TIMERx_WIDTH` register, shown in [Figure 10-26](#) and [Table 10-4](#), 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 these 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` is writable and can be updated “on-the-fly.” `TIMERx_WIDTH` is not used.

 When a timer is enabled and running, and the software writes new values to `TIMERx_PERIOD` and `TIMERx_WIDTH`, the writes are buffered and do not update the registers until the end of the current period (when the value in `TIMERx_COUNTER` equals the value in `TIMERx_PERIOD`).

If new values are not written to `TIMERx_PERIOD` or `TIMERx_WIDTH`, the value from the previous period is reused. Writes to the 32-bit `TIMERx_PERIOD` and `TIMERx_WIDTH` registers 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` 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 `TIMERx_PERIOD` and `TIMERx_WIDTH` are updated from their respective buffers at the end of the current period. See [Figure 10-2 on page 10-5](#).

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 are ready for use in the first timer period. For example, to change the values for the `TIMERx_PERIOD` 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.

Timer Registers

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 `TIMERx_PERIOD` and `TIMERx_WIDTH`. The next period may use one old value and one new values.

In order to prevent “pulse width \geq period” errors, write `TIMERx_WIDTH` before `TIMERx_PERIOD` when decreasing the values, and write `TIMERx_PERIOD` before `TIMERx_WIDTH` when increasing the value.

Timer Period Registers (`TIMERx_PERIOD`)

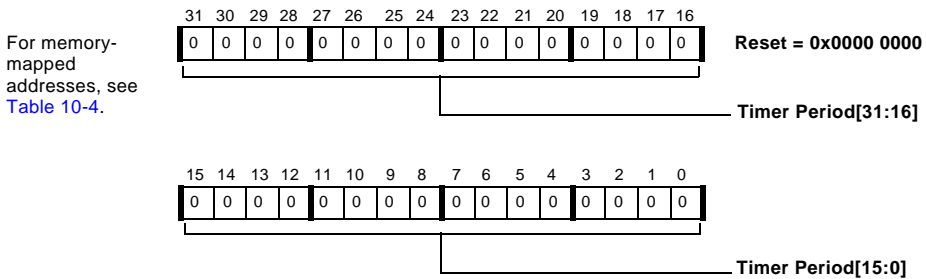


Figure 10-25. Timer Period (`TIMERx_PERIOD`) Registers

Table 10-4. Timer Period Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
<code>TIMER0_PERIOD</code>	0xFFC0 1608
<code>TIMER1_PERIOD</code>	0xFFC0 1618
<code>TIMER2_PERIOD</code>	0xFFC0 1628
<code>TIMER3_PERIOD</code>	0xFFC0 1638
<code>TIMER4_PERIOD</code>	0xFFC0 1648
<code>TIMER5_PERIOD</code>	0xFFC0 1658
<code>TIMER6_PERIOD</code>	0xFFC0 1668

Table 10-4. Timer Period Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
TIMER7_PERIOD	0xFFC0 1678
TIMER8_PERIOD	0xFFC0 0608
TIMER9_PERIOD	0xFFC0 0618
TIMER10_PERIOD	0xFFC0 0628

Timer Width Registers (TIMERx_WIDTH)

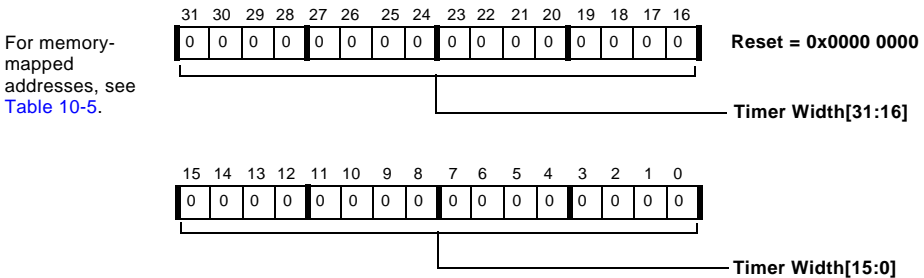


Figure 10-26. Timer Width (TIMERx_WIDTH) Registers

Table 10-5. Timer Width Register Memory-Mapped Addresses

Register Name	Memory-Mapped Address
TIMER0_WIDTH	0xFFC0 160C
TIMER1_WIDTH	0xFFC0 161C
TIMER2_WIDTH	0xFFC0 162C
TIMER3_WIDTH	0xFFC0 163C
TIMER4_WIDTH	0xFFC0 164C
TIMER5_WIDTH	0xFFC0 165C
TIMER6_WIDTH	0xFFC0 166C
TIMER7_WIDTH	0xFFC0 167C

Timer Registers

Table 10-5. Timer Width Register Memory-Mapped Addresses (Cont'd)

Register Name	Memory-Mapped Address
TIMER8_WIDTH	0xFFC0_060C
TIMER9_WIDTH	0xFFC0_061C
TIMER10_WIDTH	0xFFC0_062C

Summary

Table 10-6 summarizes control bit and register usage in each timer mode.

Table 10-6. Control Bit and Register Usage Chart

Bit / Register	PWM_OUT Mode	WDTH_CAP Mode	EXT_CLK Mode
TIMER_ENABLEx	1 - Enable timer 0 - No effect	1 - Enable timer 0 - No effect	1 - Enable timer 0 - No effect
TIMER_DISABLEx	1 - Disable timer at end of period 0 - No effect	1 - Disable timer 0 - No effect	1 - Disable timer 0 - No effect
TMODE	b#01	b#10	b#11
PULSE_HI	1 - Generate high width 0 - Generate low width	1 - Measure high width 0 - Measure low width	1 - Count rising edges 0 - Count falling edges
PERIOD_CNT	1 - Generate PWM 0 - Single width pulse	1 - Interrupt after measuring period 0 - Interrupt after measuring width	Unused
IRQ_ENA	1 - Enable interrupt 0 - Disable interrupt	1 - Enable interrupt 0 - Disable interrupt	1 - Enable interrupt 0 - Disable interrupt

Table 10-6. Control Bit and Register Usage Chart (Cont'd)

Bit / Register	PWM_OUT Mode	WDTH_CAP Mode	EXT_CLK Mode
TIN_SEL	Depends on CLK_SEL: If CLK_SEL = 1, 1 - Count TMRCLK clocks 0 - Count TACLKx clocks If CLK_SEL = 0, Unused	1 - Select TACI input 0 - Select TMRx input	Unused
OUT_DIS	1 - Disable TMRx pin 0 - Enable TMRx pin	Unused	Unused
CLK_SEL	1 - PWM_CLK clocks timer 0 - SCLK clocks timer	Unused	Unused
TOGGLE_HI	1 - One waveform period every two counter periods 0 - One waveform period every one counter period	Unused	Unused
ERR_TYP	Reports b#00, b#01, b#10, or b#11, as appropriate	Reports b#00 or b#01, as appropriate	Reports b#00, b#01, or b#10, as appropriate
EMU_RUN	0 - Halt during emulation 1 - Count during emulation	0 - Halt during emulation 1 - Count during emulation	0 - Halt during emulation 1 - Count during emulation
TMR Pin	Depends on OUT_DIS: 1 - Three-state 0 - Output	Depends on TIN_SEL: 1 - Unused 0 - Input	Input
Period	R/W: Period value	RO: Period value	R/W: Period value
Width	R/W: Width value	RO: Width value	Unused

Programming Examples

Table 10-6. Control Bit and Register Usage Chart (Cont'd)

Bit / Register	PWM_OUT Mode	WDTH_CAP Mode	EXT_CLK Mode
Counter	RO: Counts up on SCLK or PWM_CLK	RO: Counts up on SCLK	RO: Counts up on TMRx event
TRUNx	Read: Timer slave enable status Write: 1 - Stop timer if disabled 0 - No effect	Read: Timer slave enable status Write: 1 - No effect 0 - No effect	Read: Timer slave enable status Write: 1 - No effect 0 - No effect
TOVF_ERR	Set at startup or rollover if period = 0 or 1 Set at rollover if width >= Period Set if counter wraps	Set if counter wraps	Set if counter wraps or set at startup or rollover if period = 0
IRQ	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	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	Depends on IRQ_ENA: 1 - Set when counter equals period or TOVF_ERR set 0 - Not set

Programming Examples

[Listing 10-1](#) configures the `PORTA_FER` register in a way that all eight TMRx pins are connected to port A.

Listing 10-1. Port Setup

```
timer_port_setup:
    [--sp] = (r7:7, p5:5);
```

```

p5.h = hi(PORTA_FER);
p5.l = lo(PORTA_FER);
r7.l = PA1|PA5;
w[p5] = r7;
p5.l = lo(PORTA_MUX);
r7.l = PFTE;
w[p5] = r7;
(r7:7, p5:5) = [sp++];
rts;
timer_port_setup.end:

```

Listing 10-2 generates signals on the TMR4 (PA1) and TMR5 (PA5) outputs. By default, timer 5 generates a continuous PWM signal with a duty cycle of 50% (period = 0x40 SCLKs, width = 0x20 SCLKs) while the PWM signal generated by timer 4 has the same period but 25% duty cycle (width = 0x10 SCLKs).

If the preprocessor constant `SINGLE_PULSE` is defined, every TMRx pin outputs only a single high pulse of 0x20 (timer 4) and 0x10 SCLKs (timer 5) duration.

In any case, the timers are started synchronously and the rising edges are aligned, that is, the pulses are left-aligned.

Listing 10-2. Signal Generation

```

// #define SINGLE_PULSE
timer45_signal_generation:
    [--sp] = (r7:7, p5:5);
    p5.h = hi(TIMER_ENABLE0);
    p5.l = lo(TIMER_ENABLE0);
#ifdef SINGLE_PULSE
    r7.l = PULSE_HI | PWM_OUT;
#else
    r7.l = PERIOD_CNT | PULSE_HI | PWM_OUT;

```

Programming Examples

```
#endif
    w[p5 + TIMER5_CONFIG - TIMER_ENABLE0] = r7;
    w[p5 + TIMER4_CONFIG - TIMER_ENABLE0] = r7;
    r7 = 0x10 (z);
    [p5 + TIMER5_WIDTH - TIMER_ENABLE0] = r7;
    r7 = 0x20 (z);
    [p5 + TIMER4_WIDTH - TIMER_ENABLE0] = r7;
#endifdef SINGLE_PULSE
    r7 = 0x40 (z);
    [p5 + TIMER5_PERIOD - TIMER_ENABLE0] = r7;
    [p5 + TIMER4_PERIOD - TIMER_ENABLE0] = r7;
#endif
    r7.l = TIMEN5 | TIMEN4;
    w[p5] = r7;
    (r7:7, p5:5) = [sp++];
    rts;
timer45_signal_generation.end:
```

All subsequent examples use interrupts. [Listing 10-3](#) illustrates how interrupts are generated and how interrupt service routines can be registers. In this example, the timer 5 interrupt is assigned to the IVG7 interrupt channel of the CEC controller.

Listing 10-3. Interrupt Setup

```
timer5_interrupt_setup:
    [--sp] = (r7:7, p5:5);
    p5.h = hi(IMASK);
    p5.l = lo(IMASK);
/* register interrupt service routine */
    r7.h = hi(isr_timer5);
    r7.l = lo(isr_timer5);
    [p5 + EVT7 - IMASK] = r7;
/* unmask IVG7 in CEC */
    r7 = [p5];
```

```

    bitset(r7, bitpos(EVT_IVG7));
    [p5] = r7;
    p5.h = hi(SIC_IMASK2);
    p5.l = lo(SIC_IMASK2);
/* assign timer 5 IRQ = IRQ91 to IVG7 */
    r7.h = hi(P91_IVG(7));
    r7.l = lo(P91_IVG(7));
    [p5 + SIC_IAR11 - SIC_IMASK2] = r7;
/* enable timer 5 IRQ */
    r7 = [p5];
    bitset(r7, 27);
    [p5] = r7;
/* enable interrupt nesting */
    (r7:7, p5:5) = [sp++];
    [--sp] = reti;
    rts;
timer5_interrupt_setup.end:

```

The example shown in [Listing 10-4](#) does not drive the TMR_x pin. It generates periodic interrupt requests every 0x1000 SCLK cycles. If the preprocessor constant `SINGLE_PULSE` is defined, timer 5 requests an interrupt only once. Unlike in a real application, the purpose of the interrupt service routine shown in this example is clearing of the interrupt request and counting interrupt occurrences.

Listing 10-4. Periodic Interrupt Requests

```

// #define SINGLE_PULSE
timer5_interrupt_generation:
    [--sp] = (r7:7, p5:5);
    p5.h = hi(TIMER_ENABLE0);
    p5.l = lo(TIMER_ENABLE0);
#ifdef SINGLE_PULSE
    r7.l = EMU_RUN | IRQ_ENA | OUT_DIS | PWM_OUT;
#else

```

Programming Examples

```
    r7.l = EMU_RUN | IRQ_ENA | PERIOD_CNT | OUT_DIS | PWM_OUT;
#endif
    w[p5 + TIMER5_CONFIG - TIMER_ENABLE0] = r7;
    r7 = 0x1000 (z);
#ifdef SINGLE_PULSE
    [p5 + TIMER5_PERIOD - TIMER_ENABLE0] = r7;
    r7 = 0x1 (z);
#endif
    [p5 + TIMER5_WIDTH - TIMER_ENABLE0] = r7;
    r7.l = TIMEN5;
    w[p5] = r7;
    (r7:7, p5:5) = [sp++];
    r0 = 0 (z);
    rts;
timer5_interrupt_generation.end:
isr_timer5:
    [--sp] = astat;
    [--sp] = (r7:7, p5:5);
    p5.h = hi(TIMER_STATUS0);
    p5.l = lo(TIMER_STATUS0);
    r7.h = hi(TIMIL5);
    r7.l = lo(TIMIL5);
    [p5] = r7;
    r0+= 1;
    ssync;
    (r7:7, p5:5) = [sp++];
    astat = [sp++];
    rti;
isr_timer5.end:
```


Figure 10-27 explains how the signal waveform represented by the period P and the pulse width W translates to timer period and width values. Table 10-7 summarizes the register writes.

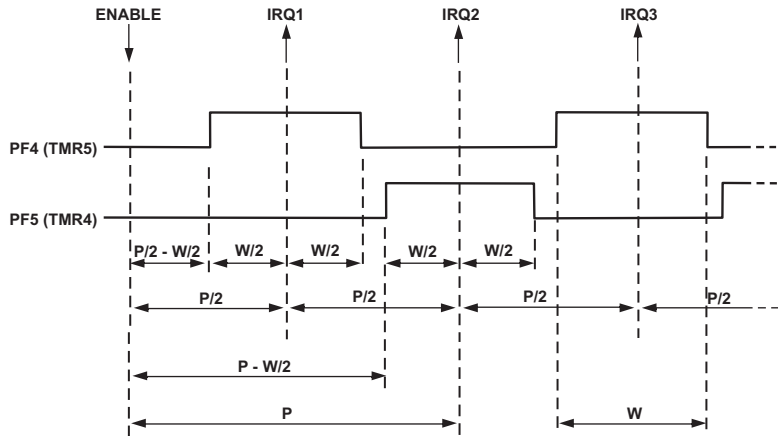


Figure 10-27. Non-Overlapping Clock Pulses

Table 10-7. Register Writes for Non-Overlapping Clock Pulses

Register	Before Enable	After Enable	At IRQ1	At IRQ2
TIMER5_PERIOD	$P/2$			
TIMER5_WIDTH	$P/2 - W/2$	$W/2$	$P/2 - W/2$	$W/2$
TIMER4_PERIOD	P	$P/2$		
TIMER4_WIDTH	$P - W/2$		$W/2$	$P/2 - W/2$

Programming Examples

Since hardware only updates the written period and width values at the end of periods, software can write new values immediately after the timers have been enabled. Note that both timers' period expires at exactly the same times with the exception of the first timer 5 interrupt (at IRQ1) which is not visible to timer 4.

[Listing 10-5](#) illustrates how two timers can generate two non-overlapping clock pulses as typically required for break-before-make scenarios. Both timers are running in PWM_OUT mode with PERIOD_CNT = 1 and PULSE_HI = 1.

[Listing 10-5](#) generates N pulses on both timer output pins. Disabling the timers does not corrupt the generated pulse pattern.

Listing 10-5. Non-Overlapping Clock Pulses

```
#define P 0x1000 /* signal period */
#define W 0x0600 /* signal pulse width */
#define N 4      /* number of pulses before disable */
timer45_toggle_hi:
    [--sp] = (r7:1, p5:5);
    p5.h = hi(TIMER_ENABLE0);
    p5.l = lo(TIMER_ENABLE0);
/* config timers */
    r7.l = IRQ_ENA | PERIOD_CNT | TOGGLE_HI | PULSE_HI | PWM_OUT;
    w[p5 + TIMER5_CONFIG - TIMER_ENABLE0] = r7;
    r7.l = PERIOD_CNT | TOGGLE_HI | PULSE_HI | PWM_OUT;
    w[p5 + TIMER4_CONFIG - TIMER_ENABLE0] = r7;
/* calculate timers widths and period */
    r0.l = lo(P);
    r0.h = hi(P);
    r1.l = lo(W);
    r1.h = hi(W);
    r2 = r1 >> 1; /* W/2 */
    r3 = r0 >> 1; /* P/2 */
```

```

    r4 = r3 - r2;    /* P/2 - W/2 */
    r5 = r0 - r2;    /* P - W/2 */
/* write values for initial period */
    [p5 + TIMER4_PERIOD - TIMER_ENABLE0] = r0;
    [p5 + TIMER4_WIDTH - TIMER_ENABLE0] = r5;
    [p5 + TIMER5_PERIOD - TIMER_ENABLE0] = r3;
    [p5 + TIMER5_WIDTH - TIMER_ENABLE0] = r4;
/* start timers */
    r7.l = TIMEN5 | TIMEN4 ;
    w[p5 + TIMER_ENABLE0 - TIMER_ENABLE0] = r7;
/* write values for second period */
    [p5 + TIMER4_PERIOD - TIMER_ENABLE0] = r3;
    [p5 + TIMER5_WIDTH - TIMER_ENABLE0] = r2;
/* r0 functions as signal period counter */
    r0.h = hi(N * 2 - 1);
    r0.l = lo(N * 2 - 1);
    (r7:1, p5:5) = [sp++];
    rts;
timer45_toggle_hi.end:
isr_timer5:
    [--sp] = astat;
    [--sp] = (r7:5, p5:5);
    p5.h = hi(TIMER_ENABLE0);
    p5.l = lo(TIMER_ENABLE0);
/* clear interrupt request */
    r7.h = hi(TIMIL5);
    r7.l = lo(TIMIL5);
    [p5 + TIMER_STATUS0 - TIMER_ENABLE0] = r7;
/* toggle width values (width = period - width) */
    r7 = [p5 + TIMER5_PERIOD - TIMER_ENABLE0];
    r6 = [p5 + TIMER5_WIDTH - TIMER_ENABLE0];
    r5 = r7 - r6;
    [p5 + TIMER5_WIDTH - TIMER_ENABLE0] = r5;
    r5 = [p5 + TIMER4_WIDTH - TIMER_ENABLE0];

```

Programming Examples

```
    r7 = r7 - r5;
    CC = r7 < 0;
    if CC r7 = r6;
    [p5 + TIMER4_WIDTH - TIMER_ENABLE0] = r7;
/* disable after a certain number of periods */
    r0+= -1;
    CC = r0 == 0;
    r5.l = 0;
    r7.l = TIMDIS5 | TIMDIS4;
    if !CC r7 = r5;
    w[p5 + TIMER_DISABLE0 - TIMER_ENABLE0] = r7;
    (r7:5, p5:5) = [sp++];
    astat = [sp++];
    rti;
isr_timer5.end;
```

Listing 10-6 configures timer 5 in `WDTH_CAP` mode. If looped back externally, this code can be used to receive `N` PWM patterns generated by one of the other timers. Ensure that the PWM generator uses the same `PERIOD_CNT` and `PULSE_HI` settings.

Listing 10-6. Timer Configured in `WDTH_CAP` Mode

```
.section L1_data_a;
.align 4;
#define N 1024
.var buffReceive[N*2];
.section L1_code;
timer5_capture:
    [--sp] = (r7:7, p5:5);
/* setup DAG2 */
    r7.h = hi(buffReceive);
    r7.l = lo(buffReceive);
    i2 = r7;
    b2 = r7;
```

```

    l2 = length(buffReceive)*4;
/* config timer for high pulses capture */
    p5.h = hi(TIMER_ENABLE0);
    p5.l = lo(TIMER_ENABLE0);
    r7.l = EMU_RUN|IRQ_ENA|PERIOD_CNT|PULSE_HI|WIDTH_CAP;
    w[p5 + TIMER5_CONFIG - TIMER_ENABLE0] = r7;
    r7.l = TIMEN5;
    w[p5 + TIMER_ENABLE0 - TIMER_ENABLE0] = r7;
    (r7:7, p5:5) = [sp++];
    rts;
timer5_capture.end:
isr_timer5:
    [--sp] = astat;
    [--sp] = (r7:7, p5:5);
/* clear interrupt request first */
    p5.h = hi(TIMER_STATUS0);
    p5.l = lo(TIMER_STATUS0);
    r7.h = hi(TIMIL5);
    r7.l = lo(TIMIL5);
    [p5] = r7;
    r7 = [p5 + TIMERO_PERIOD - TIMER_STATUS0];
    [i2++] = r7;
    r7 = [p5 + TIMERO_WIDTH - TIMER_STATUS0];
    [i2++] = r7;
    ssync;
    (r7:7, p5:5) = [sp++];
    astat = [sp++];
    rti;
isr_timer5.end:

```

Programming Examples

11 CORE TIMER

This chapter describes the core timer and includes the following sections:

- [“Overview and Features” on page 11-1](#)
- [“Timer Overview” on page 11-2](#)
- [“Description of Operation” on page 11-3](#)
- [“Core Timer Registers” on page 11-4](#)
- [“Programming Examples” on page 11-8](#)

Overview and Features

The core timer is a programmable, 32-bit interval timer that can generate periodic interrupts. Unlike other peripherals, the core timer resides inside the Blackfin processor core and runs at the core clock (CCLK) rate.

Core timer features include:

- 32-bit timer with 8-bit prescaler
- Operation at core clock (CCLK) rate
- Dedicated high-priority interrupt channel
- Single-shot or continuous operation

Timer Overview

External Interfaces

The core timer does not directly interact with any pins of the chip.

Internal Interfaces

The core timer is accessed through the 32-bit register access bus (RAB). The module is clocked by the core clock `CCLK`. The timer has its dedicated interrupt request signal which is of higher priority than all other peripherals' requests.

Figure 11-1 provides a block diagram of the core timer.

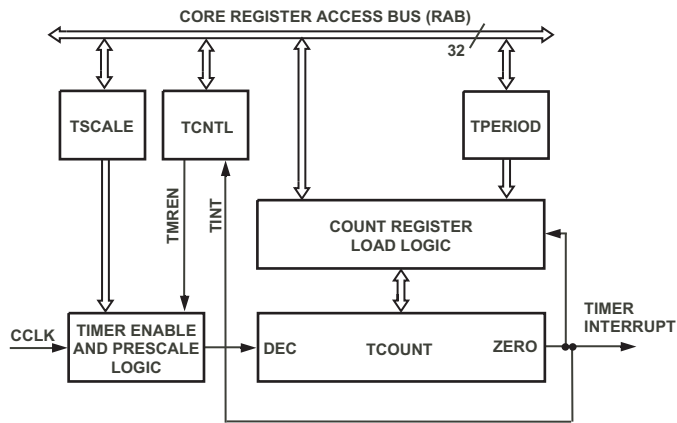


Figure 11-1. Core Timer Block Diagram

Description of Operation

It is up to software to initialize the core timer's counter (TCOUNT) *before* the timer is enabled. The TCOUNT register can be written directly. However, writes to the TPERIOD register are also passed through to the counter, TCOUNT.

When the timer is enabled by setting the TMREN bit in the core timer control register (TCNTL), the TCOUNT register is decremented once every time the prescaler (TSCALE) expires, that is, every $TSCALE + 1$ number of CCLK 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. If the TAUTORLD bit is not set, the timer stops operation.

The core timer can be put into low power mode by clearing the TMPWR bit in the TCNTL register. Before using the timer, set the TMPWR bit. This restores clocks to the timer unit. When TMPWR bit is set, the core timer may then be enabled by setting the TMREN bit in the TCNTL register.



Hardware behavior is undefined if TMREN bit is set when TMPWR = 0.

Interrupt Processing

The core timer has its dedicated interrupt request signal which is of higher priority than all other peripherals' requests. The requests goes directly to the Core Event Controller (CEC) and does not pass the System Interrupt Controller (SIC). Therefore, the interrupt processing is also completely in the CCLK domain.



Unlike requests from other Blackfin processor peripherals, the core interrupt request is edge sensitive and cleared by hardware automatically as soon as the interrupt is serviced.

Core Timer Registers

The `TINT` bit in the `TCNTL` register indicates that an interrupt is 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 time module does not provide any further interrupt enable bit. When the timer is enabled, interrupts can be masked in the CEC controller.

Core Timer Registers

The core timer includes the following four core memory-mapped registers (MMRs):

- [“Core Timer Control \(TCNTL\) Register” on page 11-5](#)
- [“Core Timer Count \(TCOUNT\) Register” on page 11-6](#)
- [“Core Timer Period \(TPERIOD\) Register” on page 11-7](#)
- [“Core Timer Scale \(TSCALE\) Register” on page 11-7](#)

Similar to all core MMRs, these registers are always accessed by 32-bit read and write operations.

Core Timer Control (TCNTL) Register

The core timer control (TCNTL) register, shown in [Figure 11-2](#), functions as a control and status register.

Core Timer Control Register (TCNTL)

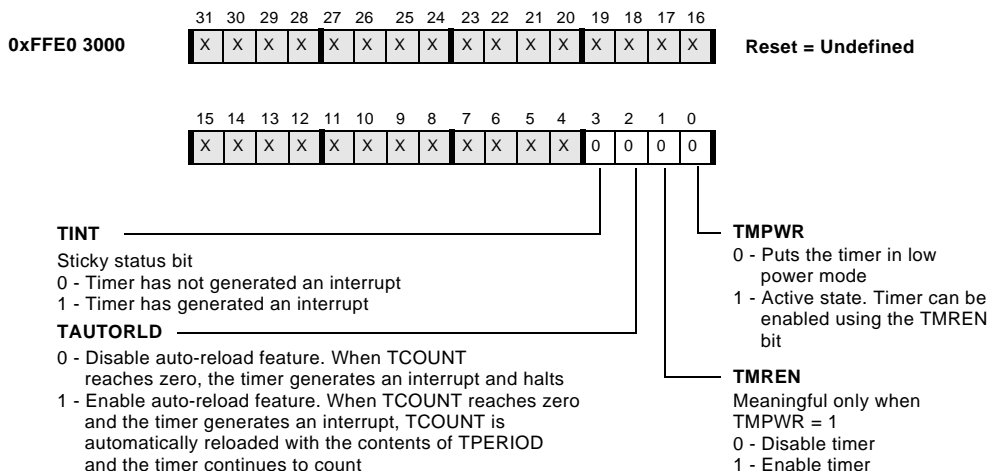


Figure 11-2. Core Timer Control (TCNTL) Register

Core Timer Count (TCOUNT) Register

The core timer count register (TCOUNT) shown in [Figure 11-3](#) decrements once every $TSCALE + 1$ clock cycles. When the value of TCOUNT reaches 0, an interrupt is generated and the TINT bit of the TCNTL register is set.

Values written to the TPERIOD register are automatically copied to the TCOUNT register as well. Nevertheless, the TCOUNT register can be written directly. In auto-reload mode the value written to TCOUNT may differ from the TPERIOD value to let the initial period be shorter or longer than the following ones. To do this, write to TPERIOD first and overwrite TCOUNT register afterward.

Writes to TCOUNT are ignored once the timer is running.

Core Timer Count Register (TCOUNT)

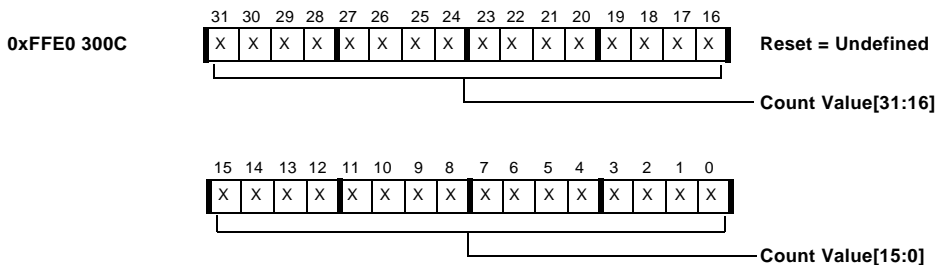


Figure 11-3. Core Timer Count (TCOUNT) Register

Core Timer Period (TPERIOD) Register

When auto-reload is enabled, the `TCOUNT` register is reloaded with the value of the core timer period register (`TPERIOD`, shown in [Figure 11-4](#)), whenever `TCOUNT` register reaches 0. Writes to `TPERIOD` register are ignored when the timer is running.

Core Timer Period Register (TPERIOD)

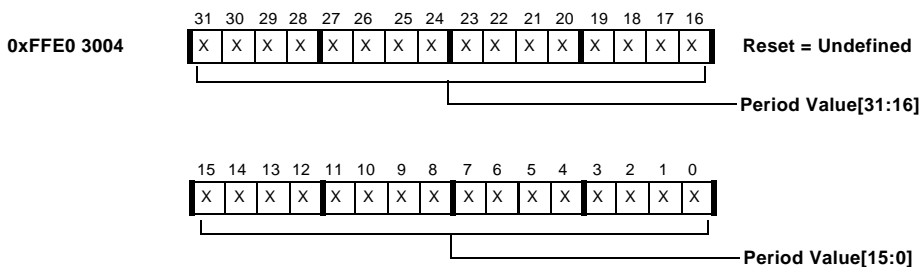


Figure 11-4. Core Timer Period (TPERIOD) Register

Core Timer Scale (TSCALE) Register

The core timer scale register (`TSCALE`, shown in [Figure 11-5](#),) stores the scaling value that is one less than the number of cycles between decrements of `TCOUNT` register. For example, if the value in the `TSCALE` register is 0, the counter register decrements once every `CCLK` clock cycle. If the value of `TSCALE` register is 1, the counter decrements once every two cycles.

Programming Examples

Core Timer Scale Register (TSCALE)

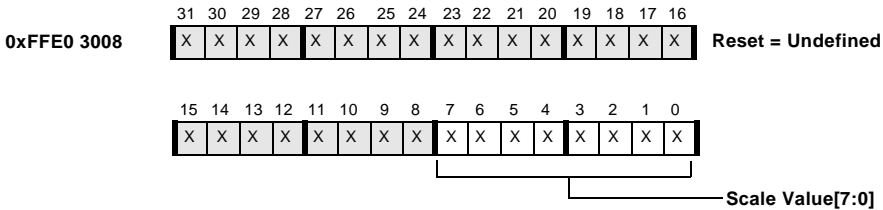


Figure 11-5. Core Timer Scale (TSCALE) Register

Programming Examples

[Listing 11-1](#) configures the core timer in auto reload mode. Assuming a CCLK of 500 MHz, the resulting period is 1 s. The initial period is twice as long as the others.

Listing 11-1. Core Timer Configuration

```
#include <blackfin.h>
.section L1_code;
.global _main;
_main:
/* Register service routine at EVT6 and unmask interrupt */
    p1.l = lo(IMASK);
    p1.h = hi(IMASK);
    r0.l = lo(isr_core_timer);
    r0.h = hi(isr_core_timer);
    [p1 + EVT6 - IMASK] = r0;
    r0 = [p1];
    bitset(r0, bitpos(EVT_IVTMR));
    [p1] = r0;
```

```

/* Prescaler = 50, Period = 10,000,000, First Period = 20,000,000
*/
    p1.l = lo(TCNTL);
    p1.h = hi(TCNTL);
    r0 = 50 (z);
    [p1 + TSCALE - TCNTL] = r0;
    r0.l = lo(10000000);
    r0.h = hi(10000000);
    [p1 + TPERIOD - TCNTL] = r0;
    r0 <<= 1;
    [p1 + TCOUNT - TCNTL] = r0;
/* R6 counts interrupts */
    r6 = 0 (z);
/* start in auto-reload mode */
    r0 = TAUTORLD | TMPWR | TMREN (z);
    [p1] = r0;
_main.forever:
    jump _main.forever;
_main.end:
/* interrupt service routine simple increments R6 */
_isr_core_timer:
    [--sp] = astat;
    r6+= 1;
    astat = [sp++];
    rti;
_isr_core_timer.end:

```

Programming Examples

12 WATCHDOG TIMER

This chapter describes the watchdog timer and includes the following sections:

- [“Overview and Features” on page 12-1](#)
- [“Interface Overview” on page 12-3](#)
- [“Description of Operation” on page 12-4](#)
- [“Watchdog Timer Registers” on page 12-6](#)
- [“Programming Examples” on page 12-10](#)

Overview and Features

The Blackfin 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 watchdog expires before being updated by software.

Watchdog timer key features include:


- 32-bit watchdog timer
- 8-bit disable bit pattern
- System reset on expire option
- NMI on expire option
- General-purpose interrupt option

Overview and Features

Typically, the watchdog timer is used to supervise stability of the system software. When used in this way, software reloads the watchdog timer in a regular manner so that the downward counting timer never expires (never becomes 0). An expiring timer then indicates that system software might be out of control. At this point a special error handler may recover the system. For safety, however, it is often better to reset and reboot the system directly by hardware control.

Especially in slave boot configurations, a processor reset cannot automatically force the part to reboot. In this case, the processor may reset without booting again and may negotiate with the host device by the time program execution starts. Alternatively, a watchdog event can cause an NMI event. The NMI service routine may request the host device to reset and/or reboot the Blackfin processor.

Often, the watchdog timer is also programmed to let the processor wake up from sleep mode after a programmable period of time.

 For easier debugging, the watchdog timer does not decrement (even if enabled) when the processor is in emulation mode.

Interface Overview

Figure 12-1 provides a block diagram of the watchdog timer.

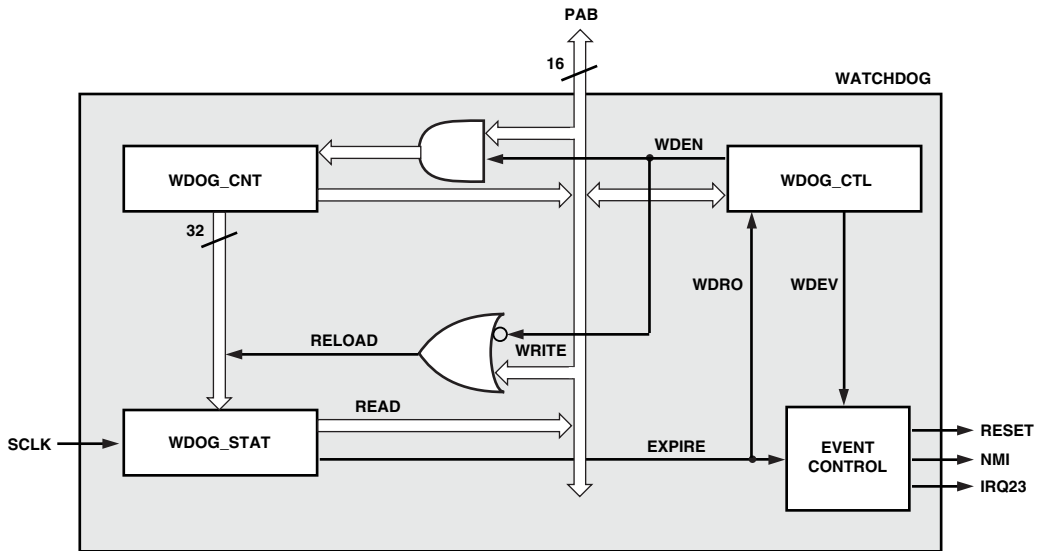


Figure 12-1. Watchdog Timer Block Diagram

External Interface

The watchdog timer does not directly interact with any pins of the chip.

Internal Interface

The watchdog timer is clocked by the system clock **SCLK**. Its registers are accessed through the 16-bit peripheral access bus **PAB**. The 32-bit registers **WDOG_CNT** and **WDOG_STAT** must always be accessed by 32-bit read/write operations. Hardware ensures that those accesses are atomic.

Description of Operation

When the counter expires, one of three event requests can be generated. Either a reset or an NMI request is issued to the core event controller (CEC) or a general-purpose interrupt request is passed to the system interrupt controller (SIC).

Description of Operation

If enabled, the 32-bit watchdog timer counts downward every `SCLK` cycle. If it becomes 0, one of three event requests can be issued to either the CEC or the SIC. Depending on how the `WDEV` bit field in the `WDOG_CTL` register is programmed, the event that is generated may be a reset, a non-maskable interrupt, or a general-purpose interrupt.

The counter value can be read through the 32-bit `WDOG_STAT` register. The `WDOG_STAT` register cannot, however, be written directly. Rather, software writes the watchdog period value into the 32-bit `WDOG_CNT` register *before* the watchdog is enabled. Once the watchdog is started, the period value cannot be altered.

To start the watchdog timer:

1. Set the count value for the watchdog timer by writing the count value into the watchdog count register (`WDOG_CNT`). Since the watchdog timer is not yet enabled, the write to the `WDOG_CNT` registers automatically preloads the `WDOG_STAT` register as well.
2. In the watchdog control register (`WDOG_CTL`), select the event to generate upon timeout.
3. Enable the watchdog timer in `WDOG_CTL`. The watchdog timer then begins counting down, decrementing the value in the `WDOG_STAT` register.

If software does not serve the watchdog in time, `WDOG_STAT` register continues decrementing until it reaches 0. Then, the programmed event is generated. The counter stops decrementing and remains at zero. Additionally, the `WDRO` latch bit in the `WDOG_CTL` register is set and can be interrogated by software in case event generation is not enabled.

When the watchdog is programmed to generate a reset, it resets the processor core and peripherals. If the `NOBOOT` bit in the `SYSCR` register was set by the time the watchdog reset the part, the chip is not rebooted. This is recommended behavior in slave boot configurations. The reset handler may evaluate the `RESET_WDOG` bit in the software reset register `SWRST` to detect a reset caused by the watchdog. For details, see [Chapter 17, “System Reset and Booting”](#).

To prevent the watchdog from expiring, software serves the watchdog by performing dummy writes to the `WDOG_STAT` register address in time. The values written are ignored, but the write commands cause the `WDOG_STAT` register to reload from the `WDOG_CNT` register.

If the watchdog is enabled with a zero value loaded to the counter and the `WDRO` bit was cleared, the `WDRO` bit of the watchdog control register is set immediately and the counter remains at zero without further decrements. If, however, the `WDRO` bit was set by the time the watchdog is enabled, the counter decrements to `0xFFFF FFFF` and continues operation.

Software can disable the watchdog timer only by writing a `0xAD` value (`WDDIS`) to the `WDEN` field in the `WDOG_CTL` register.

Watchdog Timer Registers

The watchdog timer is controlled by three registers.

- “[Watchdog Count \(WDOG_CNT\) Register](#)” on page 12-6
- “[Watchdog Status \(WDOG_STAT\) Register](#)” on page 12-7
- “[Watchdog Control \(WDOG_CTL\) Register](#)” on page 12-8

Watchdog Count (WDOG_CNT) Register

The watchdog count register (`WDOG_CNT`, shown in [Figure 12-2](#)) holds the 32-bit unsigned count value. The `WDOG_CNT` register must always be accessed with 32-bit read/writes.

The watchdog count register holds the programmable count value. A valid write to the watchdog count register also preloads the watchdog counter. For added safety, the watchdog count 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 Count Register (WDOG_CNT)

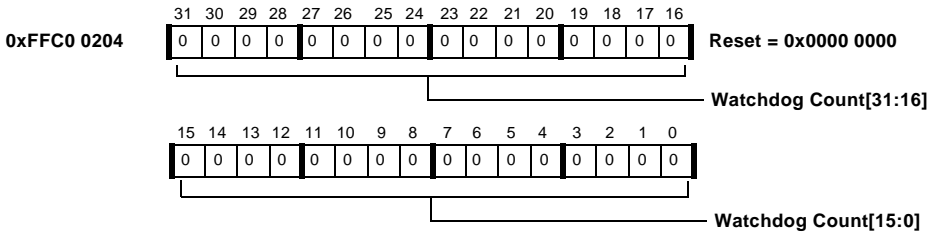


Figure 12-2. Watchdog Count (WDOG_CNT) Register

Watchdog Status (WDOG_STAT) Register

The 32-bit watchdog status register (WDOG_STAT, shown in [Figure 12-3](#)) contains the current count value of the watchdog timer. Reads to WDOG_STAT register return the current count value. Values cannot be stored directly in WDOG_STAT register, but are instead copied from WDOG_CNT register. This can happen in two ways:

- While the watchdog timer is disabled, writing the WDOG_CNT register preloads the WDOG_STAT register.
- While the watchdog timer is enabled, but not yet rolled over, writes to the WDOG_STAT register load it with the value in WDOG_CNT register.



Enabling the watchdog timer does not automatically reload WDOG_STAT register from WDOG_CNT register.

Watchdog Timer Registers

The `WDOG_STAT` register is a 32-bit unsigned system memory-mapped register that must be accessed with 32-bit reads and writes.

Watchdog Status Register (`WDOG_STAT`)

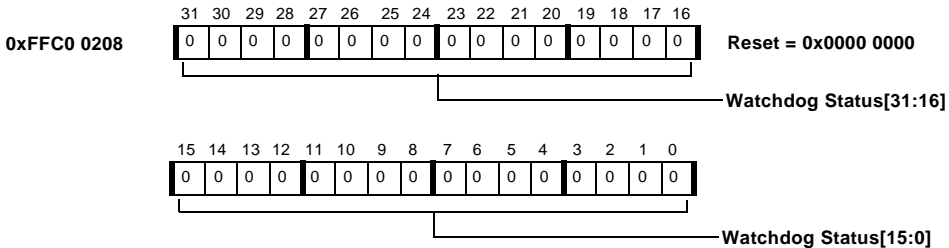


Figure 12-3. Watchdog Status (`WDOG_STAT`) Register

Watchdog Control (`WDOG_CTL`) Register

The watchdog control register (`WDOG_CTL`, shown in [Figure 12-4](#)) is a 16-bit system memory-mapped register used to control the watchdog timer.

The watchdog event (`WDEV[1:0]`) bit 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_IMASK`) 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 watchdog enable (`WDEN[7:0]`) bit field is used to enable and disable the watchdog timer. Writing any value other than the disable value (`0xAD`) 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 watchdog has expired by interrogating the watchdog rolled over (WDRO) status bit of the watchdog control register. This is a sticky bit that is set whenever the watchdog timer count reaches 0. It can be cleared only by writing a 1 to the bit when the watchdog has been disabled first.

Watchdog Control Register (WDOG_CTL)

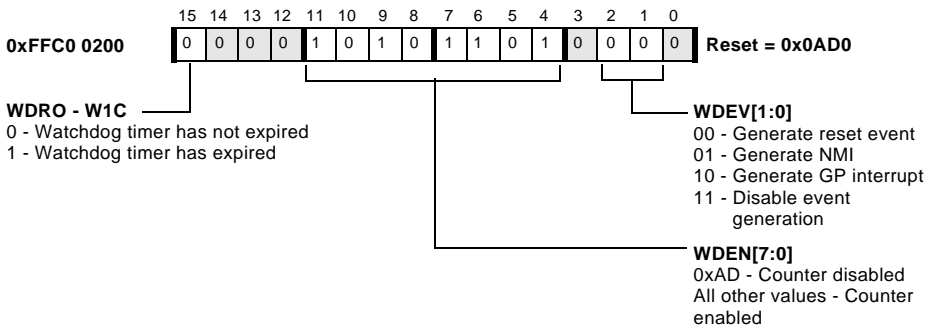


Figure 12-4. Watchdog Control (WDOG_CTL) Register

Programming Examples

[Listing 12-1](#) shows how to configure the watchdog timer so that it resets the chip when it expires. At startup, the code evaluates whether the recent reset event is caused by the watchdog. Additionally, the example sets the NOBOOT bit to prevent the memory from being rebooted.

Listing 12-1. Watchdog Timer Configuration

```
#include <blackfin.h>
#define WDOGPERIOD 0x00200000

.section L1_code;
.global _reset;
_reset:
    ...
/* optionally, test whether reset was caused by watchdog */
    p0.h=hi(SWRST);
    p0.l=lo(SWRST);
    r6 = w[p0] (z);
    CC = bittst(r6, bitpos(RESET_WDOG));
    if !CC jump _reset.no_watchdog_reset;

/* optionally, warn at system level or host device here */

_reset.no_watchdog_reset:
/* optionally, set NOBOOT bit to avoid reboot in case */
    p0.h=hi(SYSCR);
    p0.l=lo(SYSCR);
    r0 = w[p0](z);
    bitset(r0,bitpos(NOBOOT));
    w[p0] = r0;
```

```
/* start watchdog timer, reset if expires */
    p0.h = hi(WDOG_CNT);
    p0.l = lo(WDOG_CNT);
    r0.h = hi(WDOGPERIOD);
    r0.l = lo(WDOGPERIOD);
    [p0] = r0;
    p0.l = lo(WDOG_CTL);
    r0.l = WDEN | WDEV_RESET;
    w[p0] = r0;
    ...
    jump _main;
_reset.end:
```

The subroutine shown in [Listing 12-2](#) can be called by software to service the watchdog. Note that the value written to the WDOG_STAT register does not matter.

Listing 12-2. Service Watchdog

```
service_watchdog:
    [--sp] = p5;
    p5.h = hi(WDOG_STAT);
    p5.l = lo(WDOG_STAT);
    [p5] = r0;
    p5 = [sp++];
    rts;
service_watchdog.end:
```

Programming Examples

[Listing 12-3](#) is an interrupt service routine that restarts the watchdog. Note that the watchdog must be disabled first.

Listing 12-3. Watchdog Restarted by Interrupt Service Routine

```
isr_watchdog:
    [--sp] = astat;
    [--sp] = (p5:5, r7:7);
    p5.h = hi(WDOG_CTL);
    p5.l = lo(WDOG_CTL);
    r7.l = WDDIS;
    w[p5] = r7;
    bitset(r7, bitpos(WDR0));
    w[p5] = r7;
    r7 = [p5 + WDOG_CNT - WDOG_CTL];
    [p5 + WDOG_CNT - WDOG_CTL] = r7;
    r7.l = WDEN | WDEV_GPI;
    w[p5] = r7;
    (p5:5, r7:7) = [sp++];
    astat = [sp++];
    rti;
isr_watchdog.end:
```

13 ROTARY COUNTER

This chapter describes the rotary (up/down) counter, which provides support for manually-controlled rotary controllers, such as the volume wheel on a radio device. This unit also supports industrial encoders.

This chapter describes the rotary counter and includes the following sections:

- [“Overview” on page 13-1](#)
- [“Interface Overview” on page 13-3](#)
- [“Description of Operation” on page 13-4](#)
- [“Functional Description” on page 13-8](#)
- [“Programming Mode” on page 13-24](#)
- [“Rotary Counter Registers” on page 13-24](#)
- [“Programming Examples” on page 13-34](#)

Overview

The primary purpose of the rotary counter is to convert pulses from incremental position encoders into data that is representative of the actual position. This is done by integrating (counting) pulses on one or two inputs.

Overview

Since integration provides relative position, some devices also feature a zero position input (zero marker) that can be used to establish a reference point or alternative to verify that the acquired position does not drift over time.

In addition, the incremental position information can be used to determine speed, if the time intervals are measured.

The rotary counter interface provides various and flexible ways to establish position information. When used in conjunction with the general-purpose (GP) timer block, the rotary counter interface allows for the acquisition of coherent position/timestamp information that enables speed calculation.

Features

The rotary counter includes the following features:

- 32-bit rotary counter
- Quadrature encoder mode (gray code)
- Binary encoder mode
- Alternative frequency-direction mode
- Timed direction and up/down counting modes
- Zero marker/pushbutton support
- Capture event timing in association with GP timer
- Boundary comparison and boundary setting features
- Input pin noise filtering (debouncing)
- Flexible error detection/signaling

Interface Overview

A block diagram of the rotary counter interface is shown in [Figure 13-1](#). There are two input pins, the count up and direction (CUD) pin and the count down and gate (CDG) pin, that accept various forms of incremental inputs and are processed by the 32-bit counter. The third input, count zero marker (CZM), is the zero marker input. The module interfaces to the processor by way of the peripheral access bus (PAB) and can optionally generate an interrupt request through the IRQ line. There is also an output that can be used by the timer module to generate timestamps on certain events.

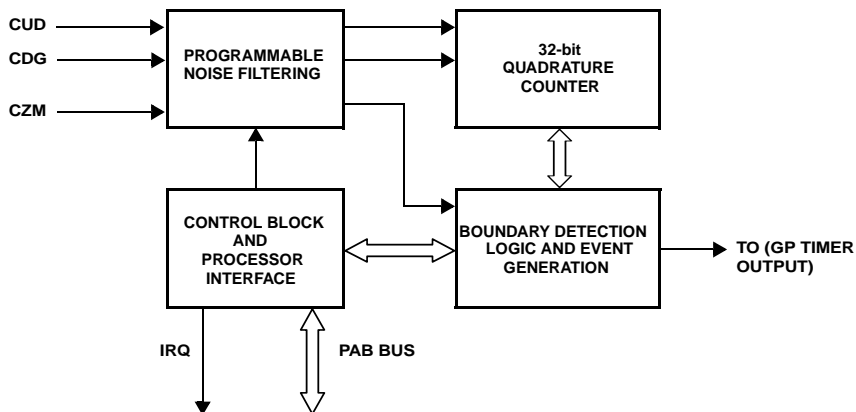


Figure 13-1. Block Diagram of the Rotary Counter Interface

The timer output signal is connected internally to the alternate capture input (TAC16) of the general-purpose timer 6. The interrupt signal goes to the IRQ68 input of the SIC2 controller.

Description of Operation

The rotary encoder block has five modes of operation that are described in this section.

With the exception of the timed direction mode, the rotary timer block can operate in conjunction with the GP timer block in order to capture additional timing information (timestamps) associated with events detected by this block.

The third input (CZM) may be used as a zero marker or to sense the pressing of a pushbutton. Refer to [“Zero Marker \(Pushbutton\) Operation” on page 13-11](#) for more details.

The three input pins may be filtered (debounced) prior to being evaluated by the rotary encoder. Refer to [“Input Noise Filtering \(Debouncing\)” on page 13-8](#) for more details.

The encoder block also features a flexible boundary comparison. In all of the operating modes, the counter can be compared to an upper and lower limit. A variety of actions can be taken when these limits are reached. Refer to [“Boundary Comparison Modes” on page 13-14](#) for more details.

Quadrature Encoder Mode

In this mode, the CUD:CDG inputs expect a quadrature-encoded signal that is interpreted as a 2-bit gray code. The order of transitions of the CUD and CDG inputs determines whether the counter increments or decrements. The register CNT_COUNTER contains the number of transitions that have occurred. Refer to [Figure 13-2](#) for more details.

[Figure 13-2](#) shows an example of a series of count up events which is causing CNT_COUNTER to increment.

Optionally, an interrupt is generated if both inputs change within one SCLK cycle. Such transitions are not allowed by gray coding. Therefore, the register CNT_COUNTER remains unchanged and an error condition is signaled.

CNT_COUNTER register value	-4	-3	-2	-1	0	+1	+2	+3	+4
CDG:CUD inputs	00	01	11	10	00	01	11	10	00

Figure 13-2. Quadrature Events and Counting Mechanism

It is possible to reverse the count direction of the gray-coded signal. This can be achieved by enabling the polarity inverter of either the CUD pin or the CDG pin, inverting both pins does not alter the behavior. This feature can be enabled with the CDGINV and CUDINV bits in the CNT_CONFIG register.

As an example, if the CDG:CUD inputs are 00 respectively and the next transition is to 01, this would normally increment the counter as is seen in [Figure 13-2](#). If the CUD polarity is inverted this generates a received input of 01 followed by 00. This will result in a decrement of the counter, altering the behavior of the connected hardware.

Binary Encoder Mode

This mode is almost identical to the previous mode, with the exception that the CUD:CDG inputs expect a binary-encoded signal. The order of transitions of the CUD and CDG inputs determines whether the counter increments or decrements. The register CNT_COUNTER contains the number of transitions that have occurred. Refer to [Figure 13-3](#).

In [Figure 13-3](#), a series of binary up count events are causing the CNT_COUNTER register to increment.

Description of Operation

Optionally, an interrupt is generated if the detected code steps by more than 1 (in binary arithmetic) within one `SCLK` cycle. Such transitions are considered erroneous. Therefore, the register `CNT_COUNTER` remains unchanged and an error condition is signaled.

<code>CNT_COUNTER</code> register value	-4	-3	-2	-1	0	+1	+2	+3	+4
<code>CDG:CUD</code> inputs	00	01	10	11	00	01	10	11	00

Figure 13-3. Binary Events and Counting Mechanism

Reversing the `CUD` and `CDG` pin polarity has a different effect for the binary encoder mode than from the quadrature encoder mode. Inverting the polarity of the `CUD` pin only or inverting both the `CUD` and `CDG` pins result in reversing the count direction.

Rotary Counter Mode

In this general-purpose mode, the counter is incremented or decremented at every active edge of the input pins.

If an active edge is detected at the `CUD` input, the counter increments. The active edge can be selected by way of the `CUDINV` bit in the `CNT_CONFIG` register. If this bit is cleared, a rising edge increments the counter. If the configuration bit is set, a falling edge increments the counter.

If an active edge is detected at the `CDG` input, the counter decrements. The active edge can be selected by way of the `CDGINV` bit in the `CNT_CONFIG` register. If this bit is cleared, a rising edge decrements the counter. If the configuration bit is set, a falling edge decrements the counter.

If simultaneous edges occur on pin `CDG` and pin `CUD`, the counter remains unchanged and both up-count and down-count events are signaled in the `CNT_STATUS` register.

Direction Counter Mode

In this mode the CUD input pin is used to determine direction and the CDG input is used as a gate.

In this general-purpose mode, the counter is incremented or decremented at every active edge of the CDG input pin.

The state of the CUD input determines whether the counter increments or decrements. The polarity can be selected by way of the CUDINV bit in the CNT_CONFIG register. If this bit is cleared, a high CUD input selects the direction to increment, a low input selects the direction to decrement. If the configuration bit is set, the polarity is inverted.

If an active edge is detected at the CDG input, the counter value changes by one in the selected direction. The active edge can be selected by way of the CDGINV bit in the CNT_CONFIG register. If this bit is cleared, a rising edge decrements the counter. If the configuration bit is set, a falling edge decrements the counter.

Timed Direction Mode

In this general-purpose mode, the counter is incremented or decremented at each SCLK cycle.

The state of the CUD input determines whether the counter increments or decrements. The polarity can be selected by way of the CUDINV bit in the CNT_CONFIG register. If this bit is cleared, a high CUD input will increment the counter, a low input decrements it. If the configuration bit is set, the polarity is inverted.

The CDG pin can be used to gate the clock. The polarity can be selected by way of the CDGINV bit in the CNT_CONFIG register. If this bit is cleared, a high CDG input enables the counter, a low input will stop it. If the configuration bit is set, the polarity is inverted.

Functional Description

The following sections describe the rotary counter in more detail.

Input Noise Filtering (Debouncing)

The rotary inputs are asynchronous to the system clock so hardware synchronizes them internally before using. This synchronization causes a fixed delay of a few clocks before any actions result from the toggling of the inputs.

Because of the synchronization, the minimum pulse width of the input signals must be the period of the system clock. For signals which don't require debouncing, the maximum input frequency is the same as the system clock.

In all modes, the three input pins can be optionally filtered in order to present clean signals to the subsequent rotary encoder logic. This feature can be enabled or disabled by way of the `DEBE` bit in the `CNT_CONFIG` register.

The filtering mechanism is implemented using counters for each pin. The counter for each pin is initialized from the `DPRESCALE` field of the `CNT_DEBOUNCE` register. Whenever a transition is detected on a pin, the corresponding counter starts counting up to the programmed number of `SCLK` cycles. The state of the pin is then latched after time t_{FILTER} , as determined by the equation below and passed on to the subsequent logic. The 5-bit `DPRESCALE` field in the `CNT_DEBOUNCE` register (see [Figure 13-12 on page 13-31](#)) is used to program the desired cycle number and therefore the debouncing time. The number of `SCLK` cycles used to program the counters for each pin can be selected in eighteen steps by way of this register, see [Table 13-1 on page 13-10](#).

The time t_{filter} is determined, given SCLK and the DPRESCALE value contained in the CNT_DEBOUNCE register, by the following formula:

$$t_{\text{FILTER}} = 128 \times (2^{\text{DPRESCALE}} \div \text{SCLK})$$

where, DPRESCALE can contain values from 0 (minimum filtering) to 17 (maximum filtering).

Figure 13-4 shows the filtering operation for the CUD pin.

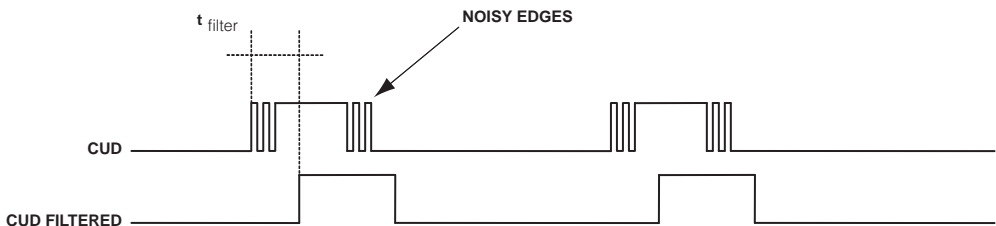


Figure 13-4. Programmable Noise Filtering

Assuming an SCLK frequency of 133 MHz, the filter time range is shown by the following two equations, [Figure 13-5 on page 13-10](#), [Table 13-1 on page 13-10](#), and [Table 13-2 on page 13-12](#):

$$\text{DPRESCALE} = 0b00000: t_{\text{FILTER}} = 128 * 1 * 7.5 \text{ ns} = 960 \text{ ns} = (\text{approx.}) 1 \mu\text{s}$$

$$\text{DPRESCALE} = 0b10001: t_{\text{FILTER}} = 128 * 131072 * 7.5 \text{ ns} = 125829 \mu\text{s} = (\text{approx.}) 128 \text{ ms}$$

Functional Description

CNT_DEBOUNCE
5 bits, R/W

@RESET

0	0	0	0	0
---	---	---	---	---

 0x00

SEE TABLE BELOW FOR VALUES

Figure 13-5. Filtering Range

Table 13-1. DPRESCALE Filtering Range

DPRESCALE	Cycles	Debounce Time
00000	1x	~1 μ s
00001	2x	~2 μ s
00010	4x	~4 μ s
00011	8x	~8 μ s
00100	16x	~16 μ s
001001	32x	~32 μ s
00110	64x	~64 μ s
00111	128x	~128 μ s
01000	256x	~256 μ s
01001	512x	~512 μ s
01010	1024x	~1 ms
01011	2048x	~2 ms
01100	4096x	~4 ms
01101	8192x	~82 ms
01110	16384x	~16 ms
01111	32768x	~32 ms

Table 13-1. DPRESCALE Filtering Range (Cont'd)

DPRESCALE	Cycles	Debounce Time
10000	65536x	~64 ms
10001	131072x	~128 ms

Zero Marker (Pushbutton) Operation

The `CZM` input pin can be used to sense the zero marker output of a rotary device or detect pressing of a pushbutton. There are four programming schemes all of which are functional in all counter modes. They are listed as follows:

- **Pushbutton mode:** This mode is enabled by setting the `CZMIE` bit in the `CNT_IMASK` register. An active edge at the `CZM` input sets the `CZMII` bit in the `CNT_STATUS` register. If enabled by the peripheral interrupt controller, this generates an interrupt request. The active edge is selected by the `CZMINV` bit in the `CNT_CONFIG` register: rising edge if cleared, falling edge if set to one.

Functional Description


Table 13-2. Prescale Value Programming to Debounce Filter Circuit

Bit Location	Name	Type	Function
4:0	DPRESCALE	R/W	<p>These bits are used to program the prescale value to the debounce filter circuit in the counter module. The predefined count value for “x” (128) determines the number of SCLK cycles to be counted.</p> <p>00000b: 1x cycles = 128 SCLK cycles</p> <p>00001b: 2x cycles = 256 SCLK cycles</p> <p>00010: 4x cycles = 512 SCLK cycles</p> <p>00011: 8x cycles = 1024 SCLK cycles</p> <p>00100: 16x cycles = 2048 SCLK cycles</p> <p>00101: 32x cycles = 4056 SCLK cycles</p> <p>00110: 64x cycles = 8112 SCLK cycles</p> <p>00111: 128x cycles = 16224 SCLK cycles</p> <p>01000b: 256x cycles = 32448 SCLK cycles</p> <p>01001b: 512x cycles = 64896 SCLK cycles</p> <p>01010b: 1024x cycles = 129792 SCLK cycles</p> <p>01011b: 2048x cycles = 259584 SCLK cycles</p> <p>01100b: 4096x cycles = 519168 SCLK cycles</p> <p>011001b: 8192x cycles = 1038336 SCLK cycles</p> <p>01110b: 16384x cycles = 2076672 SCLK cycles</p> <p>01111b: 32768x cycles = 4153344 SCLK cycles</p> <p>10000b: 65536x cycles = 8306688 SCLK cycles</p> <p>10001b: 131072x cycles = 16613376 SCLK cycles</p> <p>10010b - 11111b: Reserved</p>

- **Zero-marker-zeros-counter mode:** This mode is enabled by setting the ZMZC bit in the CNT_CONFIG register. An active level at the CZM input clears the CNT_COUNTER register and holds it until the CZM pin is deactivated. In addition, if enabled by the CZMZIE bit in the CNT_IMASK register, this mode sets the CZMZII bit in the CNT_STATUS register. If enabled by the peripheral interrupt controller, this generates an interrupt request. The active level is selected by the CZMINV bit in the CNT_CONFIG register: active high if cleared, active low if set to one.
- **Zero-marker-error mode:** This mode is used to detect discrepancies between the counter value and the zero marker output of certain rotary encoder devices. It is enabled by setting the CZMEIE bit in the CNT_IMASK register. When an active edge is detected at the CZM input pin, the four LSBs of the CNT_COUNTER register are compared to zero. If they are not zero, a mismatch is signaled by way of the CZMEII bit in the CNT_STATUS register. If enabled by the peripheral interrupt controller, this mode generates an interrupt request. The active edge is selected by the CZMINV bit in the CNT_CONFIG register: rising edge if cleared, falling edge if set to one.
- **Zero-once mode:** This mode is used to perform an initial reset of the counter value when an active zero marker is detected. After that, the zero marker is ignored (the counter is not reset anymore). This mode is enabled by setting the W1ZMONCE bit in the CNT_COMMAND register. The CNT_COUNTER register and the W1ZMONCE bit are cleared on the next active edge on the CZM pin. Thus, the W1ZMONCE bit can be read to check whether the event has already occurred, if desired. The active edge of the CZM pin is selected by the CZMINV bit in the CNT_CONFIG register: rising edge if cleared, falling edge if set to one.

Boundary Comparison Modes

The rotary encoder block includes two boundary registers, `CNT_MIN` (lower) and `CNT_MAX` (upper). The counter value is compared to the lower and upper boundary. Depending on which mode is selected, different actions are taken if the count value reaches either of the boundary values.

 For all boundary modes, compares do not occur if the change to `CNT_MIN/CNT_MAX/CNT_COUNTER` was due to a software event. Software events include writing these registers, or events caused by writing the `CNT_COMMAND` register. Boundary compare events **ONLY** occur due to up/down actions from the counter. This includes setting `MINCII/MAXCII` and zeroing the counter on a compare to either `CNT_MIN` or `CNT_MAX`.

There are four boundary modes:

- **Boundary-compare mode:** The two boundary registers are simply compared to the `CNT_COUNTER` register. If `CNT_COUNTER` after incrementing equals `CNT_MAX`, the `MAXCII` bit in the `CNT_STATUS` register is set. If the `MAXCIE` bit in the `CNT_IMASK` register is set, an interrupt request is generated. Similarly `CNT_COUNTER` after decrementing equals `CNT_MIN`, the `MINCII` status bit is set. If the `MINCIE` bit in the `CNT_IMASK` register is set, an interrupt request is generated. The `MAXCII` and `MINCII` bits are not set if the `CNT_MAX` and `CNT_MIN` registers are updated by software. For `MINCII` and `MAXCII` to be set, all that needs to happen is for `CNT_COUNTER` to equal them, regardless of the direction. As an example, if `CNT_MIN=2` and `CNT_COUNTER=1` and an up event occurs, `MINCII` will still be set. Likewise, if `CNT_MAX=2`, `CNT_COUNTER=3` and a down event occurs, `MAXCII` will still be set.

For the special case of `CNT_MIN` equals `CNT_MAX`, if `CNT_COUNTER` reaches the value in the boundary register both `MINCII` and `MAXCII` are set.

- **Boundary-zero mode:** This mode is similar to the boundary-compare mode. In addition to setting the status bits and requesting interrupts, the counter value in the `CNT_COUNTER` register is also set to zero.
- **Boundary auto-extend mode:** In this mode, the boundary registers are modified by hardware whenever the counter value reaches either of them. At startup, the application software should set both boundary registers to the initial `CNT_COUNTER` value. The `CNT_MAX` register is loaded with the current `CNT_COUNTER` value if the latter increments beyond the `CNT_MAX` value. Similarly, the `CNT_MIN` register is loaded with the `CNT_COUNTER` value if the latter decrements below the `CNT_MIN` value. This mode may be used to keep track (in hardware) of the widest angle the wheel ever reported, even if the software did not serve interrupts. The `MAXCII` and `MINCII` status bits are still set when the counter value matches the boundary register, not only when it extends the boundary.

In this mode it is envisioned that software would never change `CNT_MIN` or `CNT_MAX` by writing to them or an action from the `CNT_COMMAND` register. If software does this, the behavior is best described by a few examples:

Example 1: `CNT_MAX=2`, `CNT_COUNTER=1`. With three up events, `CNT_COUNTER=CNT_MAX=4`. Now if software writes `CNT_MAX=2`, the rotary will not auto extend until `CNT_COUNTER` decrements back down to two, then increments again.

Example 2: `CNT_MIN=2`, `CNT_COUNTER=3`. With three down events `CNT_COUNTER=CNT_MIN=0`. Now if software writes `CNT_MIN=2`, the rotary will not auto extend until `CNT_COUNTER` increments back up to two, then decrements again.

- **Boundary-capture mode:** In this mode, the `CNT_COUNTER` value is latched into the `CNT_MIN` register at one detected edge of the `CZM` input pin, and latched into `CNT_MAX` at the opposite edge. If the

Functional Description

CZMINV bit in the CNT_CONFIG register is cleared, a rising edge captures into CNT_MIN and a falling edge into CNT_MAX. If the CZMINV bit is set, the edges are inverted. The MAXCII and MINCII status bits report the capture event.

The comparison is performed with signed arithmetic. The boundary registers and the counter value are all treated as signed integer values.

Rotary Encoder Events: Control and Signaling

There are a total of 11 events that can be signaled to the processor by way of status information and optional interrupt requests. The interrupts are enabled by the respective bits in the CNT_IMASK register. Dedicated status bits in the CNT_STATUS register report events. When an interrupt from the rotary encoder is acknowledged, the application software is responsible for correct interpretation of the events. It is recommended to logically AND the content of the CNT_IMASK and CNT_STATUS registers to identify pending interrupts. Interrupt requests are cleared by write-one-to-clear (W1C) operations to the CNT_STATUS register. Hardware does not clear the status bits automatically, unless the counter module is disabled. There are four boundary modes. Status bits are available in any of the counter modes discussed in [“Description of Operation” on page 13-4](#).

Illegal Gray/Binary Code Events (Two-Step Detection)

As described in the quadrature encoder mode and binary encoder mode sections, illegal transitions can be detected in these two modes. If this event occurs, the ICII status bit is set. If enabled by the ICIE bit, an interrupt request is generated. The ICIE bit should only be used (set) in these two modes.

Up/Down Count Events

The UCII status bit informs whether the counter is incremented. Similarly, the DCII bit reports decrements. The two events are independent. For instance, if the counter first increments by one and then decrements by two, both bits remain set, even though the resulting counter value shows a decrement by one. In rotary counter mode, hardware may detect simultaneous active edges on the CUD and CDG inputs. In that case, the CNT_COUNTER remains unchanged, but both the UCII and DCII bits are set.

Interrupt requests for these events may be enabled through the UCIE and DCIE bits. This feature should be used carefully when the counter is clocked at high rates. This is especially critical when the counter operates in DIR_TMR mode, as interrupts would be generated every SCLK cycle.

These events can also be used for additional pushbuttons, if rotary encoder features are not needed. When rotary counter mode is enabled, these count events can be used to report interrupts from pushbuttons that connect to the CUD and CDG inputs.

Zero Count Events

The CZEROII status bit indicates that the CNT_COUNTER has reached a value equal to 0x0000 0000 after an increment or decrement. This bit is not set when the counter value is set to zero directly by way of a software write (write to CNT_COUNTER or setting the WILCNT_ZERO bit in the CNT_COMMAND register). If enabled by the CZEROIE bit, an interrupt request is generated.

Overflow Events

There are two status bits that indicate whether the signed counter register has overflowed from a positive to a negative value or vice versa.

Functional Description

The `COV31II` bit reports that the 32-bit `CNT_COUNT` register has either incremented from `0x7FFF.FFFF` to `0x8000.0000` or decremented from `0x8000.0000` to `0x7FFF.FFFF`. If enabled by the `COV31IE` bit, an interrupt request is generated.

Similarly, in applications where only the lower 16 bits of the counter are of interest, the `COV15II` status bit reports counter transitions from `0xxxxx.7FFF` to `0xxxxx.8000` or reversed. If enabled by the `COV15IE` bit, an interrupt request is generated.

Boundary Match Events

The `MINCII` and `MAXCII` status bits report boundary events as described in “[Boundary Comparison Modes](#)” on page 13-14. These bits are not set if the `CNT_COUNTER`, `CNT_MAX` or `CNT_MIN` registers are updated by software or the `CNT_COMMAND` register is written to.

The `MINCIE` and `MAXCIE` bits in the `CNT_IMASK` register enable interrupt generation on boundary events.

Zero Marker Events

There are three status bits `CZMII`, `CZMEII` and `CZMZII` associated with zero marker events, as described in “[Zero Marker \(Pushbutton\) Operation](#)” on page 13-11. Each of these events can optionally generate an interrupt request, if enabled by the corresponding `CZMIE`, `CZMEIE` and `CZMZIE` bits in the `CNT_IMASK` register.

Capturing Timing Information (Using the General-Purpose Timer)

In many applications, in addition to accurate count encoder pulses (count events), it is important to measure the time between two count events. This information allows for calculating speed. For more accuracy, particularly at very low speeds, it is also necessary to obtain the time that has

elapsed since the last count event. This additional information allows for estimating how much the rotary encoder has advanced since the last counter event.

For this purpose, the rotary counter has an internal timer output that connects to the alternate capture inputs (`TACIX`) of one of the timers as explained in [“Interface Overview” on page 13-3](#). It is functional in all modes, with the exception of the timed direction mode.

In order to use the timing measurements, the associated timer must be used in pulse width count and capture mode (`WDTH_CAP`). The alternative capture input is selected by setting the `TIN_SEL` bit in the timer’s configuration register. For more information about the GP Timers and their operating modes refer [“Capturing Timings from the GP Counter Module” on page 10-36](#).

Capturing Time Interval Between Successive Counter Events

When the only timing information of interest is the interval between successive count events, the associated timer should be programmed in `WDTH_CAP` mode with `PULSE_HI=1`, `PERIOD_CNT=1` and `TIN_SEL=1`. Typically, this information is sufficient if the speed of rotary encoder events is known not to reach very low values. [Figure 13-6 on page 13-20](#) shows the operation of the rotary encoder module and the GP timer in this mode. TO generates a pulse every time a count event occurs. The general-purpose timer will update the `TIMERx_PERIOD` register with the period (measured from rising edge to rising edge) of the TO signal. The `TIMERx_PERIOD` register is updated at every rising edge of the TO signal and contains the number of system clock (`SCLK`) cycles that have elapsed since the previous rising edge. Incidentally, the `TIMERx_WIDTH` register is also updated at the same time, but is generally of no interest in this mode of operation. If no reads of the `CNT_COUNTER` register occur between counter events, the `TIMERx_WIDTH` register only contains the width of the

Functional Description

TO pulse. If a read of the `CNT_COUNTER` has occurred between events, the `TIMERx_WIDTH` register will contain the time between the read of the `CNT_COUNTER` and the next event.

This mode can also be used with `PULSE_HI=0`. In this case, the period of TO is measured between falling edges. It will result in the same values as in the previous case, only the latching occurs one `SCLK` cycle later.

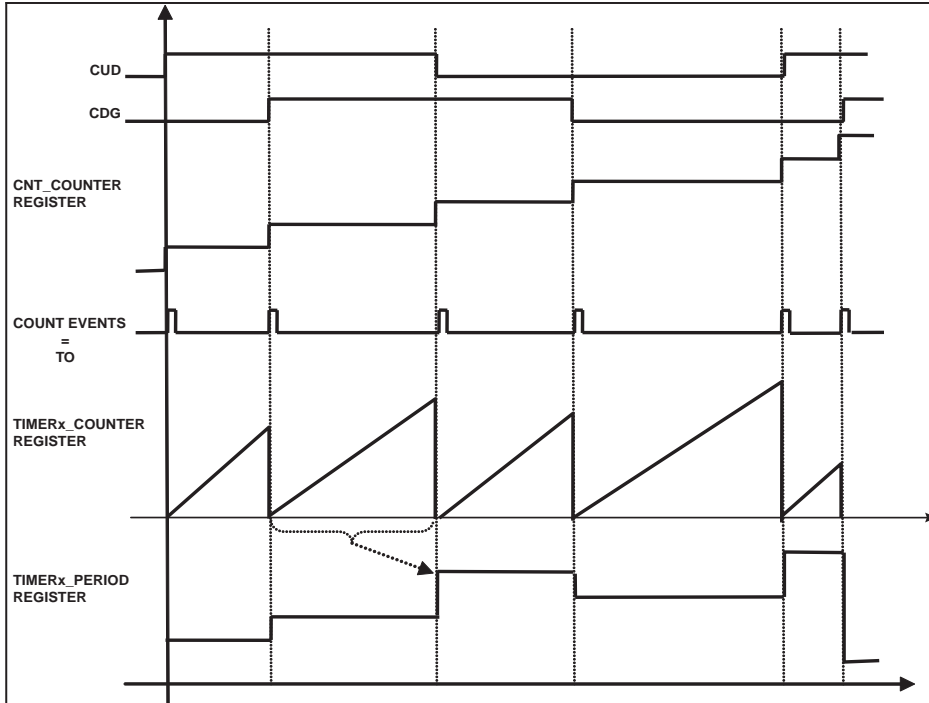



Figure 13-6. Timer Period Register

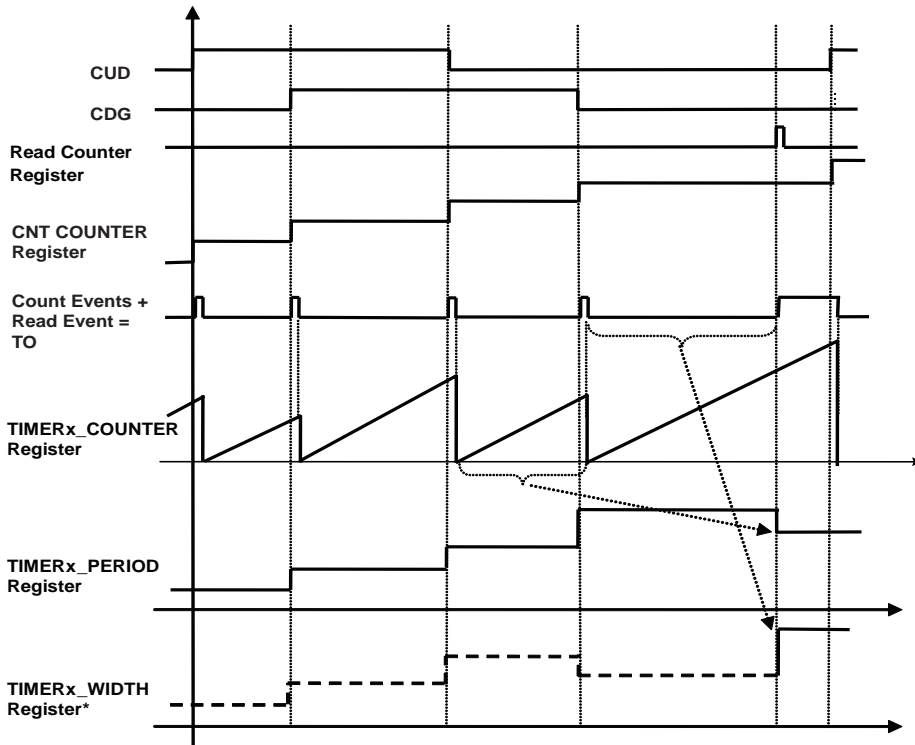
Capturing Counter Interval and CNT_COUNTER Read Timing

It is possible to also capture the time elapsed since the last count event. In this mode, the associated timer should be programmed in `WIDTH_CAP` mode with `PULSE_HI=0`, `PERIOD_CNT=0` and `TIN_SEL=1`. Typically, this additional information is used to estimate the advancement of the rotary encoder since the last count event, if the speed is very low. [Figure 13-7](#) shows the operation of the rotary encoder module and the general-purpose timer in this mode. TO generates a pulse every time a count event occurs. In addition, when the processor reads the `CNT_COUNTER` register, the TO signal presents a pulse which is extended (high) until the next count event. The general-purpose timer will update the `TMRX_PERIOD` register with the period (measured from falling edge to falling edge, because `PULSE_HI=0`) of the TO signal. The `TMRX_WIDTH` register is updated with the pulse width (the portion where TO is low, again because `PULSE_HI=0`). Both registers are updated at every rising edge of the TO signal (because `PERIOD_CNT=0`). Therefore, the period register will contain the period between the last two count events and the width register will contain the time since the last count event and the read of the `CNT_COUNTER` register, both measured in number of system clock (`SCLK`) cycles.

The result is that when reading the `CNT_COUNTER` register, the two time measurements are also latched and the user has a coherent triplet of information to calculate speed and position.

-  Restrictions apply to the use of the TO signal in terms of speed. Therefore, the user must take care to not operate at very high count events. For instance, if `CNT_COUNTER` is incremented/decremented every `SCLK` cycle (timed direction mode), the TO signal is incorrect.

Functional Description



* The solid line indicates the time between the last event and the CNT_COUNTER read. Register contents marked with dotted lines do not reflect relevant time measurements and should be ignored.

Figure 13-7. Timer Registers

Counter Commands

In order to facilitate initialization of the peripheral, a register is provided to perform various operations such as zeroing a counter register, copying or swapping boundary registers and so on. These actions are taken by writing a 1 to the appropriate bit in the `CNT_COMMAND` register.

The `CNT_COUNTER`, `CNT_MIN` and `CNT_MAX` registers can be initialized to zero by writing a 1 to the `W1LCNT_ZERO`, `W1LMIN_ZERO` and `W1LMAX_ZERO` fields. In addition to clearing registers, the boundary registers can be modified in a number of ways. The current counter value in `CNT_COUNT` can be captured and loaded into either of the two boundary registers `CNT_MAX` and `CNT_MIN` to create new boundary limits. This is performed by setting the `W1LMAX_CNT` and `W1LMIN_CNT` bits. Alternatively the counter can be loaded from `CNT_MAX` or `CNT_MIN` through the `W1LCNT_MAX` and `W1LCNT_MIN` bits. It is also possible to transfer the current `CNT_MAX` into `CNT_MIN` or vice versa through the `W1LMIN_MAX` and `W1LMAX_MIN` bits. The final supported operation is the ability to only have the zero marker clear the `CNT_COUNT` register once as described in [“Zero Marker \(Pushbutton\) Operation” on page 13-11](#).

It is possible for multiple actions to be performed simultaneously by setting multiple bits in the `CNT_COMMAND` register. The bits associated with each command have been grouped together such that all bits that involve a write to the `CNT_COUNTER` register are located within the bits 3:0 of the `CNT_COMMAND` register. All commands that involve a write to the `CNT_MIN` register are located within bits 7:4 of the `CNT_COMMAND` register and all commands that involve a write to the `CNT_MAX` register are located within bits 11:8 of the `CNT_COMMAND` register. Please refer to the register diagram ([Figure 13-11 on page 13-30](#)) for more details.



A maximum of three commands can be issued at any one time, excluding the `W1ZMONCE` command. No two commands issued simultaneously can involve a load to the same counter register. The following commands must be used exclusively: `W1LCNT_MIN`, `W1LCNT_MAX`, and `W1LCNT_ZERO`. Never set more than one of them at

Programming Mode

the same time. The same requirement stands true for `W1LMAX_MIN`, `W1LMAX_CNT` and `W1LMAX_ZERO` and also for `W1LMIN_MAX`, `W1LMIN_CNT`, and `W1LMIN_ZERO`.

Programming Mode

In a typical application, the user initializes the rotary encoder to the desired mode, without enabling it. Normally, it is chosen to process the events of interest by way of interrupts rather than polling the status bit. Therefore, clear all status bits and activate the generation of interrupt requests by way of the `CNT_IMASK` register. Set up the peripheral interrupt controller and core interrupts. If timing information is required, set up appropriate timer in `WDTH_CAP` mode with the settings described in “[Capturing Timing Information \(Using the General-Purpose Timer\)](#)” on [page 13-18](#). Then, enable interrupts and the peripheral itself.

Rotary Counter Registers

The rotary encoder interface has eight memory-mapped registers (MMRs) that regulate its operation.

Refer to [Table 13-3](#) for an overview of all MMRs associated with the rotary encoder interface.

Table 13-3. Counter Module Register Overview

Address	Register Name	Width	PAB Operation	Reset Value
0xFFC04200	<code>CNT_CONFIG</code> (on page 13-26)	16 bits	R/W	0x0000
0xFFC04204	<code>CNT_IMASK</code> (on page 13-28)	16 bits	R/W	0x0000
0xFFC04208	<code>CNT_STATUS</code> (on page 13-29)	16 bits	R/W1C	0x0000


Table 13-3. Counter Module Register Overview (Cont'd)

Address	Register Name	Width	PAB Operation	Reset Value
0xFFC0420c	CNT_COMMAND (on page 13-29)	16 bits	R/W1ACTION	0x0000
0xFFC04210	CNT_DEBOUNCE (on page 13-31)	16 bits	R/W	0x0000
0xFFC04214	CNT_COUNTER (on page 13-32)	32 bits	R/W (16/32 bits)	0x0000 0000
0xFFC04218	CNT_MAX (on page 13-32)	32 bits	R/W (16/32 bits)	0x0000 0000
0xFFC0421c	CNT_MIN (on page 13-32)	32 bits	R/W (16/32 bits)	0x0000 0000

Descriptions and bit diagrams for MMRs are provided in the following sections.

Configuration (CNT_CONFIG) Register

The configuration (CNT_CONFIG) register is used to configure counter modes and input pins and to enable the peripheral. It can be accessed at any time with 16-bit read and write operations.

-  To avoid false glitches on startup, write all bits in CNT_CONFIG first followed by a second write to the register which enables the counter (CNTE=1).

Boundary Register Mode

Since CUD, CDG, and CZM input pins are muxed with other pins, these pins might be used for a function other than rotary counter. Specifically:

- If the application needs only the pushbutton (CZM) function, then write INPDIS=0 to ignore CUD and CDG. This allows a debounced pushbutton interrupt source.
- If the application needs just the rotary pins CUD and CDG, but not the CZM, then write INPDIS=1. Then ensure your software does not enable any of the pushbutton functions in the rotary counter registers.

For further information, see BNDMODE bit in [Figure 13-8](#).

Configuration Register (CNT_CONFIG)

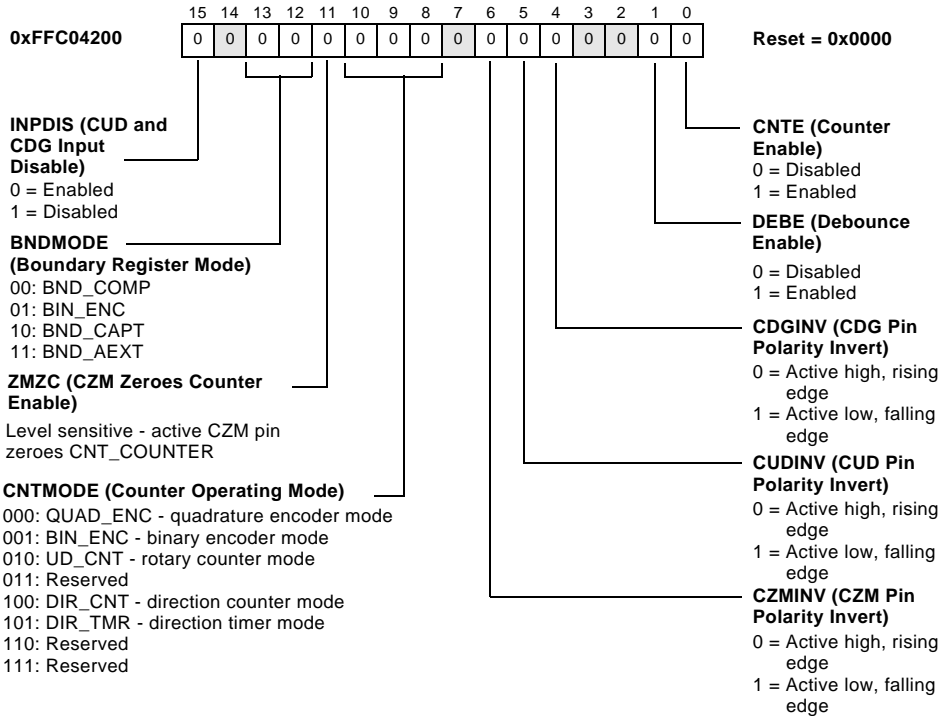


Figure 13-8. Configuration (CNT_CONFIG) Register

Interrupt Mask (CNT_IMASK) Register

The interrupt mask (CNT_IMASK) register is used to enable interrupt request generation from each of the eleven events (See [Figure 13-9](#)). It can be accessed at any time with 16-bit read and write operations.

Interrupt Mask Register (CNT_IMASK)

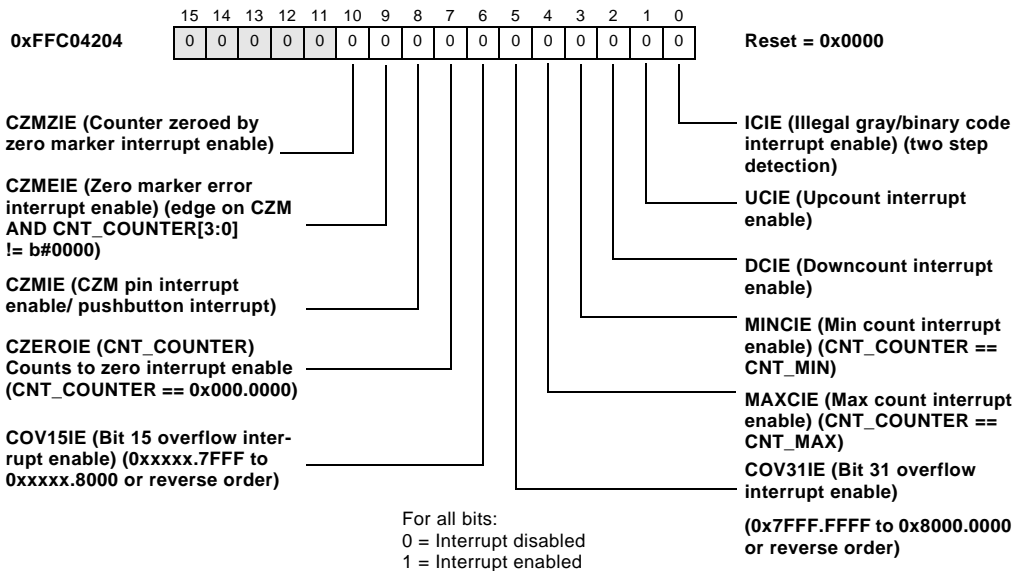


Figure 13-9. Interrupt Mask (CNT_IMASK) Register

Status (CNT_STATUS) Register

The status (CNT_STATUS) register provides status information for each of the eleven events where 0 = no interrupt pending and 1 = interrupt pending (See [Figure 13-10](#)). When an event is detected, the corresponding bit in this register is set. It remains set until either software writes a 1 to the bit (write-1-to-clear) or the rotary encoder peripheral is disabled.

Status Register (CNT_STATUS)

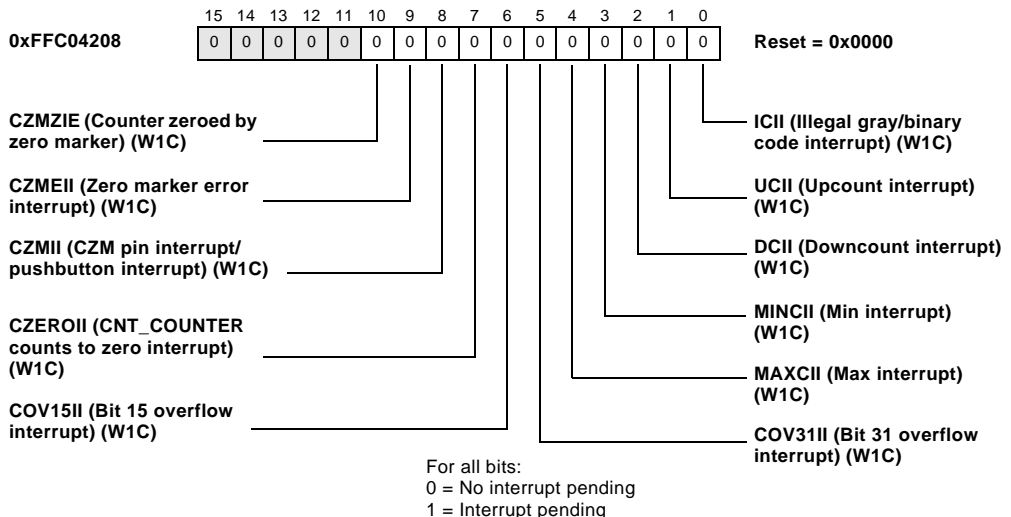


Figure 13-10. Status (CNT_STATUS) Register

Command (CNT_COMMAND) Register

The command (CNT_COMMAND) register is used to perform various actions that are needed occasionally. Each bit performs the indicated action when a 1 is written to it (See [Figure 13-11](#)).

Rotary Counter Registers

Read operations from this register do not return meaningful values. One exception is the `W1ZMONCE` bit. It is the only bit that returns a value if the register is read. A one indicates that the bit is set by software before, but no zero marker event is detected on the `CZM` pin yet. Refer to the [“Zero Marker \(Pushbutton\) Operation”](#) on page 13-11 for more details.

i Note that `W1LCNT_MIN`, `W1LCNT_MAX` and `W1LCNT_ZERO` have to be used exclusively. Never set more than one of them at the same time. The same requirement stands for `W1LMAX_MIN`, `W1LMAX_CNT` and `W1LMAX_ZERO` and also for `W1LMIN_MAX`, `W1LMIN_CNT` and `W1LMIN_ZERO`.

Command Register (CNT_COMMAND)

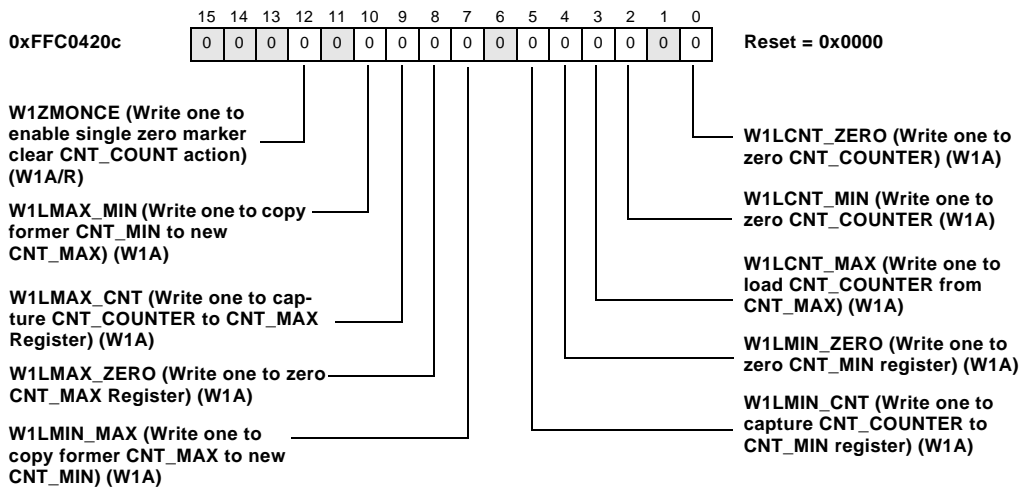


Figure 13-11. Command (CNT_COMMAND) Register

Debounce Prescale (CNT_DEBOUNCE) Register

The debounce prescale (CNT_DEBOUNCE) register is used to select the noise filtering characteristic of the input pins (See [Figure 13-12](#)). Bits [4:0] determine the filter time. The register can be accessed at any time with 16-bit read and write operations.

$$t_{\text{filter}} = 128 \times (2^{\text{DPRESCALE}} \div \text{SCLK})$$

Debounce Register (CNT_DEBOUNCE)

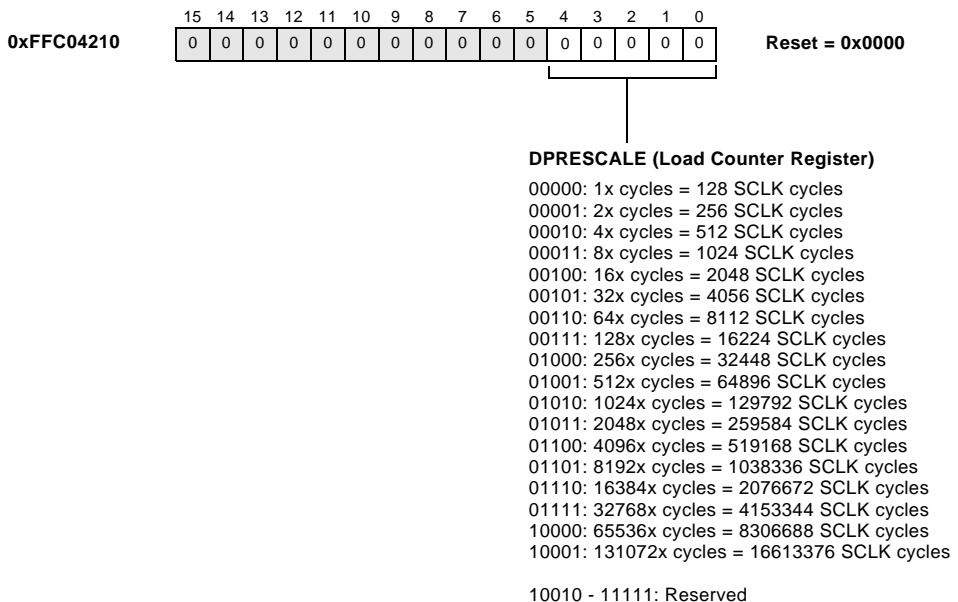


Figure 13-12. Debounce (CNT_DEBOUNCE) Register

Rotary Counter Registers

Counter (CNT_COUNTER) Register

The counter (CNT_COUNTER) register holds the 32-bit, two's-complement, count value (See [Figure 13-13](#)). It can be read and written at any time. Hardware ensures that reads and writes are atomic, by providing respective shadow registers. This register can be accessed with either 32-bit or 16-bit operations. This allows use of the rotary encoder as a 16-bit counter, if sufficient for the application.

Counter Register (CNT_COUNTER)

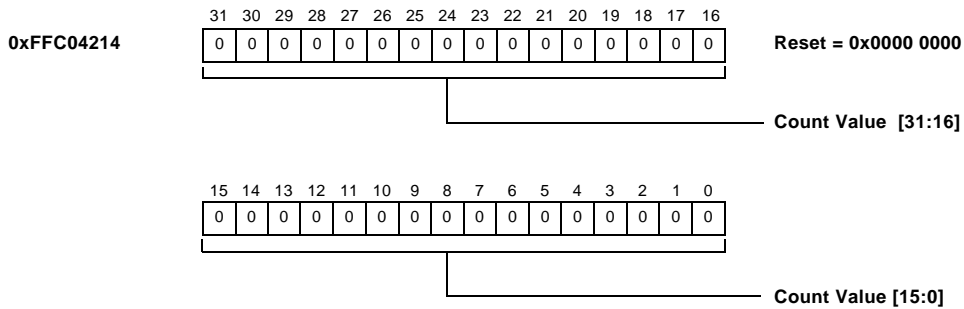


Figure 13-13. Counter (CNT_COUNTER) Register

Boundary (CNT_MIN and CNT_MAX) Registers

The boundary (CNT_MIN and CNT_MAX) registers hold the 32-bit, two's-complement, lower and upper boundary values (See [Figure 13-14](#) and [Figure 13-15](#)). They can be read from and written to at any time. Hardware ensures that reads and writes are atomic, by providing respective shadow registers. This register can be accessed with either 32-bit or 16-bit operations. This allows for using the rotary encoder as a 16-bit counter if sufficient for the application.

Maximal Count Register (CNT_MAX)

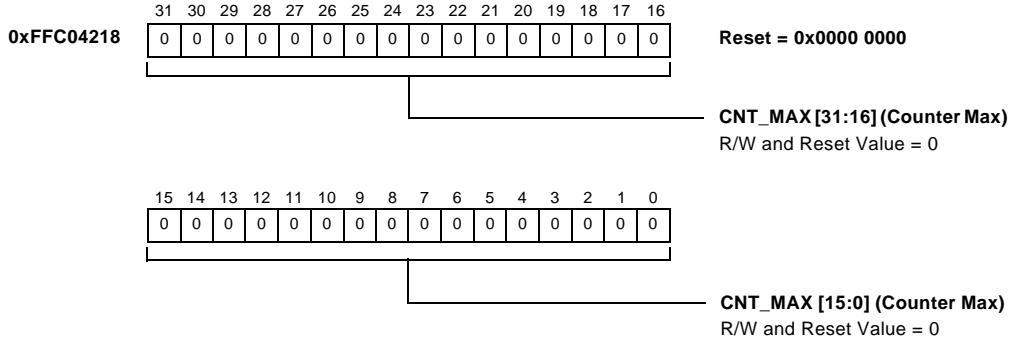


Figure 13-14. Maximal Count (CNT_MAX) Register

Minimal Count Register (CNT_MIN)

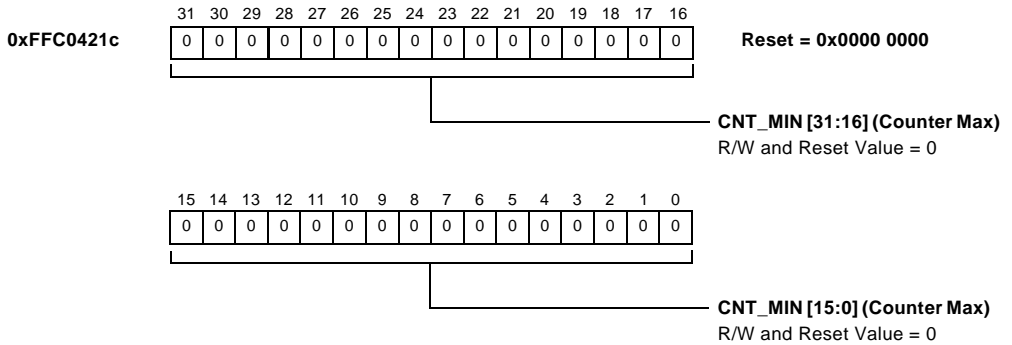


Figure 13-15. Minimal Count (CNT_MIN) Register

Programming Examples

[Listing 13-1](#) illustrates how to configure the port registers to enable rotary counter functionality through the `PORTx_MUX` and `PORTx_FER` registers.

Listing 13-1. Configuring the Port Registers to Enable Rotary Counter

```
/* enable CDG and CUD features. */
P5.H = hi(PORTH_FER);
P5.L = lo(PORTH_FER);
R5.L = nPH15 | nPH14 | nPH13 | nPH12 | PH11 | nPH10 | nPH9 | nPH8
      | nPH7 | nPH6 | nPH5 | PH4 | PH3 | nPH2 | nPH1 | nPH0;
w[P5] = R5.L;

/* enable CZM feature. */
P5.H = hi(PORTG_FER);
P5.L = lo(PORTG_FER);
R5.L = nPG15 | nPG14 | nPG13 | nPG12 | PG11 | nPG10 | nPG9 | nPG8
      | nPG7 | nPG6 | nPG5 | nPG4 | PG3 | nPG2 | nPG1 | nPG0;
w[P5] = R5.L;

/* enable CDG and CUD MUX mode. */
P5.H = hi(PORTH_MUX);
P5.L = lo(PORTH_MUX);
R5.H = hi(MUX(0,0,0,0,0,0,0,0,0,0,0,0,2,2,0,0,0));
R5.L = lo(MUX(0,0,0,0,0,0,0,0,0,0,0,0,2,2,0,0,0));
[P5] = R5;

/* enable CZM MUX mode. */
P5.H = hi(PORTG_MUX);
P5.L = lo(PORTG_MUX);
R5.H = hi(MUX(0,0,0,0,0,0,0,0,0,0,0,0,3,0,0,0,0));
R5.L = lo(MUX(0,0,0,0,0,0,0,0,0,0,0,0,3,0,0,0,0));
```

```
[P5] = R5;
```

[Listing 13-2](#) illustrates how to initialize the rotary counter for a required functionality. The required rotary counter interrupts are first unmasked. The rotary counter is then configured for the required mode of operation. Note at this point we do not enable the rotary counter. Finally we clear some of the rotary counter registers before also clearing any pending interrupts that may be pending in the CNT_STATUS register.

Listing 13-2. Initializing the Rotary Counter

```
/* Setup Counter Interrupts */
P5.H = hi(CNT_IMASK);
P5.L = lo(CNT_IMASK);
R5 = nCZMZIE /* Counter zeroed by zero marker interrupt */
    | CZMEIE /* Zero marker error interrupt */
    | CZMIE /* CZM pin interrupt (pushbutton) */
    | CZEROIE /* Counts to zero interrupt */
    | nCOV15IE /* Counter bit 15 overflow interrupt */
    | nCOV31IE /* Counter bit 31 overflow interrupt */
    | MAXCIE /* Max count interrupt */
    | MINCIE /* Min count interrupt */
    | DCIE /* Downcount interrupt */
    | UCIE /* Upcount interrupt */
    | ICIE(z); /* Illegal gray/binary code interrupt */
w[P5] = R5;

/* Configure the Rotary Counter mode of operation */
P5.H = hi(CNT_CONFIG);
P5.L = lo(CNT_CONFIG);
R5 = nINPDIS /* Enable CUD and CDG inputs */
    | BNDMODE_COMP /* Boundary compare mode */
    | nZMZC /* Disable Zero Counter Enable */
    | CNTMODE_QUADENC /* Quadrature Encoder Mode */
```

Programming Examples

```
| CZMINV          /* Polarity of CZM pin */
| nCUDINV        /* Polarity of CUD pin */
| nCDGINV        /* Polarity of CDG Pin */
| nDEBE          /* Disable the debounce */
| nCNTE (z);     /* Disable the counter */
w[P5] = R5;

/* Zero the CNT_COUNT, CNT_MIN and CNT_MAX registers
This is optional as after reset they are default to zero */
P5.H = hi(CNT_COMMAND);
P5.L = lo(CNT_COMMAND);
R5 = W1LCNT_ZERO | W1LMIN_ZERO | W1LMAX_ZERO (z);
w[P5] = R5;

/* Clear any identified interrupts */
P5.H = hi(CNT_STATUS);
P5.L = lo(CNT_STATUS);
R5.L = ICII      /* Illegal Gray/Binary Code Interrupt Identifier
*/
| UCII /* Up count Interrupt Identifier */
| DCII /* Down count Interrupt Identifier */
| MINCII /* Min Count Interrupt Identifier */
| MAXCII /* Max Count Interrupt Identifier */
| COV31II /* Bit 31 Overflow Interrupt Identifier */
| COV15II /* Bit 15 Overflow Interrupt Identifier */
| CZEROII /* Count to Zero Interrupt Identifier */
| CZMII /* CZM Pin Interrupt Identifier */
| CZMEII /* CZM Error Interrupt Identifier */
| CZMZII; /* CZM Zeroes Counter Interrupt Identifier */
w[P5] = R5;
```


Listing 13-3 illustrates how to set up the peripheral and core interrupts for the rotary counter. The counter interrupts generated on IRQ68 are mapped to the IVG7 interrupt. Finally the system and peripheral interrupts are unmasked and then the rotary counter is enabled.

Listing 13-3. Setting Up the Interrupts for the Rotary Counter

```

/* Assign the CNT interrupt to IVG7 */
P5.H = hi(SIC_IAR8);
P5.L = lo(SIC_IAR8);
R6.H = hi(0xFFFF0FFFF);
R6.L = lo(0xFFFF0FFFF);
R7.H = hi(0x00000000);
R7.L = lo(0x00000000);
R5 = [P5];
R5 = R5 & R6; /* zero the Counter interrupt field */
R5 = R5 | R7; /* set Counter interrupt to required priority */
[P5] = R5;

/* Set up the interrupt vector for the rotary counter */
R5.H = hi(_IVG7_handler);
R5.L = lo(_IVG7_handler);
P5.H = hi(EVT7);
P5.L = lo(EVT7);
[P5] = R5;

/* Unmask IVG7 interrupt in the IMASK register */
P5.H = hi(IMASK);
P5.L = lo(IMASK);
R5 = [P5];
bitset(R5, bitpos(EVT_IVG7));
[P5] = R5;

/* Unmask interrupt 68 generated by the counter */

```

Programming Examples

```
P5.H = hi(SIC_IMASK2);
P5.L = lo(SIC_IMASK2);
R5 = [P5];
bitset(R5, bitpos(IRQ_CNT));
[P5] = R5;

/* Enable the Rotary Counter */
P5.H = hi(CNT_CONFIG);
P5.L = lo(CNT_CONFIG);
R5 = w[P5](z);
bitset(R5, bitpos(CNTE));
w[P5] = R5.L;
```

[Listing 13-4](#) illustrates a sample interrupt handler that is responsible for servicing the rotary counter interrupts. On entry to the handler, the SIC_ISR2 register is interrogated to determine if the counter is waiting for a service interrupt. If a counter interrupt is waiting to be serviced, then the handler that is responsible for processing all counter interrupts is called.

Listing 13-4. Sample Interrupt Handler Rotary Counter Interrupts

```
_IVG7_handler:
    /* Stack management */
    [--SP] = RETS;
    [--SP] = ASTAT;
    [--SP] = (R7:0, P5:0);

    /* Was it a counter interrupt? */
    P5.H = hi(SIC_ISR2);
    P5.L = lo(SIC_ISR2);
    R5 = [P5];
    CC = bittst(R5, bitpos(IRQ_CNT));
    IF !CC JUMP _IVG7_handler.completed;
    CALL _IVG7_handler.counter;
```

```

_IVG7_handler.completed:

SSYNC;
/* Restore from stack */
(R7:0, P5:0) = [SP++];
ASTAT = [SP++];
RETS = [SP++];
RTI; /* Exit the interrupt service routine */
_IVG7_handler.end:

_IVG7_handler.counter:
/* Stack management */
[--SP] = RETS;
[--SP] = (R7:0, P5:0);

/* Determine what counter interrupts we wish to service */
P5.H = hi(CNT_IMASK);
P5.L = lo(CNT_IMASK);
R5 = w[P5](z);

P5.H = hi(CNT_STATUS);
P5.L = lo(CNT_STATUS);
R6 = w[P5](z);
R5 = R5 & R6;

/* Interrupt handlers for all rotary counter interrupts */
_IVG7_handler.counter.illegal_code:
CC = bittst(R5, bitpos(ICII));
IF !CC JUMP _IVG7_handler.counter.up_count;

/* Clear the serviced request */
R6 = ICII (z);
w[P5] = R6;

```

Programming Examples

```
        /* insert illegal code handler here */

_IVG7_handler.counter.illegal_code.end:

_IVG7_handler.counter.up_count:
    CC = bittst(R5, bitpos(UCII));
    IF !CC JUMP _IVG7_handler.counter.down_count;

    /* Clear the serviced request */
    R6 = UCII (z);
    w[P5] = R6;

    /* insert up count handler here */

_IVG7_handler.counter.up_count.end:

_IVG7_handler.counter.down_count:
    CC = bittst(R5, bitpos(DCII));
    IF !CC JUMP _IVG7_handler.counter.min_count;

    /* Clear the serviced request */
    R6 = DCII (z);
    w[P5] = R6;

    /* insert down count handler here */

_IVG7_handler.counter.down_count.end:

_IVG7_handler.counter.min_count:
    CC = bittst(R5, bitpos(MINCII));
    IF !CC JUMP _IVG7_handler.counter.max_count;
```

```

/* Clear the serviced request */
R6 = MINCII (z);
w[P5] = R6;

/* insert min count handler here */

_IVG7_handler.counter.min_count.end:

_IVG7_handler.counter.max_count:
    CC = bittst(R5, bitpos(MAXCII));
    IF !CC JUMP _IVG7_handler.counter.b31_overflow;

/* Clear the serviced request */
R6 = MAXCII (z);
w[P5] = R6;

/* insert max count handler here */

_IVG7_handler.counter.max_count.end:

_IVG7_handler.counter.b31_overflow:
    CC = bittst(R5, bitpos(COV31II));
    IF !CC JUMP _IVG7_handler.counter.b15_overflow;

/* Clear the serviced request */
R6 = COV31II (z);
w[P5] = R6;

/* insert bit 31 overflow handler here */

_IVG7_handler.counter.b31_overflow.end:

_IVG7_handler.counter.b15_overflow:

```

Programming Examples

```
CC = bittst(R5, bitpos(COV15II));
IF !CC JUMP _IVG7_handler.counter.count_to_zero;

/* Clear the serviced request */
R6 = COV15II (z);
w[P5] = R6;

/* insert bit 15 overflow handler here */

_IVG7_handler.counter.b15_overflow.end:

_IVG7_handler.counter.count_to_zero:
CC = bittst(R5, bitpos(CZEROII));
IF !CC JUMP _IVG7_handler.counter.czm;

/* Clear the serviced request */
R6 = CZEROII (z);
w[P5] = R6;

/* insert count to zero handler here */

_IVG7_handler.counter.count_to_zero.end:

_IVG7_handler.counter.czm:
CC = bittst(R5, bitpos(CZMII));
IF !CC JUMP _IVG7_handler.counter.czm_error;

/* Clear the serviced request */
R6 = CZMII (z);
w[P5] = R6;

/* insert czm handler here */

_IVG7_handler.counter.czm.end:
```

```
_IVG7_handler.counter.czm_error:
    CC = bittst(R5, bitpos(CZMEII));
    IF !CC JUMP _IVG7_handler.counter.czm_zeroes_counter;

    /* Clear the serviced request */
    R6 = CZMEII (z);
    w[P5] = R6;

    /* insert czm error handler here */

_IVG7_handler.counter.czm_error.end:

_IVG7_handler.counter.czm_zeroes_counter:
    CC = bittst(R5, bitpos(CZMZII));
    IF !CC JUMP _IVG7_handler.counter.all_serviced;

    /* Clear the serviced request */
    R6 = CZMZII (z);
    w[P5] = R6;

    /* insert czm zeroes counter handler here */

_IVG7_handler.counter.czm_zeroes_counter.end:

_IVG7_handler.counter.all_serviced:

    /* Restore from stack */
    (R7:0, P5:0) = [SP++];
    RETS = [SP++];
    RTS;
_IVG7_handler.counter.end:
```

Programming Examples

[Listing 13-5](#) illustrates how to set up timer 6 in order to capture the period of counter events. The timer is configured for `WDTH_CAP` mode and the period between the last two successive counter events is read from within the up count interrupt handler that was provided in [Listing 13-4](#).

Listing 13-5. Setting Up Timer 6 for Counter Event Period Capture

```
/* configure the timer for WDTH_CAP mode */
P5.H = hi(TIMER6_CONFIG);
P5.L = lo(TIMER6_CONFIG);
R5 = PULSE_HI | PERIOD_CNT | TIN_SEL | WDTH_CAP (z);
w[P5] = R5.L;

/* Enable Timer 6 */
P5.H = hi(TIMER_ENABLE0);
P5.L = lo(TIMER_ENABLE0);
R5 = TIMEN6 (z);
w[P5] = R5.L;

...

_IVG7_handler.counter.up_count:
    CC = bittst(R5, bitpos(UCII));
    IF !CC JUMP _IVG7_handler.counter.down_count;

/* Clear the serviced request */
R6 = UCII (z);
w[P5] = R6;

/* insert up count handler here */

/* Read the period between the last two successive events */
P5.H = hi(TIMER6_PERIOD);
P5.L = lo(TIMER6_PERIOD);
```



```
R5 = [P5];  
  
P5.H = hi(_event_period);  
P5.L = lo(_event_period);  
[P5] = R5;  
_IVG7_handler.counter.up_count.end:
```

Programming Examples

14 REAL-TIME CLOCK

This chapter describes the real-time clock (RTC) and includes the following sections:

- [“Overview” on page 14-1](#)
- [“Interface Overview” on page 14-3](#)
- [“Description of Operation” on page 14-5](#)
- [“RTC Programming Model” on page 14-7](#)
- [“RTC Registers” on page 14-20](#)
- [“Programming Examples” on page 14-24](#)

Overview

The 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, which counts the elapsed time since the last system reset.

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.

Overview

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 has elapsed, the RTC generates an interrupt.

Interface Overview

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 SIC (system interrupt controller).

Interface Overview

The RTC has dedicated power supply pins that power the clock functions at all times, including when the core power supply is turned off.

Figure 14-1 provides a block diagram of the RTC.

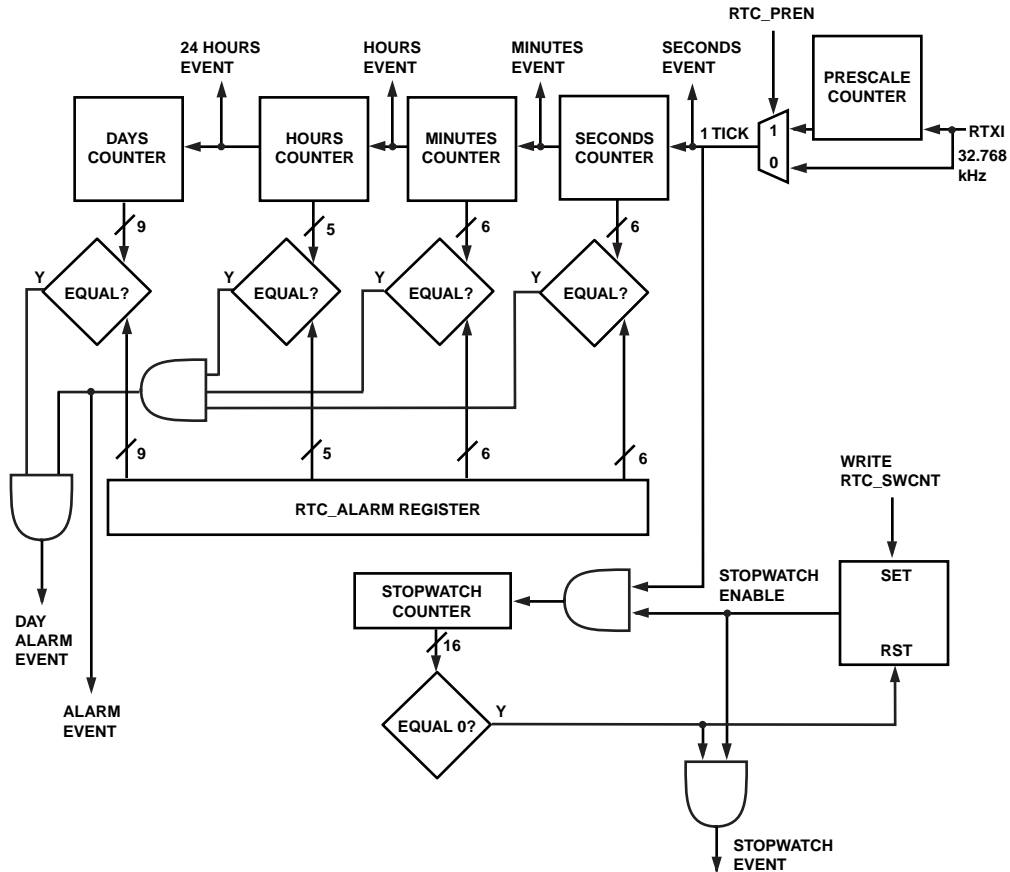


Figure 14-1. RTC Block Diagram

Description of Operation

The following sections describe the operation of the RTC.

RTC Clock Requirements

The RTC timer 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 `RTXI` and `RTCGND` pins to `EGND`, tie the `RTCVDD` pin to `EVDD`, and leave the `RTX0` pin unconnected. Additionally, writing `RTC_PREN` to 0 saves a small amount of power.

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 this bit is cleared, the prescaler is disabled, and the RTC runs at the 32.768 kHz crystal frequency.

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. In order for the RTC to operate at the proper

Description of Operation

rate, software must set the prescaler enable bit after initial powerup. When this bit is set, the prescaler is enabled, and the RTC runs at a frequency of 1 Hz.

Write `RTC_PREN` and then wait for the write complete event before programming the other registers. It is safe to write `RTC_PREN` to 1 every time the processor boots. The first time sets the bit, and subsequent writes have no effect, as no state is changed.

 Do not disable the prescaler by clearing the bit in `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 through `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 `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 `RTXI` cycles after a 0-to-1 transition of `RTC_PREN`.

Use the following sequence to achieve synchronization to within 100 μ s.

1. Write `RTC_PREN` to 0.
2. Wait for the write to complete.
3. Wait for the external event.
4. Write `RTC_PREN` to 1.
5. Wait for the write to complete.
6. Reprogram the time into `RTC_STAT`.

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 through the system MMR status registers at any time.

The primary RTC functionality, shown in [Figure 14-1 on page 14-4](#), consists of registers and counters that are powered by an independent RTC V_{DD} supply. This logic is never reset; it comes up in an unknown state when RTC V_{DD} is first powered on.

The RTC also contains logic powered by the same internal V_{DD} 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_{DD} -powered registers. This logic is reset by the same system reset and clocked by the same SCLK as the other peripherals.

[Figure 14-2](#) shows the connections between the RTC V_{DD} -powered RTC MMRs and their corresponding internal V_{DD} -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).

RTC Programming Model

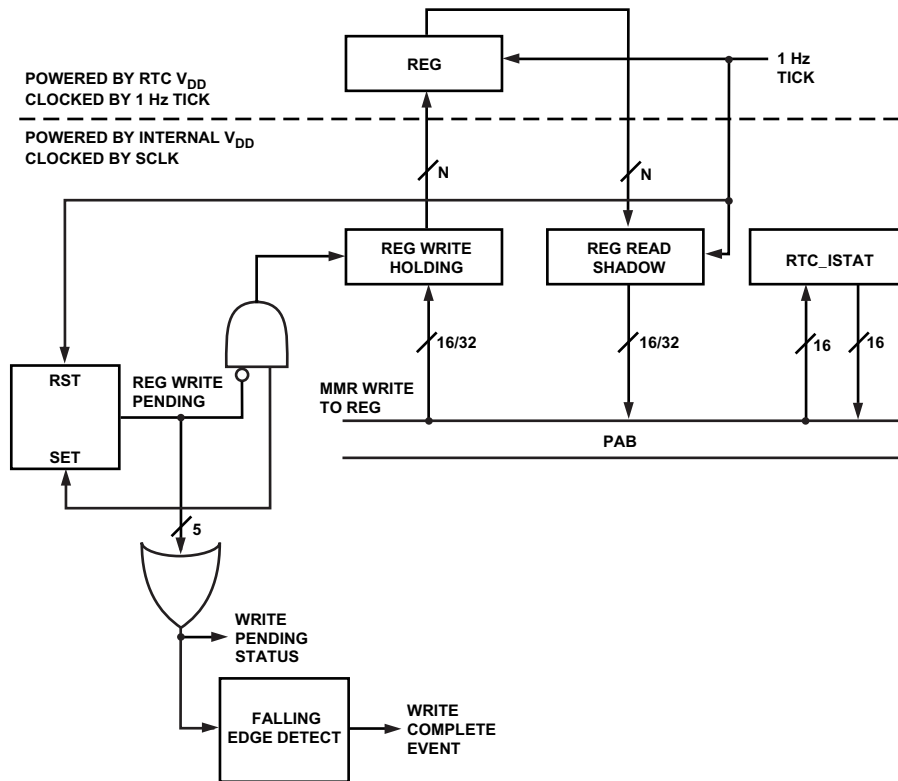


Figure 14-2. RTC Register Architecture

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_IOCTL` to cause an

interrupt. Software does not have to wait for writes to an RTC MMR to 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 are 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 is written before the write complete flag is set returns 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 RTC 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 incrementer. 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 μ s after a 1 Hz tick completes on the next 1 Hz tick, but the simplest, most predictable and recommended

RTC Programming Model

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 V_{DD} 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 V_{DD} counters continue to increment. During deep sleep, the internal V_{DD} 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 V_{DD} RTC control logic generates a virtual 1 Hz tick within one `RTXI` period (30.52 μ s) 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 write-1-to-clear the W1C bits in `RTC_ISTAT`. All W1C bits are already cleared by hardware. The seconds event flag is set when the RTC internal V_{DD} 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.

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.

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 and write pending status

Always valid.

- Minutes event flag

Valid only after the second field in `RTC_STAT` is valid. Use the write

RTC Programming Model

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.

- 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 and day alarm event flags

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.

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-1-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.

Setting Time of Day

The RTC status register (`RTC_STAT`) 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.

RTC Programming Model

3. Write `RTC_STAT` to current time, 13:10:59.
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.

Using the Stopwatch

The RTC stopwatch count (`RTC_SWCNT`) register contains the countdown value for the stopwatch. 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.

The stopwatch can be programmed to any value between 0 and $(2^{16} - 1)$ seconds, which 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_SWCNT` 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.

Interrupts

The RTC can provide interrupts at several programmable intervals:

- Per second, minute, hour, and day—based on increments to the respective counters in `RTC_STAT`
- On countdown from a programmable value—value in `RTC_SWCNT` transitions to 0 or is written with 0 by software (whether it was previously running or already stopped with a count of 0)
- Daily at a specific time—all fields of `RTC_ALARM` must match `RTC_STAT` except the day field
- On a specific day and time—all fields of `RTC_ALARM` register must match `RTC_STAT`

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`). The eight RTC interrupt events 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 wakeup 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 wakeup to the PLL are handled specially and are masked (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

RTC Programming Model

interrupts are needed immediately after reset, it is recommended to write `RTC_ICTL` to `0x0000` so that later read-modify-write accesses function as intended.

Interrupt status can be determined by reading the RTC interrupt status register (`RTC_ISTAT`). All bits in `RTC_ISTAT` 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.

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 14-3](#), 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 `RTC_ICTL` occurs. The assertion of the deep sleep wake-up signal causes the processor core clock (`CCLK`) and the system clock (`SCLK`) to restart. Any enabled event that asserts the RTC deep sleep wake-up signal also causes the RTC IRQ to assert once `SCLK` restarts.

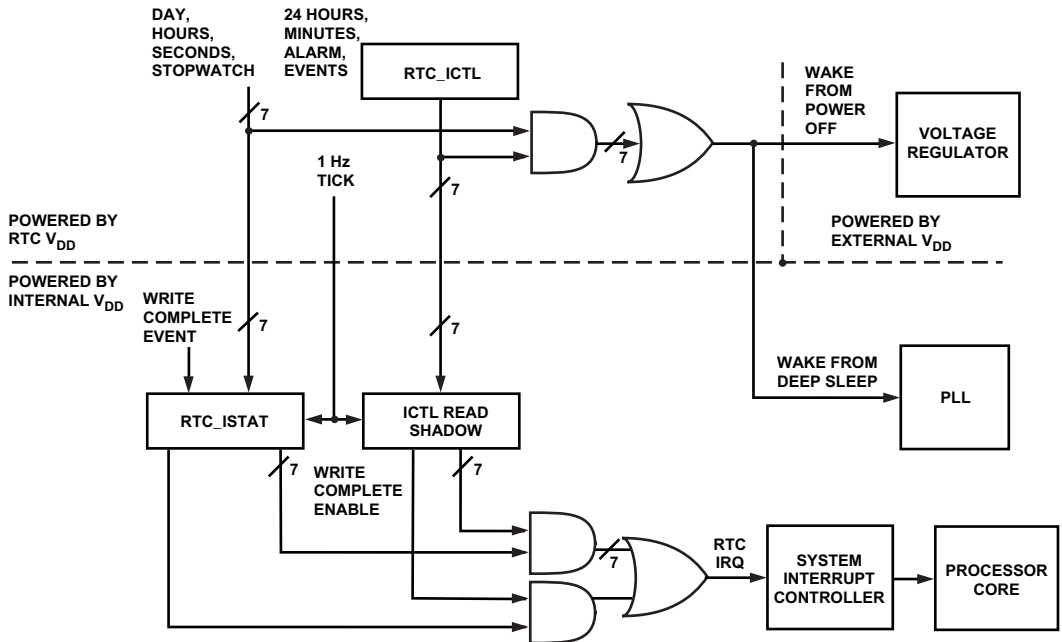


Figure 14-3. RTC Interrupt Structure

State Transitions Summary

Table 14-1 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 18, “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).

RTC Programming Model

Table 14-1. Effect of States on RTC MMRs

RTC V _{DD}	IV _{DD}	System State	RTC_ICTL	RTC_ISTAT	RTC_STAT RTC_SWCNT	RTC_ALARM RTC_PREN
Off	Off	No power	X	X	X	X
On	On	Reset	As written	0	Counting	As written
On	On	Full on	As written	Events	Counting	As written
On	On	Sleep	As written	Events	Counting	As written
On	On	Active	As written	Events	Counting	As written
On	On	Deep sleep	As written	0	Counting	As written
On	Off	Off	As written	X	Counting	As written

Table 14-2 summarizes software’s responsibilities with respect to the RTC at various system state transition events.

Table 14-2. RTC System State Transition Events

At This Event:	Execute This Sequence:
Power on from no power	Write RTC_PREN = 1. Wait for write complete. Write RTC_STAT to current time. Write RTC_ALARM, if needed. Write RTC_SWCNT. Write RTC_ISTAT to clear any pending RTC events. Write RTC_ICTL to enable any desired RTC interrupts or to disable all RTC interrupts.
Full on after reset or Full on after power on from off	Wait for seconds event, or write RTC_PREN = 1 and wait for write complete. Write RTC_ISTAT to clear any pending RTC events. Write RTC_ICTL to enable any desired RTC interrupts or to disable all RTC interrupts. Read RTC MMRs as required.

Table 14-2. RTC System State Transition Events (Cont'd)

At This Event:	Execute This Sequence:
Wake from deep sleep	Wait for seconds event flag to set. Write RTC_ISTAT to acknowledge RTC deep sleep wakeup. Read RTC MMRs as required. The PLL state is now active. Transition to full on as needed.
Wake from sleep	If wakeup came from RTC, seconds event flag is set. In this case, write RTC_ISTAT to acknowledge RTC wakeup IRQ. Always, read RTC MMRs as required.
Before going to sleep	If wakeup by RTC is desired: Write RTC_ALARM and/or RTC_SWCNT as needed to schedule a wakeup event. Write RTC_ICTL to enable the desired RTC interrupt sources for wakeup. Wait for write complete. Enable RTC for wakeup in the system interrupt wakeup-enable register (SIC_IWR).
Before going to deep sleep	Write RTC_ALARM and/or RTC_SWCNT as needed to schedule a wakeup event. Write RTC_ICTL to enable the desired RTC event sources for deep sleep wakeup. Wait for write complete.
Before going to off	Write RTC_ALARM and/or RTC_SWCNT as needed to schedule a wakeup event. Write RTC_ICTL to enable any desired RTC event sources for powerup wakeup. Wait for write complete. Set the wake bit in the voltage regulator control register (VR_CTL).

RTC Registers

The following sections provide register definitions. Illustrations are shown in [Figure 14-4](#) through [Figure 14-9](#).

[Table 14-3](#) shows the functions of the RTC registers.

Table 14-3. RTC Register Mapping

Register Name	Function	For More Info	Notes
RTC_STAT	RTC status register	on page 14-21	Holds time of day
RTC_ICTL	RTC interrupt control register	on page 14-21	Bits 14:7 are reserved
RTC_ISTAT	RTC interrupt status register	on page 14-22	Bits 13:7 are reserved
RTC_SWCNT	RTC stopwatch count register	on page 14-22	Undefined at reset
RTC_ALARM	RTC alarm register	on page 14-23	Undefined at reset
RTC_PREN	Prescaler enable register	on page 14-23	Always set PREN = 1 for 1 Hz ticks

RTC Status (RTC_STAT) Register

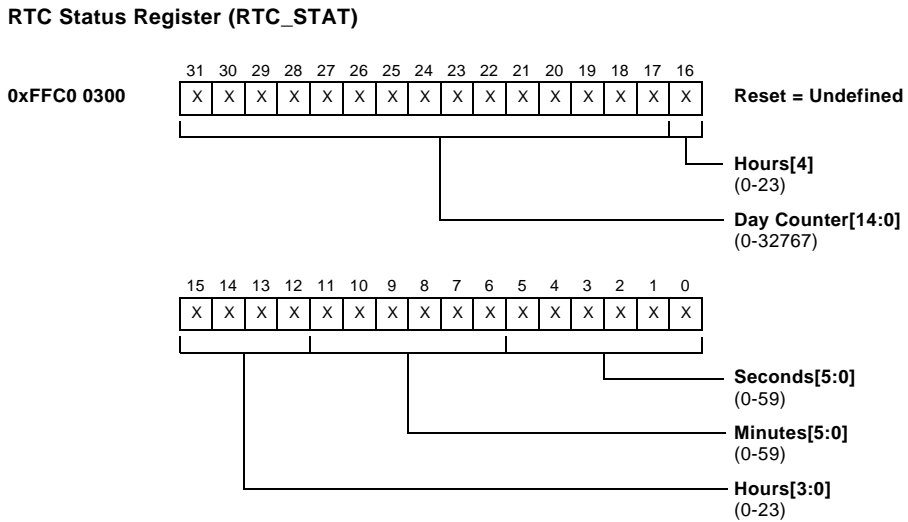


Figure 14-4. RTC Status (RTC_STAT) Register

RTC Interrupt Control (RTC_ICTL) Register

RTC Interrupt Control Register (RTC_ICTL)

0 - Interrupt disabled, 1 - Interrupt enabled

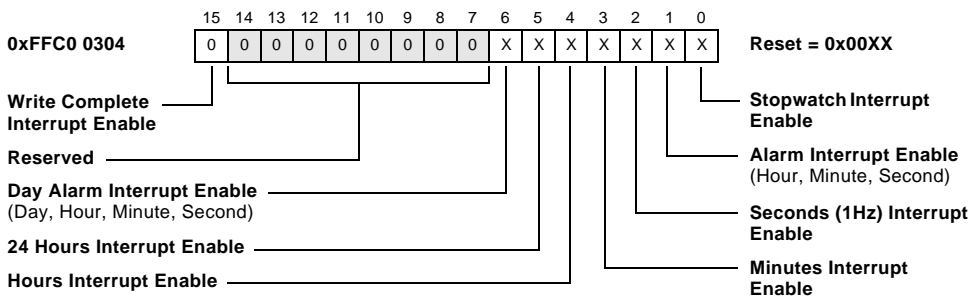


Figure 14-5. RTC Interrupt Control (RTC_ICTL) Register

RTC Registers

RTC Interrupt Status (RTC_ISTAT) Register

RTC Interrupt Status Register (RTC_ISTAT)

All bits are write-1-to-clear, except bit 14

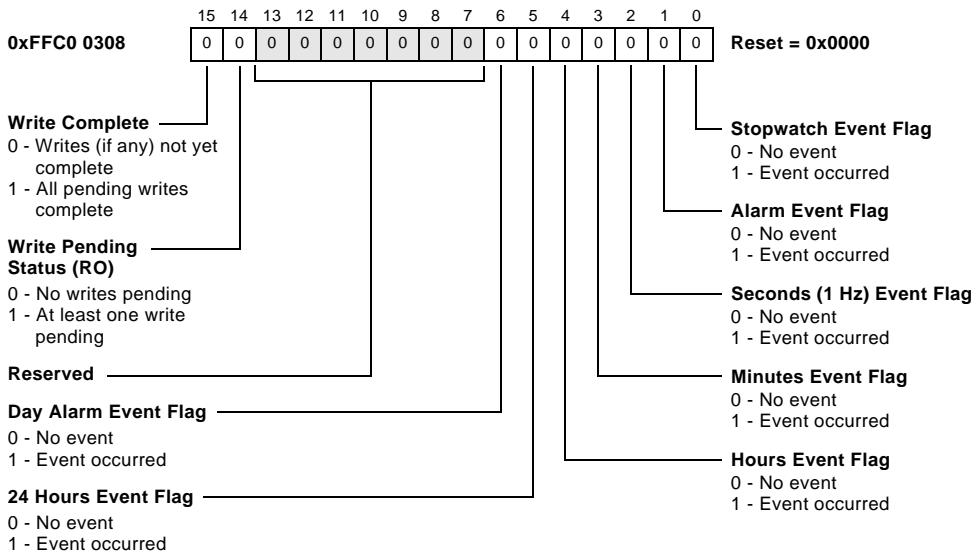


Figure 14-6. RTC Interrupt Status (RTC_ISTAT) Register

RTC Stopwatch Count (RTC_SWCNT) Register

RTC Stopwatch Count Register (RTC_SWCNT)

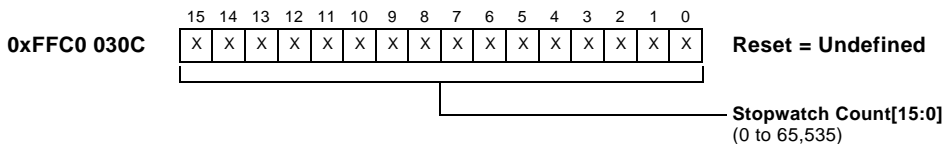


Figure 14-7. RTC Stopwatch Count (RTC_SWCNT) Register

RTC Alarm (RTC_ALARM) Register

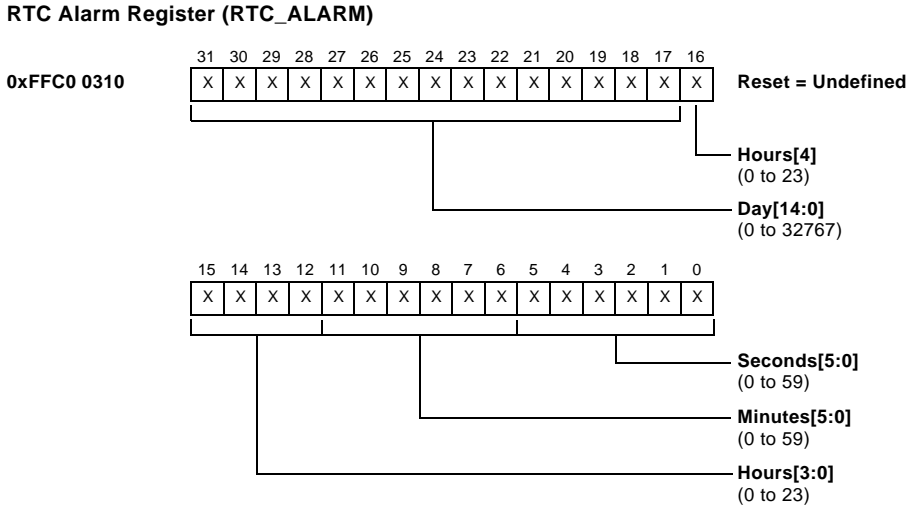


Figure 14-8. RTC Alarm (RTC_ALARM) Register

RTC Prescaler Enable (RTC_PREN) Register

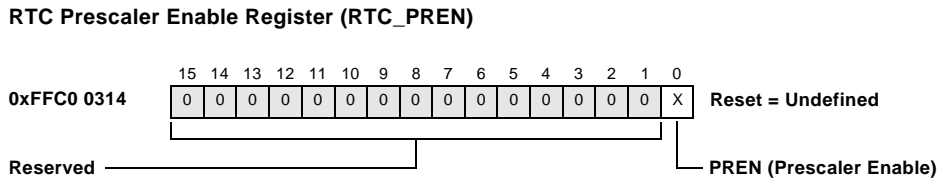


Figure 14-9. Prescaler Enable (RTC_PREN) Register

Programming Examples

The following RTC code examples show how to enable the RTC prescaler, how to set up a stopwatch event to take the RTC out of deep sleep mode, and how to use the RTC alarm to exit hibernate state. Each of these code examples assumes that the appropriate header file is included in the source code (that is, `#include <defBF54x.h>` for ADSP- BF54x projects).

Enable RTC Prescaler

[Listing 14-1](#) properly enables the prescaler and clears any pending interrupts.

Listing 14-1. Enabling the RTC Prescaler

```
RTC_Initialization:
    P0.H = HI(RTC_PREN);
    P0.L = LO(RTC_PREN);
    R0=PREN(Z); /* enable pre-scalar for 1 Hz ticks */
    W[P0] = R0.L;

    P0.L = LO(RTC_ISTAT);
    R0 = 0x807F(Z);
    W[P0] = R0.L; /* clear any pending interrupts */

    R0 = WRITE_COMPLETE(Z); /* mask for WRITE-COMPLETE bit */
Poll_WC:    R1 = W[P0](Z);
            R1 = R1 & R0; /* wait for Write Complete */
            CC = AZ;
            IF CC JUMP Poll_WC;

RTS;
```

RTC Stopwatch For Exiting Deep Sleep Mode

[Listing 14-2](#) sets up the RTC to utilize the stopwatch feature to come out of deep sleep mode. This code assumes that the `_RTC_Interrupt` is properly registered as the ISR for the real-time clock event, the RTC interrupt is enabled in both `IMASK` and `SIC_IMASK`, and that the RTC prescaler has already been enabled properly.

Listing 14-2. RTC Stopwatch Interrupt to Exit Deep Sleep

```

/* RTC Wake-Up Interrupt To Be Used With Deep Sleep Code */
_RTC_Interrupt:
    PO.H = HI(PLL_CTL);
    PO.L = LO(PLL_CTL);
    RO = W[P0](Z);
    BITCLR (RO, BITPOS(BYPASS));
    W[P0] = RO; /* BYPASS Set By Default, Must Clear It */

    IDLE; /* Must go to IDLE for PLL changes to be effected */

    RO = 0x807F(Z);
    PO.H = HI(RTC_ISTAT);
    PO.L = LO(RTC_ISTAT);
    W[P0] = R7; /* clear pending RTC IRQs */

    RO = WRITE_COMPLETE(Z); /* mask for WRITE-COMPLETE bit */
Poll_WC_IRQ:    R1 = W[P0](Z);
                R1 = R1 & RO; /* wait for Write Complete */
                CC = AZ;
                IF CC JUMP Poll_WC_IRQ;

    RTI;

Deep_Sleep_Code:
    PO.H = HI(RTC_SWCNT);

```

Programming Examples

```
P0.L = LO(RTC_SWCNT);
R1 = 0x0010(Z); /* set stop-watch to 16 seconds */
W[P0] = R1.L; /* will produce ~15 second delay */

P0.L = LO(RTC_ICTL);
R1 = STOPWATCH(Z);
W[P0] = R1.L; /* enable Stop-Watch interrupt */
P0.L = LO(RTC_ISTAT);
R1 = 0x807F(Z);
W[P0] = R1.L; /* clear any pending RTC interrupts */

R0 = WRITE_COMPLETE(Z); /* mask for WRITE-COMPLETE bit */
Poll_WC1: R1 = W[P0](Z);
          R1 = R1 & R0; /* wait for Write Complete */
          CC = AZ;
          IF CC JUMP Poll_WC1;

/* RTC now running with correct stop-watch count and interrupts
*/
P0.H = HI(PLL_CTL);
P0.L = LO(PLL_CTL);
R0 = W[P0](Z);
BITSET (R0, BITPOS(PDWN)); /* set PDWN To Go To Deep Sleep */
W[P0] = R0.L; /* Issue Command for Deep Sleep */

CLI R0; /* Perform PLL Programming Sequence */
IDLE;
STI R0; /* In Deep Sleep When Idle Exits */

RTS;
```

RTC Alarm to Come Out of Hibernate State

[Listing 14-3](#) sets up the RTC to utilize the alarm feature to come out of hibernate state. This code assumes that the prescaler has already been properly enabled.

Listing 14-3. Setting RTC Alarm to Exit Hibernate State

```

Hibernate_Code:
    PO.H = HI(RTC_ALARM);
    PO.L = LO(RTC_ALARM);
    RO = 0x0010(Z); /* set alarm to 16 seconds from now */
    W[PO] = RO.L;

    PO.L = LO(RTC_STAT);
    RO = 0; /* Clear RTC Status to Start Counting at 0 */
    W[PO] = RO.L;

    PO.L = LO(RTC_ICTL);
    RO = ALARM(Z);
    W[PO] = RO.L; /* enable Alarm interrupt */

    PO.L = LO(RTC_ISTAT);
    RO = 0x807F(Z);
    W[PO] = RO.L; /* clear any pending RTC interrupts */

    RO = WRITE_COMPLETE(Z);
Poll_WC1:  R1 = W[PO](Z);
           R1 = R1 & RO; /* wait for Write Complete */
           CC = AZ;
           IF CC JUMP Poll_WC1;

/* RTC now running with correct RTC status */
GoToHibernate:

```

Programming Examples

```
PO.H = HI(VR_CTL);
PO.L = LO(VR_CTL);
RO = W[P0](Z);
BITCLR(RO, 0); /* Clear FREQ (bits 0 and 1) to */
BITCLR(RO, 1); /* go to Hibernate State      */
BITSET(RO, BITPOS(WAKE)); /* Enable RTC Wakeup */
W[P0] = RO.L;

CLI RO; /* Use PLL programming sequence to */
IDLE;   /* make VR_CTL changes take effect */
RTS;    /* Should Never Execute This */
```

15 SECURITY

This chapter describes security features of the ADSP-BF54x Blackfin processor and how they can be used to facilitate a secure system.

This chapter includes the following sections:


- [“Overview” on page 15-2](#)
- [“Description of Operation” on page 15-6](#)
- [“Programming Model” on page 15-27](#)
- [“Security Registers” on page 15-51](#)

This chapter describes the security features and functionality of the ADSP-BF54x Blackfin processor. Following an overview and a list of key features are a description of operation and functional modes of operation.

The intention of the chapter is to describe security features of the ADSP-BF54x Blackfin processor and how they can be used to facilitate a secure system. It is beyond the scope of this chapter to fully describe various ways to implement secure systems or to describe security protocols and primitives in any great detail.

Overview

Lockbox™ Secure Technology for Analog Devices Blackfin processors is comprised of a mix of hardware and software mechanisms designed to prevent unauthorized accesses and allow trusted code to execute on the processor. Throughout the rest of this chapter, the terms Blackfin Lockbox™ secure technology and Lockbox will be used interchangeably.

 The developer's decision to use security features is completely optional. No security features are enabled by default. The developer can choose to never implement security features in their application if it is so desired. The Blackfin will always power up/boot in Open Mode with no security features or restrictions enabled.

Blackfin Lockbox™ secure technology allows users to:

- Safeguard as little as a single function, as much as a complete system, or anything in-between.
- Uniquely identify each processor by a Unique Chip ID.
- Utilize secure key storage provided by non-volatile, write-protectable One Time Programmable (OTP) memory.
- Perform digital signature authentication using elliptic curve cryptography (ECC) and secure one-way hash (SHA-1) algorithms implemented in firmware.
- Keep secret information in secure OTP Memory.
- Use any encryption algorithm to protect code or other assets.
- Ensure data integrity through digital signature authentication.
- Safeguard confidentiality via encryption of any or all of the system -from core IP (code security) to data integrity.

These features in combination provide the following benefits.

- **Authenticity/Origin verification** - Lockbox secure technology allows for verification of a code image against its associated digital signature, and provides for a process to identify entities and data origins.
- **Integrity** - Developers can use a digital signature authentication process to ensure that the message or the content of the storage media has not been altered in any way. If either the message or its digital signature was altered, Lockbox will fail during the authentication process.
- **Confidentiality** - Cryptographic encryption/decryption supports situations that require the ability to prevent unauthorized users from seeing and using designated files and streams. Lockbox's secure processing environment (Secure Mode) and secure memory support methods for ensuring confidentiality.
- **Renewability**

Renewability refers to the updating of system components to enhance security.

Lockbox's Unique Chip ID enables end users to identify each Blackfin processor and hence each OEM device in which the processor resides.

This Lockbox feature can be used in support of revocation and renewability of licenses in case of security violations in digital rights management systems, for example:

Overview

Unique Chip ID, in combination with a trusted DRM agent (sourced by the OEM), enables developers to implement renewability in DRM systems.

Unique Chip ID provides capability to identify each OEM device and “blacklist” devices to remove them from a system.

- Prevention of Mass Copying - Lockbox supports cryptographic encryption/decryption algorithms for situations when confidentiality is required. The Unique Chip ID can also be utilized to “bind” the processor to one specific boot source/device and can be used to facilitate antitheft schemes and prevent OEM device cloning.

The ADSP-BF54x Blackfin processors featuring Lockbox™ secure technology provide security features that enable developer’s applications to use secure protocols consisting of code authentication and execution of code within a secure environment. Together these features protect secure memory spaces and restrict control of security features to authenticated developer code.

Features

Lockbox is comprised of a combination of hardware and software elements. These elements are:

- **OTP Memory.** An array of non-volatile write-protectable memory that can be programmed by the developer only one time. Half of the array is public (accessible in any mode) and the other half is private (only accessible in secure mode). For more information on OTP memory, refer to Chapter 16 One-Time-Programmable (OTP) Memory of the ADSP-BF54x Blackfin Processor Hardware Reference manual.
- **Secured System Switches.** Programmable bitfields in the Secured System Switches MMR to disable and enable different methods of memory access in support of a secured environment. Some of these protection mechanisms include disabling DMA access to L1 and L2 memory and disabling ADI JTAG instructions from the ICE port.
- **Secure Mode Control.** This involves the Secure State Machine hardware required to support a transition from an unsecured state of operation (Open Mode), through an authentication state (Secure Entry Mode) and finally to a secured state (Secure Mode) where secrets are accessible.
- **Firmware.** Code that resides in on-chip L1 instruction ROM and performs digital signature authentication. Having the code that performs the digital signature authentication in ROM ensures integrity of the code.

Description of Operation

- **User callable cryptographic ciphers.** In addition to the control code that resides in the on-chip L1 instruction ROM used for authentication, there exists a number of cryptographic functions (SHA-1, AES and ARC4) that are callable. The APIs are documented in the Programming Model section of this chapter.
- **Unique Chip ID.** Each ADSP-BF54x Blackfin processor will have a 128-bit Unique Chip ID value stored in public OTP memory which will be unique. The Unique Chip ID will always be programmed and write protected before a processor leaves the Analog Devices factory and it will always be located at the same OTP page address.



The 128-bit Unique Chip ID value can be read but cannot be modified by the developer or end user. A total of 64K-bits of OTP memory is available to the developer if additional user-defined ID values are desired. These IDs can be stored in either public or private areas of OTP memory depending on application requirements. Please refer to Chapter 16 One-Time-Programmable (OTP) Memory for more details.

Description of Operation

Blackfin Lockbox technology is based upon the concept of authentication of digital signatures using standards-based algorithms and provides a secure processing environment in which to execute code and access protected assets.

Digital signatures are created using a public-key signature algorithm, the Elliptic Curve Cryptography (ECC) public-key cipher, and a secure one-way hash algorithm, SHA-1. A public-key algorithm actually uses two different keys; the public key and the private key (called a key pair). The private key is known only to its owner and is not stored on-chip, while the public key can be available to anyone and is stored in the public OTP memory region on-chip. Public-key algorithms such as ECC are designed

so that if one key is used for encryption, the other is necessary for decryption. Furthermore, the encryption key cannot be reasonably calculated from the decryption key. In a digital signature authentication scheme like Lockbox, the private key is used to generate the signature and the corresponding public key is used to validate it. Each ADSP-BF54x Blackfin processor has an on-chip ROM that contains firmware with the Elliptic Curve Cryptography (ECC) and SHA-1 algorithms. These are called to verify the digital signatures (ECDSA¹).

JTAG emulation and test features are disabled in hardware and certain memory access restrictions are enabled during verification of the digital signature. Once the signature is authenticated, the access restrictions are still in effect and can only be controlled by the authenticated user code.

Secure State Machine

The ADSP-BF54x processor includes a Secure State Machine to handle the different protection configurations of the processor depending on the security situation. The machine states are “Open Mode”, “Secure Entry Mode” and “Secure Mode” (See [Figure 15-1](#)). The following sections describe these machine states.

The state of the Secure State Machine can be identified by reading SECURE_STATUS[1:0] bits. The bit values in the upper right of the states shown in [Figure 15-1](#) correspond to the bit values in SECURE_STATUS[1:0].

For more information on SECURE_STATUS MMR, see the section on “[Security Registers](#)” on [page 15-51](#) of this chapter.

¹ ECDSA implementation on ADSP-BF54x Blackfin products only supports the Koblitz curve.

Description of Operation

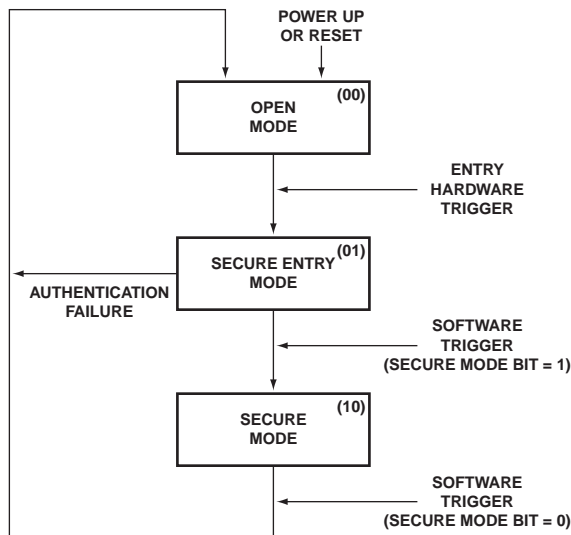


Figure 15-1. Secure State Machine Modes

Open Mode

Default operating state of the processor in which no restrictions are present except restricted access to Private OTP memory area. The processor will power-up and boot in Open Mode. This is the default state upon power up and after processor reset. No Lockbox security features or protection mechanisms are enabled in this state.

State flow illustrated in [Figure 15-1](#) shows that the Secure State Machine can only transition from Open Mode into Secure Entry Mode and there is no direct path from Open Mode into Secure Mode.

Secure Entry Mode

The on-chip ROM firmware performs the authentication process in this operating state. This mode is entered when NMI is active and the Program Counter (PC) is vectored to the first address of the authentication firmware in the on-chip ROM. The PC is monitored to ensure that it remains within the address range allocated to the Authentication firmware code. If the PC vectors outside of the address range of the authorization code, authentication will fail and the state returns to Open Mode. Any errors caught by firmware or hardware monitor will result in authentication failure and an abortion of the authentication process with the firmware exiting Secure Entry Mode and transitioning back to Open Mode. If authentication is successful, the firmware will initiate the transition from Secure Entry Mode to Secure Mode.

In Secure Entry Mode, no DMA access is allowed to certain regions of internal SRAM, and JTAG emulation is disabled. It is recommended that the user disable cache prior to initiating authentication. Interrupts are disabled by firmware prior to entry into Secure Mode. Interrupts are either re-enabled by dropping the interrupt level from NMI via the SESR arguments or by waiting until the authentication is successful and re-enabling them in the authenticated code after entry into Secure Mode. In addition, only the public area of OTP memory is accessible in this mode. (for more information on memory access restrictions within Secure Entry Mode, see [“Secure Entry Service Routine \(SESR\) API” on page 15-27](#)).

State flow illustrated in [Figure 15-1](#) shows that the Secure State Machine can only transition from Secure Entry Mode into Secure Mode upon successful digital signature authentication. A transition from Secure Entry Mode back into Open Mode can occur if digital signature authentication fails or if the authentication process is aborted due to an error observed by the firmware. Such errors include illegal memory boundary conditions or jumps outside of the firmware range (for example, servicing an interrupt).


Description of Operation

Secure Mode

Secure operating state in which trusted, authenticated code is allowed unrestricted access to the processor resources, execution of authenticated code occurs, decryption of sensitive information, etc. This is the only mode that allows access (reads and writes) to the private OTP memory space where secure data such as secret keys can be stored. Hence, the private area of OTP memory can be used to store confidential, secret information that only authorized authenticated code can only access. Therefore, this is the only operating state in which users can securely run their own Blackfin implementation of any cryptographic cipher in which secret keys are used.

Only the code (or message) digitally signed by a trusted source and successfully passes through Lockbox's authentication process can gain access to Secure Mode.

State flow illustrated in [Figure 15-1](#) shows that the Secure State Machine can only transition from Secure Mode back into Open Mode and there is no direct path from Secure Mode into Secure Entry Mode. Exit from Secure Mode is implemented through software control by writing a "0" value to the SECURE0 bit within the SECURE_CONTROL register.

 Assertion of reset or power cycling will also return the processor to the default Open Mode regardless of the state of operation when the reset or power cycle event occurred. See also special handling of hardware reset in the Reset Handling in Secure Mode section.

Access to private OTP memory is restricted in Open Mode and Secure Entry Mode regardless of whether or not other security features are enabled or disabled.

SecureMode Control

Figure 15-2 describes the inputs that control the secure state machine flow.

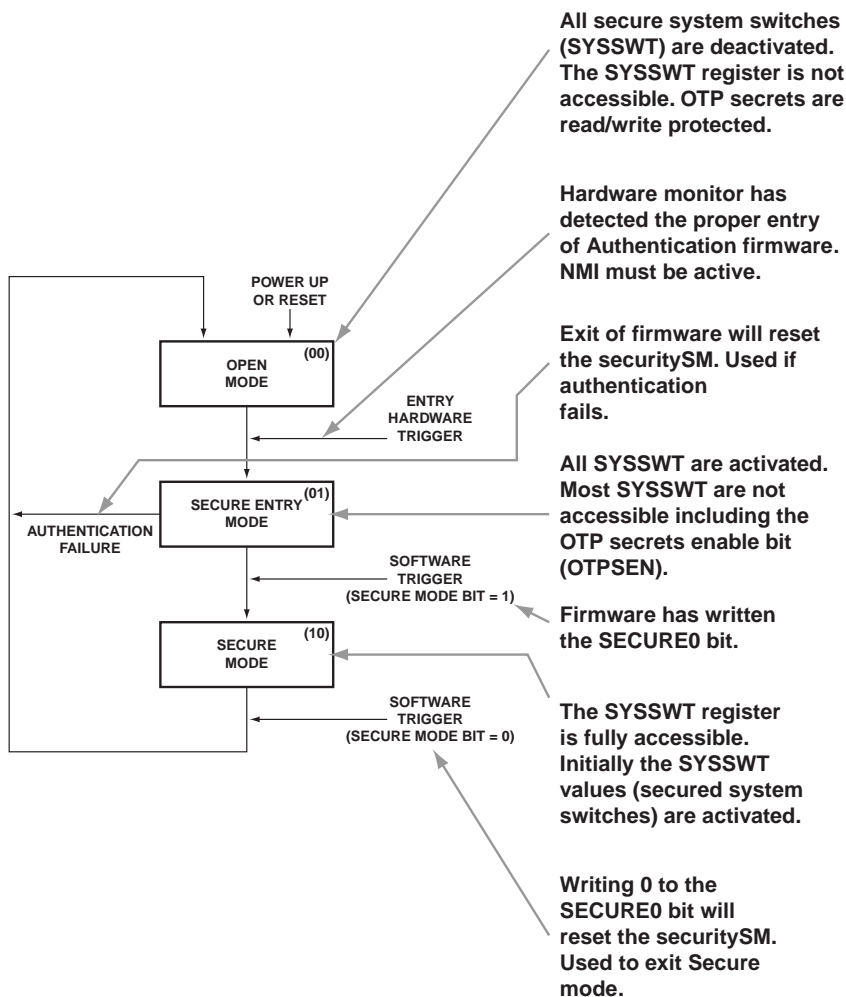


Figure 15-2. Secure Mode Control

Hardware supports transition from an Open Mode of operation, through a Secure Entry Mode and finally to a Secured Mode where secrets are accessible.

Description of Operation

Open Mode is characterized by being the default mode of processor upon power up/reset/boot, holding all secured system switches deactivated and protecting the private OTP memory area from access. The processor is open with all features being available with no restrictions (except for the private area of OTP memory).

Secure Entry Mode is characterized by executing firmware out of internal ROM memory to authenticate information loaded into on-chip memory. All secured system switches are activated. However, private OTP Memory is not accessible yet.

Secure Mode is entered only after a successful digital signature authentication process from Secure Entry Mode. It provides access to the private OTP memory area and makes secured system switches accessible to user (authenticated) code. This is the mode of operation in which to perform sensitive decryption or execution of trusted, authenticated code.

Authentication can only be requested and initiated while the processor is operating in Open Mode. If authentication is requested while the processor is operating in Secure Mode, the Secure State Machine will not transition into Secure Entry Mode. Instead, the Secure State Machine will remain in Secure Mode.



Please note that Open Mode, Secure Entry Mode and Secure Mode are states which pertain to the Secure State Machine. User Mode and Supervisor Mode are modes of operation which pertain to the core. The use of the term “mode” should not be confused and are not necessarily mutually exclusive. In Open Mode, the processor can operate in either User or Supervisor Mode. Since the firmware is entered when the NMI is being handled, Secure Entry Mode must start in Supervisor Mode. Finally, authenticated code executing in Secure Mode must be either operating at NMI interrupt level or the interrupt level that triggered the NMI.

Functional Description

The following sections provide a functional description of the Security features.

Protection relies on the on-chip ROM code that includes Elliptic Curve Cryptography (ECC) and SHA-1 algorithms, applied towards verification of code authenticity using a digital signature. A processor has emulation and test features disabled in hardware as well as certain memory access restrictions upon entry into Secure Entry Mode (where authentication is performed) and maintained into Secure Mode. These functions can only be controlled by authenticated user application software executing in Secure Mode.

User code must request authentication by complying with two criteria; (1) asserting a Non-Maskable Interrupt (NMI) and (2) vector the Program Counter (PC) to the first executable address in the Secure Entry Service Routine (SESR) in firmware which resides in L1 Instruction ROM.

During the authentication process JTAG emulation will be disabled, memory protection restrictions will be enabled and interrupts will be masked. The user has the option to pass arguments to the security firmware to control certain functionality during the authentication process. Please refer to the Secure Entry Service Routine API section for more information.

Digital Signature Authentication

Digital signatures are created off-chip (typically on a host computer) using the ECC algorithm and SHA-1, both of which are available in the public domain. In digital signature authentication, the private key generates the signature (off-chip) and the corresponding public key validates it (on-chip). The private key is known only to its owner and is not stored on-chip, while the public key can be available to anyone and is stored on-chip in OTP memory.

Description of Operation

Lockbox uses standards-based cryptographic algorithms for digital signature authentication. ECDSA¹ is implemented in the Blackfin BF54x processors. Digital signature validation on ADSP-BF54x utilizes Elliptic Curve Cryptography² (ECC) based on a binary field size of 163 bits and SHA-1³ secure one-way hash (which produces a 160-bit message digest).

In order to generate public/private key pairs or prepare digital signatures and apply them to application code, developers can use any method that complies with the Elliptic Curve Digital Signature Algorithm (ECDSA) specified in FIPS 186-2 with Change Notice 1 dated October 5, 2001, Digital Signature Standard (DSS). ECDSA is described in ANSI X9.62-1998. The Lockbox implementation in BF54x processors supports the following Koblitz curve, which is recommended in FIPS 186-2 for US Federal Government use:

1. m : 163 (degree of binary field)
2. a : 1
3. b : 1 (a and b are the constants in the elliptic curve equation: $y^2 + xy = x^3 + ax + b$)
4. X_g : 2FE13C0537BBC11ACAA07D793DE4E6D5E5C94EEE8
5. Y_g : 289070FB05D38FF58321F2E800536D538CCDAA3D9 (X_g and Y_g define the base point G)
6. r : 4000000000000000000020108A2E0CC0D99F8A5EF (r is the order of the base point G)

¹ ECDSA implementation on these Blackfin products only supports the Koblitz curve.

² These implementations are based on the Elliptic Curve Digital Signature Algorithm (ECDSA) specified in FIPS 186-2 with Change Notice 1 dated October 5, 2001, Digital Signature Standard (DSS) (<http://csrc.nist.gov/cryptval/dss.htm>), and specified in ANSI X9.62-1998.

³ SHA-1 is based on the publicly available standard for FIPS 180-2 (Secure Hash Signature Standard [SHS]) (FIPS PUB 180-2), <http://csrc.nist.gov/CryptoToolkit/tkhash.html>.

7. T: 4 (T is the normal basis type)
8. $p(t): t^{163} + t^7 + t^6 + t^3 + 1$ ($p(t)$ is the field polynomial)


The following steps summarize the Digital Signature Authentication process. Steps 1 to 3 correspond to the off-chip creation of a digital signature of a file or message. Steps 4 to 6 correspond to the on-chip digital signature authentication. These steps are preceded by generation of a key pair (Private Key and Public Key) and the programming of the Public Key in the Public OTP Memory.

1. A one-way hash of the file (message to be authenticated) is produced using SHA-1 off-chip (for example, using a host PC).
2. The hash is encrypted through ECC off-chip with the private key, thereby signing the file and completing the generation of the digital signature.
3. The file and the signed hash are stored on an external device such as Flash memory or a host device.
4. Upon transfer to the Blackfin's internal memory, a one-way hash of the file is calculated on-chip through SHA-1 (residing in Blackfin on-chip ROM)
5. Using the ECC algorithm (residing in Blackfin on-chip ROM), the Blackfin decrypts the signed hash with the user's public key stored in the Blackfin's OTP memory.
6. The two hash results are then compared. If the signed hash matches the calculated hash, the signature is valid and the file is intact.

If the digital signature authentication process is successful, the Blackfin will transition from Secure Entry Mode to Secure Mode. At this time, all of the access restrictions mentioned will be in place. JTAG will be disabled and certain portions of on-chip SRAM memory are restricted from DMA

Description of Operation

access. The restrictions can be controlled once in Secure Mode by having the authenticated code modify the Secure System Switches (SECURE_SYSSWT) appropriate for use by the developer's application.

 Encryption/decryption is only necessary when an application requires *confidentiality*. It is not always necessary to work with encrypted code to ensure code security. Authentication alone can be used when confidentiality is not required when ensuring tamper-proof code image and/or non-repudiation in a system. Authentication thus safeguards code *integrity*.

Since the digital signature uniquely describes its corresponding code/message, the code/message itself does not have to be encrypted if *confidentiality* is not required. If the code/message is modified, either intentionally or inadvertently, authentication will fail since the *integrity* of the code message has been compromised.

Digital Signature Authentication Performance Measurement

Authentication can be performed at any point during processor operation in Open Mode. It can be performed immediately upon boot or it can be performed any time after boot.

The algorithms used in the Lockbox firmware are highly optimized Blackfin code running from L1 instruction ROM in the core clock domain. Firmware execution time for the digital signature authentication process is on the order of 5 million core clock cycles depending upon the size of the digitally signed application code. This must be considered when architecting an application in order to allow a sufficient window of time in which authentication can proceed without requiring servicing of interrupts in the system.

The time it takes for authentication is dependent on several factors. These include the size of the message to be authenticated. This affects the amount of calculations done in the secure hash function (SHA-1). It also affects the DMA time required to move the message out of L1 data memory and place it into L1 code memory.

Protection Features

In order to establish a secure processing environment and protect the security of applications that establish trust and reach the privileged mode of operation, Lockbox implements access restrictions. These restrictions include disabling JTAG emulation and disabling DMA access to portions of on-chip SRAM memory. The memory access restrictions implemented in hardware on the Blackfin are not applied to off-chip memory. Therefore, external memory is always considered insecure and caching external memory while operating in Secure Mode represents a security risk.

Description of Operation

Protection features include the following:

- Secure State Machine for implementing privileged states of operation in which access restrictions may be imposed to protect code and data.
- Protection of L1 and on-chip L2 regions of memory with DMA access controlled when in Secure Mode.

Disable DMA access to L1 and on-chip L2 memory

These restrictions to memory areas are configurable (See SECURE_SYSSWT)

- Disable ADI JTAG emulation from ICE port
- Divert hardware reset to NMI during Secure Mode operation to prevent “reset attack”.
- Provide software control over hardware protection features accessible to trusted code operating in Secure Mode.
- OTP memory for storage of customer programmable cipher keys, unique chip ID or a customer ID
- OTP write protection to protect programmed OTP memory locations from future tampering
- Private/Secret OTP memory region accessible only in Secure Mode
Store private key(s) for decryption of data or other validation
- A privileged mode (including firmware execution out of on-chip ROM) to perform code authentication

Protection mechanisms enabled within each of the States of the Secure State Machine along with the Secure System Switch register (SECURE_SYSSWT) providing control over the protection feature are summarized [Table 15-1](#).

Table 15-1. Secure State Machine

Secure State Machine	SECURE_SYSSWT	Description	Protected Memory Range
Open Mode (0x00000000)	The switches are involuntarily set with all controls OFF (unrestricted access).	No protection mechanisms or restrictions enabled.	No restrictions ¹
Secure Entry (0x000704D9)	EMUDABL	Emulation Disable	Emulation disabled
	L1IDABLE	L1 Instruction Memory Disable 0xFFA00000 - 0xFFA07FFF SRAM	32 KB
	L1DADABL	L1 Data Bank A Memory Disable 0xFF800000 - 0xFF807FFF SRAM and SRAM/Cache	32 KB
	L1DBDABL	L1 Data Bank B Memory Disable 0xFF900000 - 0xFF901FFF SRAM	8 KB
	L2DABL	l2 Memory Disable	64 KB

Description of Operation

Table 15-1. Secure State Machine (Cont'd)

Secure State Machine	SECURE_SYSSWT	Description	Protected Memory Range
Secure Mode (0x000704D9)	EMUDABL	Emulation Disable	User Configurable
	RSTDABL	RESET Disable	User Configurable
	L1IDABLE	L1 Instruction Memory Disable 0xFFA00000 - 0xFFA07FFF SRAM	0-32 KB
	L1DADABL	L1 Data Bank A Memory Disable 0xFF800000 - 0xFF807FFF SRAM and SRAM/Cache	0-32 KB
	L1DBDABL	L1 Data Bank B Memory Disable 0xFF900000 - 0xFF901FFF SRAM	0-32 KB
	L2DABL	I2 Memory Disable	0-64 KB

- 1 Private OTP is only accessible when operating in Secure Mode with OTPSEN bit set in SECURE_SYSSWT register

On-chip SRAM memory protection takes the form of DMA access restrictions only. There is no need to protect the on-chip SRAM from processor core access due to the fact that while operating in Secure Mode, the developer's authenticated code has full control over the processor core and execution of all software instructions so there is no need to protect against core instructions. It is the responsibility of the developer to take steps to avoid surrendering control of the Program Sequencer and the core to untrusted code execution.

Operating in Secure Mode

Entering Secure Mode

Upon successful digital signature authentication, the Secure State Machine transitions into Secure Mode. The same default protection features enabled in Secure Entry Mode are carried forward into Secure Mode. This includes JTAG emulation being disabled, DMA access restrictions to memory and interrupts being masked. It is the responsibility of the authenticated code to manipulate or remove these restrictions if so desired.

Exiting Secure Mode

Secure Mode provides a secure operating environment to execute sensitive code, run cryptographic ciphers and process sensitive data. Upon exiting Secure Mode, the authenticated code should remove any sensitive code and data from memory because this sensitive information will still be accessible in Open Mode if it is not removed prior to exiting Secure Mode. Exit from Secure Mode is implemented through software control by writing a “0” value to the SECURE0 bit within the SECURE_CONTROL register. Please refer to the Security Registers section and Clearing Private Data section of this chapter for more information.

Reset Handling in Secure Mode

Hardware Reset

Hardware reset is diverted to NMI when operating in Secure Mode only. When operating outside of Secure Mode, hardware reset behaves normally. This protection feature is configurable via the RSTDABL bit within the SECURE_SYSSWT register when operating within Secure Mode.

Description of Operation

This is a protection feature to prevent malicious entities from attempting to assert hardware reset while sensitive code or data is present in the processor's on-chip SRAM or in the processor's registers. A "reset attack" could take the following form: If hardware reset were left unprotected and reset was asserted while sensitive information were present on-chip, the processor would return to the default state of Open Mode with no protection features enabled and a malicious entity could gain access to the on-chip memory and registers, for example via JTAG emulation. In such a scenario assets intended to be protected could be compromised.


By diverting hardware reset to NMI while the processor operates in Secure Mode, servicing of hardware reset can be controlled and delayed in order to first implement a memory clean-up routine in software to purge sensitive information from internal memory and registers prior to servicing reset. At the completion of the memory clean-up, the processor can then be reset via software command and safely returned to Open Mode with no sensitive information available to be compromised.

By default, the SESR loads the address of a memory clean-up routine stored in the on-chip L1 instruction ROM into the NMI EVT2 prior to transitioning from Secure Entry Mode into Secure Mode. See Clearing Private Data section of this chapter for more information.

Clearing Private Data

As part of the SESR firmware, there is a small routine stored in the on-chip L1 instruction ROM that clears the internal memory, generates a RESET event and puts the processor into idle. It is recommended that the user sets this routine as the new EVT2 NMI vector once the user's authenticated application code is executing. This will prevent a malicious user from trying to reset the processor while it is operating in Secure Mode and

then view the contents of internal memory when the processor returns to Open Mode after servicing RESET. The “Clear Private Data” routine is located at address 0xffa147e8.

 It is recommended that user software running in Secure Mode should also perform RAM clean-up prior to clearing the SECURE0 Secure Mode bit and exiting Secure Mode via normal code execution within user’s secure function. If sensitive code/data remains in on-chip RAM after exiting Secure Mode without wiping memory and register contents or cycling power to the processor, it will be visible and accessible in Open Mode.

This memory wipe routine in the ROM executes a watchdog RESET to reset the processor at the completion of the memory wipe. The code also performs a wipe of the OTP_DATA0-3 registers which are used to hold data from OTP access reads (i.e. which could contain secret key or other sensitive data left by user code execution).

If a custom memory cleanup routine is part of an authenticated message, the user can use that routine instead of the one provided with the Lockbox firmware. The user just has to update EVT2 in the event vector table to point to the start of the custom memory cleanup routine.

Due to the fact that hardware reset is configured by default to be redirected to NMI when the processor is operating in Secure Mode, it is recommended that the user implements a watchdog reset within the EVT2 NMI ISR in order to reset the processor. A Watchdog reset is implemented by writing a value 2'b00 in WDOG_CTL[2:1] and this will cause a complete core reset. The watchdog reset will not be redirected to the NMI pin as in the case of the external hardware reset and it will properly reset the processor. For more details of watchdog reset, please refer to Software Resets section of the System Reset and Booting chapter.

This “reset attack” protection scheme need only protect against hardware RESET which can be applied externally as the system developer typically has no control over this in an embedded system. While operating in

Description of Operation

Secure Mode, the developer's authenticated code has full control over the processor core and execution of all software instructions so there is no need to protect against soft reset instructions. It is not recommended that the user's secure application code implement a soft reset without first deleting sensitive information from memory and registers.

Public Key Requirements

A valid ECC public key must be a non-zero value and meet the following criteria:

Given the public key value shown here:

```
369368AF243193D001E39CE76BB1D5DA08A9BC0A6  
15F7A90C841D4F1E1B005E70F167F6EF7CD2E251B
```

format in 32-bit little endian as follows:

```
8A9BC0A6  
BB1D5DA0  
1E39CE76  
43193D00  
69368AF2  
00000003  
CD2E251B  
167F6EF7  
B005E70F  
41D4F1E1  
5F7A90C8  
00000001
```

The values should be stored in OTP pages 0x10, 0x11, 0x12 as follows (where 'L' denotes lower half of page, 'H' denotes upper or high half of page):

```
page: 0x010L: 0xbb1d5da08a9bc0a6,  
page: 0x010H: 0x43193d001e39ce76,  
page: 0x011L: 0x0000000369368af2,  
page: 0x011H: 0x167f6ef7cd2e251b,  
page: 0x012L: 0x41d4f1e1b005e70f,  
page: 0x012H: 0x000000015f7a90c8,
```

The general format takes the form of twelve (12) 32-bit words:

```
Word 1  
Word 2  
Word 3  
Word 4  
Word 5  
Word 6  
Word 7  
Word 8  
Word 9  
Word 10  
Word 11  
Word 12
```

Stored into OTP pages in the following order (where 'L' denotes lower half of page, 'H' denotes upper or high half of page):


```
page: 0x010L:Word 2Word 1  
page: 0x010H:Word 4Word 3  
page: 0x011L:Word 6Word 5
```

Description of Operation

page: 0x011H:Word 8Word 7
page: 0x012L:Word 10Word 9
page: 0x012H:Word 12Word 11

Storing public cipher key in public OTP


In order to make use of security features, the user must first store an ECC public key in the Blackfin's public region of OTP memory pages 0x10, 0x11 and 0x12 as specified in the Firmware's Secure Entry Service Routine (SESR) API and the OTP memory map (see [“Secure Entry Service Routine \(SESR\) API” on page 15-27](#)). If no ECC public key is stored in this area of OTP, digital signature authentication cannot be successfully completed and no Lockbox security features can be enabled. For more information on programming the OTP memory, please refer to the OTP memory chapter.

 If security features which rely upon the ECC public key are not going to be used, it is recommended that customers write-protect the ECC public key OTP memory space in order to prevent malicious entities from writing a value into this memory and potentially exploiting this feature without the developer's consent.

Cryptographic Ciphers

Lockbox uses SHA-1 and ECC to implement ECDSA as part of the authentication process to enter into Secure Mode. These ciphers reside in the firmware in the on-chip L1 instruction ROM. In addition to these ciphers, the Advanced Encryption Standard (AES) and ARC4 are also available in the ROM. The SHA-1, AES and ARC4 ciphers are user-callable in Open Mode or in Secure Mode. The APIs are documented in the

Programming Model section of this chapter. Note that ECC is not user-callable and is only executed as part of firmware during the authentication process.

 Since AES uses symmetric keys that need to be private, and these private keys typically require confidentiality, it is recommended that this cipher be executed in Secure Mode to access the keys from the private area of OTP memory.

Keys

Although Lockbox uses an ECC public key for digital signature authentication, and has private OTP memory to store private keys for other cryptographic algorithms, Lockbox does not implement key management. Lockbox does not implement key generation nor does it implement key exchanges natively in the Blackfin hardware.

In order to use Lockbox, an ECDSA key pair must be generated. The private key is used off-chip (typically on a host PC) to sign the message and the public key is placed in the public OTP memory where it is used to authenticate the signed message. Lockbox is only part of a full cryptosystem. It is the responsibility of the user to develop the other parts of the cryptosystem necessary for the intended application.

Programming Model

Secure Entry Service Routine (SESR) API

This section describes the procedure to use Lockbox to authenticate a message. Memory configuration, input arguments and return codes are also described here.

Programming Model

In this chapter, the term “message” was widely used to describe the entity being digitally signed off-chip, and later authenticated on-chip by the SESR security firmware. “Message”, “secure function” (SF), and “secure application” are used interchangeably in this section and mean the same thing.

Starting Authentication

For an application to establish trust and reach the privileged mode of operation (for example, enter Secure Mode), the Secure State Machine has to transition from Open Mode, through Secure Entry Mode, to Secure Mode. In order to transition from Open Mode to Secure Entry Mode, NMI must be asserted and the program counter (PC) must vector to the beginning address of the firmware (SESR) at location 0xffa14000.

This can be achieved by loading the beginning address of the SESR (0xffa14000) as the NMI handler in the event vector table (EVT2). Then in supervisor mode, issue a “raise 2;” instruction. Similarly, NMI hardware pin may be asserted instead of issuing a software raise instruction. Once the PC vectors to the SESR, while NMI assertion is sensed by the hardware, the Secure State Machine will transition into Secure Entry Mode.

Before actually going into Secure Entry Mode, the user will have to set up the memory environment. This includes specifying the arguments (described in this section) and moving the message to be authenticated into L1 data memory.

Memory Configuration

Figure 15-3 illustrates the Secure Entry Mode default memory configuration upon initiating authentication and entering the SESR.

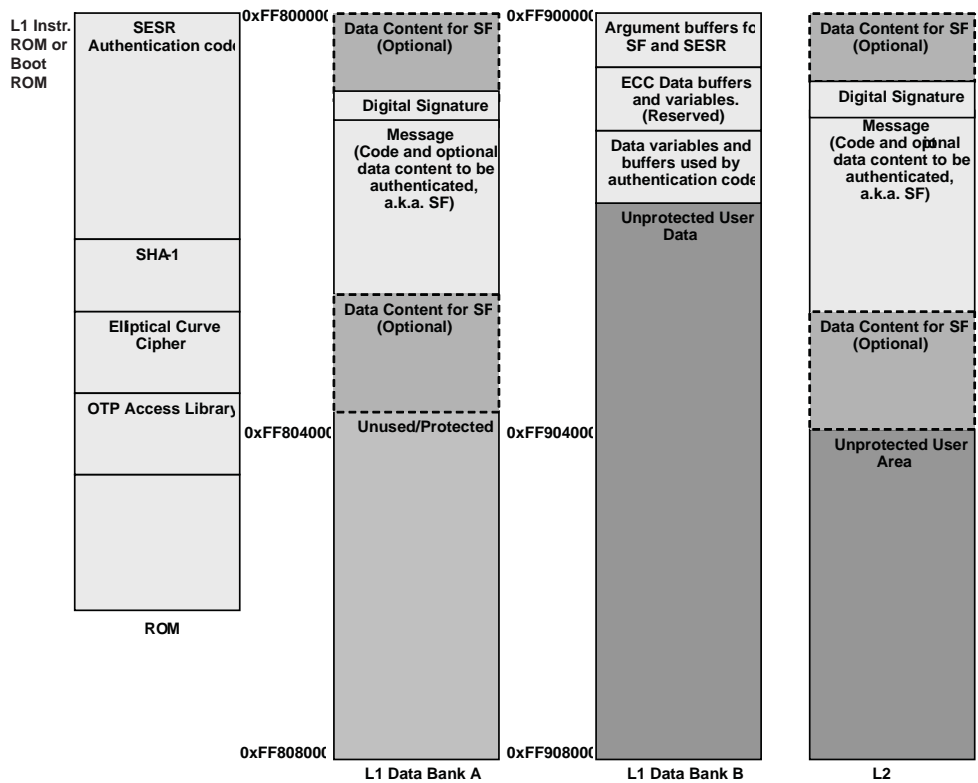


Figure 15-3. Memory Configuration for Authentication

Programming Model

Message Placement

The message can be placed in either L1A or in L2 for authentication. If the message (for example, code) is put into L1A for authentication, it must be DMA'd to either L1 code space or L2, where it can execute. If the message is placed into L2 for processing, it has the option of staying where it is and can be executed directly from L2. It is the user's responsibility to provide the message in L1A or L2 memory for the SESR. If authentication is successful, the SESR will then move the message via DMA to the final destination according to the SESR arguments. No further action is required by the developer to perform this DMA as it is executed by the firmware.

Digital Signature

The digital signature is a pair of 163 bit integers. Each integer is padded to the nearest 32-bit word, resulting in 192 bits for each integer resulting in a total size of 384 bits. The authentication firmware always expects the digital signature to be followed by the message. For example, if the message is placed in L1A data memory, and the digital signature starts at address 0xff800000, the message must immediately follow the digital signature and be located at address 0xff800030. The same holds true if the message and digital signature are placed in L2 memory as they must be stored together contiguously in memory with the message always immediately following the digital signature.

Message Size Constraints

The maximum size of any message to be authenticated is limited by the size of on-chip memory in the Blackfin. When the Secure State Machine enters into Secure Entry Mode (authentication), certain portions of on-chip SRAM memory are protected from DMA accesses. These protected memory regions include L1A (32KB) and L1B data memory (8KB each), L1 code memory (32KB) and half of L2 memory (64KB). This

means that the maximum allowable message/code size that can be authenticated is 32KB less 48 bytes for the digital signature, if placed in L1A data memory and 64KB less 48 bytes if placed in L2 memory.

Memory Usage

In data bank B of the L1 memory, the arguments for both the SESR and the secure function are stored beginning at address 0xff900000. In addition, a portion of the L1B data memory is reserved for the firmware for scratch space. All memory above address 0xff901f00 is reserved for authentication. The user can either allocate this area of memory solely for Lockbox or save any data elsewhere in memory prior to starting authentication.



Any user information residing in the scratch space reserved area of L1 Data Bank B will be overwritten during the authentication process.

Memory Protection

This Secure Entry Mode default memory configuration with both protected and unprotected regions of on-chip SRAM is implemented in order to allow developers to initiate digital signature authentication at any time during Open Mode processor operation. If an application is already running on the processor, the unprotected memory regions can be used for placement of data buffers. When authentication occurs, access to these data buffers will not be restricted and the application can thus be given higher precedence over the authentication process if necessary.

The Secure Entry Mode default memory protection configuration put into place upon initiating authentication cannot be modified by the developer. This is to ensure integrity of the secure processing environment during the authentication process and help prevent malicious tampering.

Secure Function and Secure Entry Service Routine Arguments

Prior to initiating the authentication, the arguments for both the SESR and the message (also known as Secure Function) must be set up. The arguments are stored in argument buffers stored in L1B data memory. Specifically, the arguments for the Secure Function are stored at the top of L1B data memory, at address 0xff900000. There are 24 bytes allocated for the arguments for the secure function. Following the argument buffer for the Secure Function is the argument buffer for the SESR, at address 0xff900018. For security reasons this authentication protocol accesses fixed locations for arguments. When the user starts executing the SF, it receives 2 arguments. The first argument (R0) contains the address of the SF argument buffer. The second argument (R1) holds the IMASK value before shut off interrupts

Secure Function Arguments

When the message is successfully authenticated, the Program Counter will vector to the Secure Function with the 1st argument (R0) containing a pointer to top of L1B data memory. The 2nd argument that the secure function will have (R1), is the IMASK value. This value is obtained when the SESR successfully authenticates the message. Before the message is transferred via DMA to its final target run location, interrupts are shut off so tampering cannot occur between the time of successful authentication and execution of the secure function. The prototype for the secure function will be

```
void secure_function(tSecureFunctionArgs *, unsigned short  
imask);
```

The 24-byte Secure Function argument buffer is for the convenience of user to be able to pass arguments to the Secure Function prior to starting authentication.

It will be the Secure Function's responsibility to re-enable interrupts by using the saved IMASK value or by using a new IMASK value.

The 24-byte Secure Function argument buffer can be used in any aligned fashion. For example, it can be used to store six 32-bit words or twelve 16-bit words, or any combination of data types such as integers, shorts and characters, as long as the accesses are aligned.

Secure Entry Service Routine Arguments

The argument buffer for the SESR is shown in [Listing 15-1](#)

Listing 15-1. Argument Buffer for SESR

```
/* SESR argument structure. Expected to reside at address
0xFF900018*/
typedef struct SESR_args {

    unsigned short usFlags; /* security firmware flags*/
    unsigned short usIRQMask; /* interrupt mask*/
    unsigned long ulMessageSize; /* message length in bytes*/
    unsigned long ulSFEntryPoint; /* entry point of secure function*/
    unsigned long ulMessagePtr; /* pointer to the buffer containing
the digital signature and message */
    unsigned long ulReserved1; /* reserved*/
    unsigned long ulReserved2; /* reserved*/
} tSESR_args;
```

usFlags

The first argument, usFlags, is a 16bit bit flag that signals authentication what to do. [Figure 15-4](#) shows the meaning of the bits.

Programming Model

Bit 0 tells the authentication firmware whether or not to drop the interrupt level. To execute “raise 2;”, the Blackfin processor must be operating in supervisor mode, in other words, operating at one of the interrupt levels. NMI must be asserted when authentication is initiated. The caller/user has the option to deassert NMI and drop back down to a lower interrupt level (the interrupt level in effect when NMI was asserted to initiate authentication) or continue authentication at NMI level.

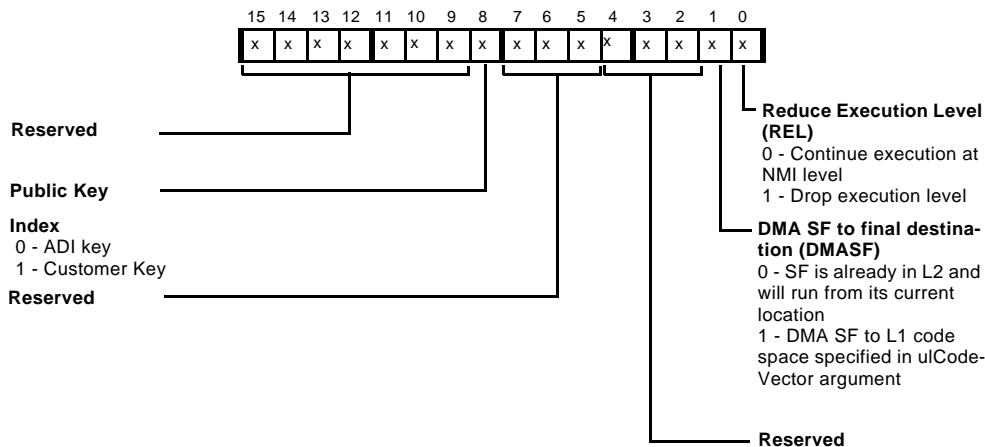


Figure 15-4. Bit Fields for Flags Argument

By lowering the interrupt level at which the authentication firmware executes, other interrupts can be serviced. Please be aware that if another interrupt is serviced and the PC vectors out of the authentication firmware during authentication, the authentication process will fail and return with an error code.

Bit 1 in the flags argument tells the authentication firmware whether or not to move the message/code to a final code space where it will be executed. This is only valid on certain ADSP-BF54x processors with L2 memory. Processors that have no L2 memory must move the message/code from L1A data memory to L1 code memory. The moves are done via memory DMA and are executed by the firmware.

Bit 8 tells the firmware which public key is used for authentication. The OTP memory holds two public keys. One is programmed by Analog Devices for failure analysis purposes only and the other is programmed by the developer.

usIRQMask

The usIRQMask argument is a 16-bit user-defined bitmask to be loaded into the lower 16 bits of the IMASK MMR if the execution level is to be lowered from NMI level. This argument allows the user to specify which, if any, interrupts will be allowed to be serviced should they occur during the time authentication occurs. Note that if any interrupt is serviced, the authentication process will fail and return with an error code as mentioned above. For more information regarding IMASK, please refer to the Blackfin Programming Reference manual.

ulMessageSize

The ulMessageSize argument is a 32-bit non-negative integer that tells the SESR how big the message is, in bytes. The ulMessageSize must be a multiple of 2, otherwise the SESR will return with an error.

ulSFEntryPoint

The ulSFEntryPoint argument is the final address that the message will be moved to and executed from. Again, since the authentication firmware expects code as the first portion of the message, the address must be a multiple of 4 since instructions can be either 16 bit or 32 bit lengths. If the

Programming Model

message consists of both code and data, it is the user's responsibility to move the data to the proper area of data memory for subsequent use within their application.

ulMessagePtr

The ulMessagePtr argument holds the address of where the digital signature and message is found.

Secure Message Execution

If the authentication of the digital signature is successful, the authentication firmware will directly vector the Program Counter to the Secure Function at its final target location, plus an offset of 4 bytes. The offset provides a location for the overlay ID if overlays are used with Lockbox. To return to the calling function, the authenticated message must execute "rtn;" if execution level was not signaled to be lowered in the authentication firmware. Otherwise, if the execution level was lowered, the Secure Function can return via "rts;".

To prevent tampering, interrupts and the watchdog timer are shut off near the end of successful authentication. It is the user's responsibility to re-enable the interrupts and the watchdog timer in the secure function if they are required in the user's application.

Return Codes

If for any reason, an error occurs, the SESR will return with an error code and bit 7 in the SECURE_STAT MMR will be set to indicate that register R0 will contain a valid error code. [Table 15-2](#) lists a portion of the valid return codes.

In addition to the return codes listed in [Table 15-2](#), a return value between -62 to -252 is also a valid error return code. These errors are from OTP accesses.

Table 15-2. List of Return Codes from SESR

Return Codes	Value	Description
SECFW_SUCCESS	0	Success
SECFW_ERROR_INV_FLAGS	-1	“Flags” argument to firmware is invalid.
SECFW_ERROR_INV_INTMASK	-2	IRQ mask specified is invalid.
SECFW_ERROR_INV_CODESZ	-3	Code size specified is either non-positive or odd.
SECFW_ERROR_OOB_CODE	-6	The message (Secure function) is too big and surpasses the protected region in L1A.
SECFW_ERROR_BAD_EVT	-10	One of the ISR specified in the Event Vector table points inside the authentication firmware.
SECFW_ERROR_PUBKEY_ZERO	-11	Invalid public key of (0,0).
SECFW_ERROR_AUTH_FAILED	-12	Invalid message/signature pair.
SECFW_ERROR_DMA	-15	MDMA error occurred during DMA transfer or the message to the final target vector.
SECFW_ERROR_DROPPING_INT_FAILED	-17	Could not drop interrupt level from NMI.
SECFW_ERROR_FUSE_READ_FAILED	-18	Error occurred while reading OTP memory.
SECFW_ERROR_TGTVECT_NONALIGNED	-19	Target vector is not 4 Byte aligned.
SECFW_ERROR_SECURE0_WRITE_FAILED	-20	Write to Secure0 bit failed. Secure State Machine might be blocking the write because ISR was taken.
SECFW_ERROR_SM_NOT_ENTERED	-21	Secure0 bit was written three times but secure mode was still not entered.

Programming Model

Table 15-2. List of Return Codes from SESR (Cont'd)

Return Codes	Value	Description
SECFW_ERROR_BAD_TGT_ADDR	-22	Target vector must be in L1 code space or L2 (for BF54x).
SECFW_ERROR_SF_TOO_BIG	-23	Message (Secure function) too big to fit at target location.

To decipher the error from an OTP access, there is an offset that must be added to the error code. The macro `OTP_READ_ERROR_OFFSET` (defined in `VDSP++` header files with a value of `-285`) is first added to the return value. The result is a bit mask. [Figure 15-5](#) shows the definition of the bit fields.

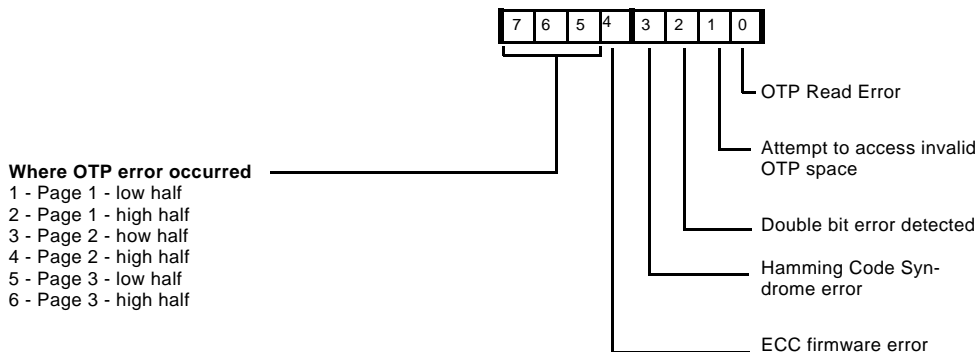


Figure 15-5. Bit Field Definition Return Value if OTP Error Occurred

Advanced Encryption Standard (AES) API

The ADSP-BF54x family of processors include a software implementation of the Advanced Encryption Standard (AES) in L1 ROM. This implementation of the AES symmetric-cipher is C-callable.

The following describes the application programming interface (API) for using AES including both data types and ROM routines.

ADI_AES_DATA Data Type

```
typedef struct ADI_AES_DATA
{
    u32  *pKeyExpTmp;
    u32  *pKR;
    u32  *pState;
    u32  *pIV;
    u32  *pRcon;
    u16  *pStateShiftExtract;
    u16  *pInvStateShiftExtract;
    u8   *pSBox;
    u8   *pSBoxMixC;
    u8   *pGFMPyTbl;
    u8   *pInvSBox;
    u8   *pInvSBoxMixC;
    u32  *pStatePointers;

} ADI_AES_DATA;
```

The AES initialization routine, `bfrom_AesInit()`, when provided with a reference to an object of type `ADI_AES_DATA`, will initialize some of the buffers specified in the object. The caller of `bfrom_AesInit()` has to allocate storage both for the object of type `ADI_AES_DATA` and for the buffers specified in that object.

Programming Model

Some of the buffers specified in ADI_AES_DATA are necessary only for encryption or only for decryption. Therefore, if only one of encryption or decryption is used, fewer buffers from ADI_AES_DATA need to be allocated.

Table 15-3 shows the buffers specified by ADI_AES_DATA, their sizes, and whether or not they are used in encryption and in decryption.

Table 15-3. Buffers Specified in ADI_AES_DATA

Buffer	Encryption	Decryption	Size (in bytes)
pKeyExpTmp	X	X	32
pKR	X	X	16
pState	X	X	16
pIV	X	X	16
pRcon	X	X	64
pStateShiftExtract	X	X	32
pInvStateShiftExtract	X	X	32
pSBox	X	X	256
pSBoxMixC	X		1024
pGFMPyTbl		X	1024
pInvSBox		X	256
pInvSBoxMixC		X	1024
pStatePointers	X	X	64

ADI_AES_KEYEXPANSION Data Type

```
typedef struct ADI_AES_KEYEXPANSION
{
    u8          *pCipherKey;
    u8          *pRoundKeys;
    u32         udKeySize;
    ADI_AES_DATA *pAesData;

} ADI_AES_KEYEXPANSION;
```

The AES key expansion routines, `bfrom_AesKeyexp()` and `bfrom_AesInvKeyexp()`, when provided with a reference to an object of type `ADI_AES_KEYEXPANSION`, will perform an AES key expansion on the `udKeySize`-long key stored in `pCipherKey`, and will store the resulting AES rounds keys in `pRoundKeys`. See [Table 15-4](#) for elements in an object of type.

Table 15-4. Elements in an Object of Type `ADI_AES_KEYEXPANSION`

<code>pCipherKey</code>	Pointer to the cipher key buffer, which is expected to hold the 128, 192, or 256-bit AES key.
<code>pRoundKeys</code>	Pointer to a buffer allocated by the caller of the key expansion routines. This buffer will hold the rounds keys generated by the key expansion routines.
<code>udKeySize</code>	The AES key size used (in multiple of 32-bit words). May take on the values 4, 6, and 8 to specify keys of size 128, 192, and 256-bits respectively.
<code>pAesData</code>	Pointer to an object of type <code>ADI_AES_DATA</code> , which is initialized through a call to <code>bfrom_AesInit()</code>

ADI_AES_CIPHER Data Type

```
typedef struct ADI_AES_CIPHER {
    u8      *pInputData;
    u8      *pOutputData;
```

Programming Model

```
u8      *pRoundKeys;  
u32     udDataLength;  
u8      *pInitVector;  
u32     udKeySize;  
u32     udMode;  
ADI_AES_DATA *pAesData;  
  
} ADI_AES_CIPHER;
```

The AES cipher routines, `bfrom_AesCipher()` and `bfrom_AesInvCipher()`, when provided with a reference to an object of type `ADI_AES_CIPHER`, will encrypt/decrypt the data in `pInputData` and will store the output in `pOutputData`. See [Table 15-5](#) for elements in an object of type.

Table 15-5. Elements in an Object of Type `ADI_AES_CIPHER`

<code>pInputData</code>	Pointer to the input data buffer. In the case of encryption, this buffer should contain plaintext. In the case of decryption, this buffer should contain ciphertext.
<code>pOutputData</code>	Pointer to the output data buffer. After encryption, this buffer will contain ciphertext. After decryption, this buffer will contain plaintext.
<code>pRoundKeys</code>	Pointer to a buffer containing the AES round keys, which are generated by the key expansion routines <code>bfrom_AesKeyexp()</code> and <code>bfrom_AesInvKeyexp()</code> .
<code>udDataLength</code>	The length of the input data in multiples of blocks of size 128-bits (16-bytes) each.
<code>pInitVector</code>	Certain block cipher modes of operation require an initialization vector. When an initialization vector is necessary, <code>pInitVector</code> points to the buffer containing the initialization vector.
<code>udKeySize</code>	The AES key size used (in multiples of 32-bit words). May take on the values 4, 6, and 8 to specify keys of size 128, 192, and 256-bits respectively.

Table 15-5. Elements in an Object of Type ADI_AES_CIPHER (Cont'd)

udMode	udMode specifies the block cipher mode of operation. The supported modes are: BLOCK_CIPHER_MODE_ECB for electronic codebook mode BLOCK_CIPHER_MODE_CBC for cipher block chaining mode BLOCK_CIPHER_MODE_OFB for output feedback mode BLOCK_CIPHER_MODE_CTR for counter mode
pAesData	Pointer to an object of type ADI_AES_DATA, which is initialized through a call to bfrom_AesInit()

bfrom_AesInit() ROM Routine

Entry address: 0xFFA14028

Arguments:

R0: Flags,

AES_ENCRYPTION
AES_DECRYPTION
AES_BOTH

R1: Pointer to an object of type ADI_AES_DATA

C prototype:

```
void bfrom_AesInit (u32 udFlags, ADI_AES_DATA *pAesData);
```

This function initializes the data buffers, which are referenced in ADI_AES_DATA and allocated by the caller, for use by the AES module.

This function is called first before other calls to the AES module.

Certain buffers are only necessary for encryption or decryption. Therefore, storage space may be saved by only allocating required buffers. The first argument specifies whether the user wishes to only encrypt, to only decrypt, or to both encrypt and decrypt. [Table 15-3](#) lists the buffers referenced by ADI_AES_DATA, their sizes, and whether or not they are necessary

Programming Model

for encryption and/or for decryption. If, for example, the user only needs to encrypt data (no decryption necessary), then the user can specify `AES_ENCRYPTION` as the first argument to `bfrom_AesInit()`, thus eliminating the need to allocate the buffers `pGFMPyTbl`, `pInvSBox`, and `pInvSBoxMixC`.

`bfrom_AesKeyexp()` ROM Routine

Entry address: `0xFFA1402C`

Arguments:

`RO`: Pointer to an object of type `ADI_AES_KEYEXPANSION`

C prototype:

```
s32 bfrom_AesKeyexp (ADI_AES_KEYEXPANSION *pAesKeyexpData);
```

Return Values:

```
AES_SUCCESS  
AES_INVALID_KEY_SIZE
```

This function produces the rounds keys for the forward cipher from the cipher key. It should be called before executing `bfrom_AesCipher()`.

`bfrom_AesKeyexp()` should be called everytime a new cipher key is used. If, for example, several data buffers need to be encrypted and all of them need to be encrypted using the same key then only one call to `bfrom_AesKeyexp()` is necessary. However, if each data buffer needs to be encrypted using a different key, then `bfrom_AesKeyexp()` should be called prior to calling `bfrom_AesCipher()` for each buffer encryption.

Notice that for the modes of operation `BLOCK_CIPHER_MODE_OFB` and `BLOCK_CIPHER_MODE_CTR`, encryption and decryption are identical. Therefore, in these modes `bfrom_AesKeyexp()` should be used instead of `bfrom_AesInvKeyexp()` to produce the round keys for the inverse cipher.

bfrom_AesInvKeyexp() ROM Routine

Entry address: 0xFFA14030

Arguments:

R0: Pointer to an object of type ADI_AES_KEYEXPANSION

C prototype:

```
s32 bfrom_AesInvKeyexp (ADI_AES_KEYEXPANSION *pAesKeyexpData);
```

Return Values:

```
AES_SUCCESS
AES_INVALID_KEY_SIZE
```

This function produces the rounds keys for the inverse cipher from the cipher key. It should be called before executing `bfrom_AesInvCipher()`.

`bfrom_AesInvKeyexp()` should be called everytime a new cipher key is used. If, for example, several data buffers need to be decrypted and all of them need to be decrypted using the same key then only one call to `bfrom_AesInvKeyexp()` is necessary. However, if each data buffer needs to be decrypted using a different key, then `bfrom_AesInvKeyexp()` should be called prior to calling `bfrom_AesInvCipher()` for each buffer decryption.

Notice that for the modes of operation `BLOCK_CIPHER_MODE_OFB` and `BLOCK_CIPHER_MODE_CTR`, encryption and decryption are identical. Therefore, in these modes `bfrom_AesKeyexp()` should be used instead of `bfrom_AesInvKeyexp()` to produce the round keys for the inverse cipher.

bfrom_AesCipher() ROM Routine

Entry address: 0xFFA14020

Arguments:

Programming Model

R0: Pointer to an object of type ADI_AES_CIPHER

C prototype:

```
s32 bfrom_AesCipher (ADI_AES_CIPHER *pAesCipherData);
```

Return Values:

```
AES_SUCCESS  
AES_INVALID_MODE
```

This function performs the AES forward cipher operation.

Notice that for the modes of operation BLOCK_CIPHER_MODE_OFB and BLOCK_CIPHER_MODE_CTR, encryption and decryption are identical. Therefore, in these modes `bfrom_AesCipher()` should be used instead of `bfrom_AesInvCipher()` to perform the AES inverse cipher operation.

`bfrom_AesInvCipher()` ROM Routine

Entry address: 0xFFA14024

Arguments:

R0: Pointer to an object of type ADI_AES_CIPHER

C prototype:

```
s32 bfrom_AesInvCipher (ADI_AES_CIPHER *pAesCipherData);
```

Return Values:

```
AES_SUCCESS  
AES_INVALID_MODE
```

This function performs the AES inverse cipher operation.

Notice that for the modes of operation `BLOCK_CIPHER_MODE_OFB` and `BLOCK_CIPHER_MODE_CTR`, encryption and decryption are identical. Therefore, in these modes `bfrom_AesCipher()` should be used instead of `bfrom_AesInvCipher()` to perform the AES inverse cipher operation.

SECURE HASH ALGORITHM (SHA-1) API

The ADSP-BF54x processor includes a software implementation of the Secure Hash Algorithm (SHA-1) in L1 ROM. This implementation of the SHA-1 hash algorithm is C-callable.

The following describes the application programming interface (API) for using SHA-1 including both data types and ROM routines.

ADI_SHA1 Data Type

```
typedef struct ADI_SHA1 {
    u8    *pInputMessage;
    u32   udMessageSize;
    u8    *pOutputDigest;
    u8    *pScratchBuffer;

} ADI_SHA1;
```

The SHA1 hash routine, `bfrom_Sha1Hash`, when provided with a reference to an object of type `ADI_SHA1`, will hash the `udMessageSize`-long message referenced by `pInputMessage`, and will store the hash value (also referred to as message digest) in the buffer referenced by `pOutputDigest`. See [Table 15-6](#) for elements in an object of type.

Table 15-6. Elements in an Object of Type `ADI_SHA1`

<code>pInputMessage</code>	Pointer to the input buffer.
<code>udMessageSize</code>	The size, in bytes, of the valid input data in <code>pInputMessage</code> .

Programming Model

Table 15-6. Elements in an Object of Type ADI_SHA1 (Cont'd)

pOutputDigest	Pointer to the output data buffer. After hashing, this buffer will contain the digest of the input message. The digest is 160-bits (SHA1_HASH_SIZE-bytes) long
pScratchBuffer	Pointer to a data buffer of size, SHA1_SCRATCH_BUFFER_SIZE-bytes, used by the SHA-1 module.

bfrom_Sha1Init ROM Routine

Entry address: 0xFFA14024

Arguments:

R0: Pointer to a buffer of size SHA1_SCRATCH_BUFFER_SIZE

C prototype:

```
void bfrom_Sha1Init (u8 *pScratchBuffer);
```

This function initializes some data elements in pScratchBuffer. It is called first before making any calls to bfrom_Sha1Hash.

bfrom_Sha1Hash ROM Routine

Entry address: 0xFFA14024

Arguments:

R0: Pointer to an object of type ADI_SHA1

C prototype:

```
void bfrom_Sha1Hash (ADI_SHA1 *pSha1);
```

This function performs the hash operation.

ARC4 API

The ADSP-BF54x processor includes a software implementation of the ARC4 algorithm in L1 ROM. This implementation of ARC4 is C-callable.

The following describes the application programming interface (API) for using ARC4 including both data types and ROM routines.

ADI_ARC4_KEY Data Type

```
typedef struct ADI_ARC4_KEY {
    u32  *pSBox;
    u32  *pKey;
    u32  udKeyLength;

} ADI_ARC4_KEY;
```

See [Table 15-7](#) for elements in an object of type.

Table 15-7. Elements in an Object of Type ADI_ARC4_KEY

pSBox	Pointer to an ARC4 substitution box.
pKey	A pointer to a buffer containing the ARC4 key.
udKeyLength	The size of the ARC4 key specified in pKey.

ADI_ARC4_DATA Data Type

```
typedef struct ADI_ARC4_DATA {
    u32  *pSBox;
    u32  *pData;
    u32  udDataLength;

} ADI_ARC4_DATA;
```

Programming Model

See [Table 15-8](#) for elements in an object of type.

Table 15-8. Elements in an Object of Type ADI_ARC4_DATA

pSBox	Pointer to an ARC4 substitution box, which has already been initialized through a call to <code>bfrom_Arc4Init()</code> .
pData	A pointer to a buffer containing the data to be encrypted or decrypted. Notice that the ARC4 module performs encryption and decryption in-place. Therefore, after calling the ARC4 cipher function, <code>bfrom_Arc4Cipher()</code> , this buffer will contain the encrypted or decrypted output.
udDataLength	The size of the data pointed to by <code>pData</code> .

bfrom_Arc4Init ROM Routine

Entry address: 0xFFA14018

Arguments:

R0: Pointer to an object of type ADI_ARC4_KEY

C prototype:

```
void bfrom_Arc4Init (ADI_ARC4_KEY *pArc4Key)
```

The ARC4 initialization routine, `bfrom_Arc4Init()` initializes the buffer pointed to by `pSBox` based on the key specified in `pKey` and `udKeyLength`.

`bfrom_Arc4Init()` should be called first before executing `bfrom_Arc4Cipher()`.

bfrom_Arc4Cipher ROM Routine

Entry address: 0xFFA1401C

Arguments:

R0: Pointer to an object of type ADI_ARC4_DATA

C prototype:

```
void bfrom_Arc4Cipher (ADI_ARC4_DATA *pArc4Data)
```

The ARC4 encryption/decryption routine, `bfrom_Arc4Cipher()`, encrypts/decrypts the data specified in `pData` and `udDataLength` using the substitution box specified in `pSBox`.

`bfrom_Arc4Cipher()` should be called after the substitution box has been initialized through a call to `bfrom_Arc4Init()`.

Security Registers

There are three registers which provide information that can be used during security mode control and to return status of the Secure State Machine states. These registers require privileged access depending on the operating state of the processor.

Table 15-9. Security Registers

Register	Description	Size (Bits)	Memory-Mapped Address
SECURE_SYSSWT	Secure System Switches	32	0xFFC04320
SECURE_CONTROL	Secure Control	16	0xFFC04324
SECURE_STATUS	Secure Status	16	0xFFC04328

Secured System Switches

Secured system switches control hardware that would otherwise allow a threat of attack to a secured system. Hardware is controlled voluntarily and involuntarily as follows:

- During Open Mode the switches are involuntarily set with all controls off (unrestricted access) with exception of access to OTP protected “secrets” area. OTP secrets are always protected and can only be accessible upon entry into Secure Mode.
- During Secure Entry Mode all switches are initially set with all controls on (restricted access). Two exceptions are the OTP secrets control (OTPSSEN bit) which is not accessible and access to the secrets OTP area remains restricted and the RSTDABL bit remains deactivated (External Reset is allowed).
- During Secured Mode operation all switches are voluntary (initially set) and under the control of authenticated code. Restricted access controls can therefore be reconfigured by authenticated user code. This includes the activation of Reset Disable (RSTDABL bit).

SECURE_SYSSWT (0xFFC04320)

The following MMR is the Secure System Switches. Limited write access to a few bits is allowed in Secure Entry Mode and full write access to all bits is allowed in Secured mode. No write access is allowed in Open Mode.

32-bit wide register. Requires 32-bit access.

SECURE_SYSSWT (0xFFC04320)

Secure System Switches. Limited write access to a few bits is allowed in Secure Entry and full write access to all bits is allowed in Secured mode. No write access is allowed in Open Mode.

32-bit wide register. Requires 32-bit access.

Table 15-10. SECURE_SYSSWT

Bit Position	Bit Name	Bit Description
		Reset = 0x0000 Secured Entry = 0x000704d9 Secured Mode = 0x000704db
0	EMUDABL	Emulation Disable. Upon Secured Entry EMUDABL's setting is based on the previous state of EMUOVR. Upon re-entering Open Mode, EMUDABL is cleared. This bit is always read accessible. This bit is write accessible only in Secured Mode. 0 - Analog Devices JTAG emulation instructions is recognized and executed. Once this bit is cleared while in Secured Mode it will not be set upon Secured Entry. This condition will remain until reset at which time it is cleared. This feature is used in security debug. 1 - Analog Devices JTAG emulation instructions are ignored. Standard emulation commands such as bypass is allowed.

Security Registers

Table 15-10. SECURE_SYSSWT

Bit Position	Bit Name	Bit Description
1	RSTDABL	<p>Reset Disable.</p> <p>This bit is not effected upon Secured Entry. This bit is set upon entering Secured Mode. Upon re-entering Open Mode, RSTDABL is cleared. This bit is always read accessible. This bit is write accessible only in Secured Mode.</p> <p>0 - External Resets are generated and serviced normally. 1 - External Resets are redirected to the NMI pin. This avoids circumventing memory clean operations.</p>
4:2	L1IDABL	<p>L1 Instruction Memory Disable.</p> <p>Upon Secured Entry L1IDABL is set to 0x6. Upon re-entering Unsecured Mode, L1IDABL is cleared. These bits are always read accessible. These bits are write accessible only in Secured Mode. In the event a DMA access is performed to a restricted memory area a DMA memory access error will occur resulting in a DMA_ERR interrupt and a clearing of DMA_RUN.</p> <p>000 - All DMA accesses are allowed to L1 Instruction areas. 001 - 1 Kbyte of memory (0xFFA00000 - 0xFFA003FF) has restricted non core access 010 - 2 Kbyte of memory (0xFFA00000 - 0xFFA007FF) has restricted non core access 011 - 4 Kbyte of memory (0xFFA00000 - 0xFFA00FFF) has restricted non core access 100 - 8 Kbyte of memory (0xFFA00000 - 0xFFA01FFF) has restricted non core access 101 - 16 Kbyte of memory (0xFFA00000 - 0xFFA03FFF) has restricted non core access 110 - 32 Kbyte of memory (0xFFA00000 - 0xFFA07FFF) has restricted DMA access. This is the initial setting upon entering Secured Entry. 111 - Reserved</p>

Table 15-10. SECURE_SYSSWT

Bit Position	Bit Name	Bit Description
7:5	L1DADABL	<p>L1 Data Bank A Memory Disable. Upon Secured Entry L1DADABL is set to 0x6. Upon re-entering Open Mode, L1DADABL is cleared. These bits are always read accessible. These bits are write accessible only in Secured Mode. In the event a DMA access is performed to a restricted memory area a DMA memory access error will occur resulting in a DMA_ERR interrupt and a clearing of DMA_RUN.</p> <p>000 - All DMA accesses are allowed to L1 data bank A areas.</p> <p>001 - 1 Kbyte of memory (0xFF800000 - 0xFF8003FF) has restricted non core access</p> <p>010 - 2 Kbyte of memory (0xFF800000 - 0xFF8007FF) has restricted non core access</p> <p>011 - 4 Kbyte of memory (0xFF800000 - 0xFF800FFF) has restricted non core access</p> <p>100 - 8 Kbyte of memory (0xFF800000 - 0xFF801FFF) has restricted non core access</p> <p>101 - 16 Kbyte of memory (0xFF800000 - 0xFF803FFF) has restricted non core access</p> <p>110 - 32 Kbyte of memory (0xFF800000 - 0xFF807FFF) has restricted DMA access. This is the initial setting upon entering Secured Entry.</p> <p>111 - Reserved</p>

Security Registers

Table 15-10. SECURE_SYSSWT

Bit Position	Bit Name	Bit Description
10:8	L1DBDABL	<p>L1 Data Bank B Memory Disable.</p> <p>Upon Secured Entry L1DBDABL is set to 0x4 giving L1 Data Bank B 8 Kbyte of non core restricted access. Upon re-entering Open Mode, L1DBDABL is cleared. These bits are always read accessible. These bits are write accessible only in Secured Mode. In the event a DMA access is performed to a restricted memory area a DMA memory access error will occur resulting in a DMA_ERR interrupt and a clearing of DMA_RUN.</p> <p>000 - All DMA accesses are allowed to L1 data bank B areas. This is the initial setting upon entering Secured Entry.</p> <p>001 - 1 Kbyte of memory (0xFF900000 - 0xFF9003FF) has restricted non core access</p> <p>010 - 2 Kbyte of memory (0xFF900000 - 0xFF9007FF) has restricted non core access</p> <p>011 - 4 Kbyte of memory (0xFF900000 - 0xFF900FFF) has restricted non core access</p> <p>100 - 8 Kbyte of memory (0xFF900000 - 0xFF901FFF) has restricted non core access. This is the initial setting upon entering Secured Entry.</p> <p>101 - 16 Kbyte of memory (0xFF900000 - 0xFF903FFF) has restricted non core access</p> <p>110 - 32 Kbyte of memory (0xFF900000 - 0xFF907FFF) has restricted DMA access.</p> <p>111 - Reserved</p>

Table 15-10. SECURE_SYSSWT

Bit Position	Bit Name	Bit Description
11	DMA0OVR	<p>DMA0 Memory Access Override</p> <p>Entering Secured Entry or Secured Mode does not effect this bit. Upon re-entering Open Mode, DMA0OVR is cleared. This bit is always read accessible. This bit is write accessible in both Secured Entry and Secured Mode.</p> <p>Controls DMA0 access to L1 Instruction, L1 Data and memory other than L1 regions. When clear access restrictions are based on Memory Disable settings within this register.</p> <p>0 - DMA0 accesses are restricted based on Memory Disable settings.</p> <p>1 - Unrestricted DMA0 accesses are allowed to all memory areas.</p>
12	DMA1OVR	<p>DMA1 Memory Access Override</p> <p>Entering Secured Entry or Secured Mode does not effect this bit. Upon re-entering Open Mode, DMA1OVR is cleared. This bit is always read accessible. This bit is write accessible in both Secured Entry and Secured Mode.</p> <p>Controls DMA1 access to L1 Instruction, L1 Data and memory other than L1 regions. When clear access restrictions are based on Memory Disable settings within this register.</p> <p>0 - DMA1 accesses are restricted based on Memory Disable settings.</p> <p>1 - Unrestricted DMA1 accesses are allowed to all memory areas.</p>
13	RESERVED	Reserved bit. This reserved bit always returns a “0” value on a read access. Writing this bit with any value has no effect.

Security Registers

Table 15-10. SECURE_SYSSWT

Bit Position	Bit Name	Bit Description
14	EMUOVR	<p>Emulation Override</p> <p>This bit is always read accessible. This bit may be written with a 1 in secured mode only.</p> <p>This bit can be cleared in any mode (Unsecured mode, Secured Entry and Secured mode). Controls the value of EMUDABL upon Secured Entry.</p> <p>0 - Upon Secured Entry the EMUDABL bit is set.</p> <p>1 - Upon Secured Entry the EMUBABL bit is cleared.</p> <p>This bit can only be set when EMUDABL (bit-0) is written with a “0” while this bit (bit-14) is simultaneously written with a 1.</p>

Table 15-10. SECURE_SYSSWT

Bit Position	Bit Name	Bit Description
15	OTPSEN	<p>OTP Secrets Enable.</p> <p>This bit can be read in all modes but is write accessible in Secured Mode only.</p> <p>0 - Read and Programming access of the “secured” OTP Fuse area is restricted. Accesses will result in an access error (FERROR)</p> <p>1 - Read and Programming access of the “secured” OTP Fuse area is allowed. If the corresponding program protection bit for an access is set, a program access is protected regardless of this bit’s setting.</p>
18:16	L2DABL	<p>L2 Disable.</p> <p>Upon Secured Entry L2DABL is set to 0x7. Upon re-entering Open Mode, L2DABL is cleared. These bits are always read accessible. These bits are write accessible only in Secured Mode. In the event a DMA access is performed to a restricted memory area a DMA memory access error will occur resulting in a DMA_ERR interrupt and a clearing of DMA_RUN.</p> <p>000 - All DMA accesses are allowed to L2.</p> <p>001 - 1 Kbyte of memory (0xFEB00000 - 0xFEB003FF) has restricted non core access</p> <p>010 - 2 Kbyte of memory (0xFEB00000 - 0xFEB007FF) has restricted non core access</p> <p>011 - 4 Kbyte of memory (0xFEB00000 - 0xFEB00FFF) has restricted non core access</p> <p>100 - 8 Kbyte of memory (0xFEB00000 - 0xFEB01FFF) has restricted non core access</p> <p>101 - 16 Kbyte of memory (0xFEB00000 - 0xFEB03FFF) has restricted non core access</p> <p>110 - 32 Kbyte of memory (0xFEB00000 - 0xFEB07FFF) has restricted non core access</p> <p>111 - 64 Kbyte of memory (0xFEB00000 - 0xFEB0FFFF) has restricted DMA access. This is the initial setting upon entering Secured Entry.</p>

Security Registers

SECURE_SYSSWT (0xFFC04320)

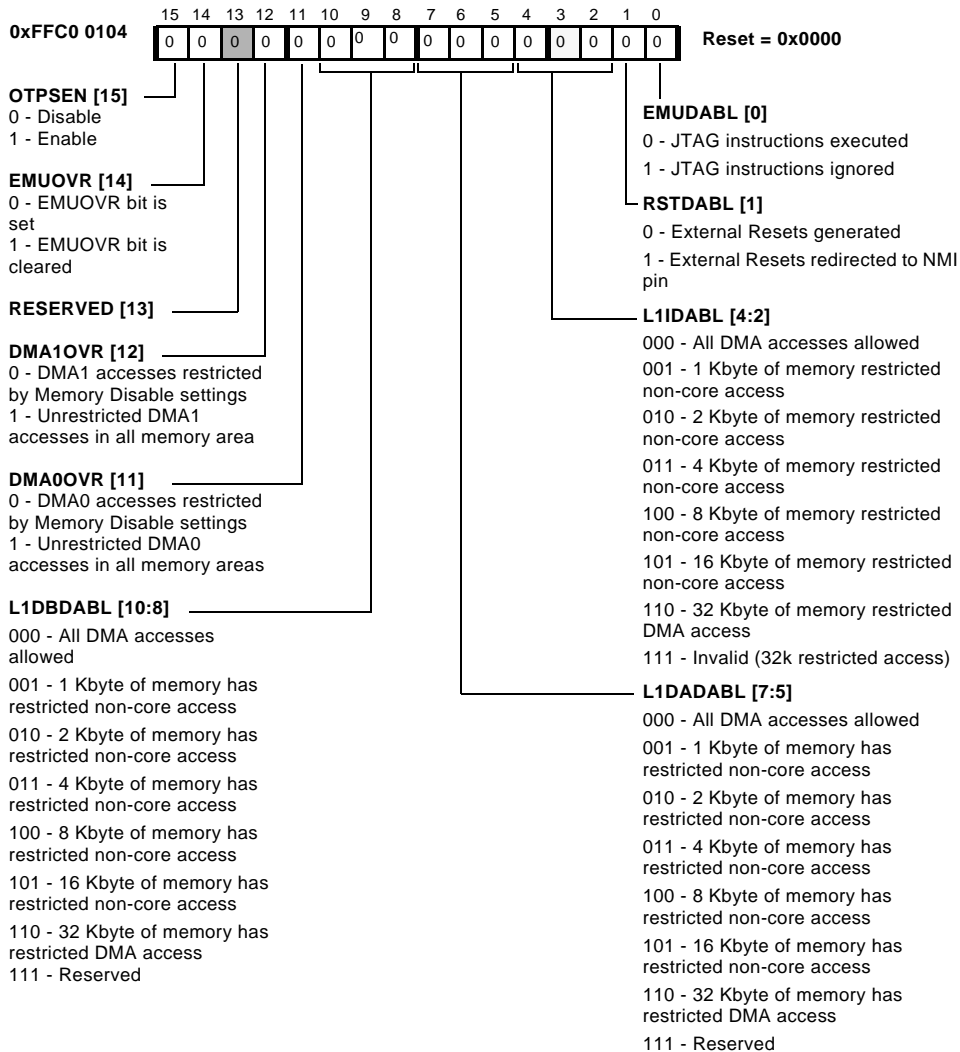


Figure 15-6. SECURE_SYSSWT, Bits 15:0

SECURE_SYSSWT (0xFFC04320)

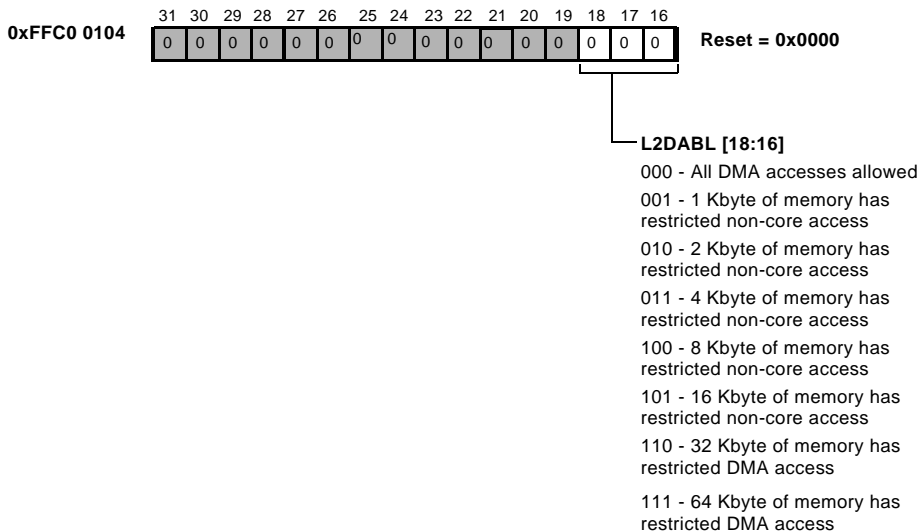


Figure 15-7. SECURE_SYSSWT, Bits 31:16

SECURE_CONTROL (0xFFC04324)

Secure Control (SECURE_CONTROL) is used during Secure Entry Mode authentication. This register is used to establish Secure Mode transition and can be used at any time to exit from Secure Mode. The reset value of the register is 0x0000.

16-bit wide register. Requires 16-bit access.

Table 15-11. SECURE_CONTROL

Bit Position	Bit Name	Bit Description
		Reset = 0x0000
0	SECURE0	<p>SECURE 0</p> <p>This is a write only bit. A read always returns “0”. A 1 value can only be written to SECURE0 when in Secured Entry. The purpose of this control bit is to require 3 successive writes with a value of 1 to SECURE0 in order to enter Secured Mode.</p> <p>0 - When written with a “0” value, all SECURE bits within this register are cleared and Open Mode is entered. All SYSSWT bits are cleared with the exception of EMUOVR. If EMUOVR had been set by the user, it will remain set (until RESET is asserted or until it is written with a “0”).</p> <p>1 - Initially when written with a 1 value SECURE1 is set. With a subsequent 1 written SECURE2 is set. A subsequent 1 written will set SECURE3. Upon a set of SECURE3 Secured Mode is entered.</p>
1	SECURE1	<p>SECURE 1</p> <p>This is a read-only bit and indicates a successful write of SECURE0 with a data value of 1</p> <p>0 - SECURE0 has not been written with a 1 value</p> <p>1 - SECURE0 is written with a 1 value</p>

Table 15-11. SECURE_CONTROL

Bit Position	Bit Name	Bit Description
2	SECURE2	SECURE 2 This is a read-only bit and indicates two successful writes of SECURE0 with a data value of 1 has occurred 0 - SECURE0 has not been written with a 1 value while SECURE1 was set. 1 - SECURE0 is written with a 1 value for a second time.
3	SECURE3	SECURE 3 This is a read-only bit and indicates three successful writes of SECURE0 with a data value of 1 has occurred. 0 - SECURE0 has not been written with a 1 value while SECURE2 was set 1 - SECURE0 is written with a 1 value for a third time. The part is currently in Secured Mode and the SYSSWT register is writable by Authenticated code.

SECURE0 bit is user accessible and is used to exit from Secure Mode. Bits SECURE1, SECURE2, and SECURE3 are not user accessible and are accessed only by the firmware during the digital signature validation process.

Security Registers

SECURE_CONTROL (0xFFC04324)

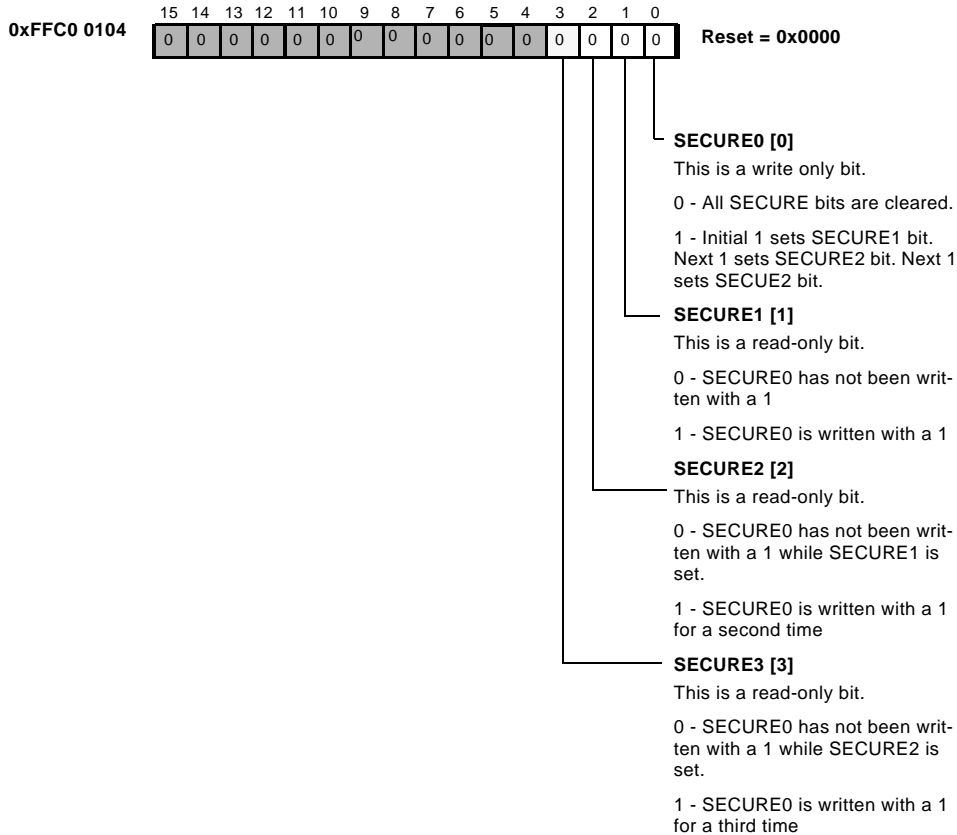


Figure 15-8. SECURE_SYSSWT, Bits 31:16

SECURE_STATUS (0xFFC04328)

Secure Status (SECURE_STATUS) provides information about the current secure state. This information can be used during security mode control as well as understanding why an authentication attempt has failed.

16-bit wide register. Requires 16-bit access.

Table 15-12. SECURE_STATUS

Bit Position	Bit Name	Bit Description
		Reset = 0x0000
1:0	SECMODE	Secured Mode Control State This are read-only bits that reflects the current Secure Mode Control's state. 00 - Open Mode 01 - Secured Entry 10 - Secured Mode 11 - Illegal
2	NMI	This is a read-only bit that reflects the detection of NMI. 0 - Currently NMI is not detected. 1 - Currently NMI is detected.
3	AFVALID	Authentication Firmware Valid This is a read-only bit that reflects the state of the Real Time Trace logic. If execution of authentication has begun properly and has had un interrupted operation the authentication is considered valid. A valid authentication is required for Secured Entry and Secured Mode operation. 0 - Authentication has not begun properly or is interrupted. 1 - Authentication is valid and is progressing properly and uninterrupted.

Security Registers

Table 15-12. SECURE_STATUS

Bit Position	Bit Name	Bit Description
4	AFEXIT	Authentication Firmware Exit This is a write one to clear status bit. In the event authentication has begun properly but has had an improper exit before completion, this bit is set. This can only occur on an exit from Secured Entry back to Open Mode. 0 - No improper exit is made while executing authentication firmware. 1 - An improper exit from authentication firmware is made.
7:5	SECSTAT	Secure Status These are some read write bits which is defined later. These are intended to pass a status back to the handler in the event an authentication has failed. 000 - Reset value 001 - Reserved 010 - Reserved 011 - Reserved 100 - Reserved 101 - Reserved 110 - Reserved 111 - Reserved

NOTE: RTT (AFVALID) is an input to the Secure State Machine and not an output control/status. RTT goes active based on hitting the correct Program Counter address.

SECURE_STATUS (0xFFC04328)

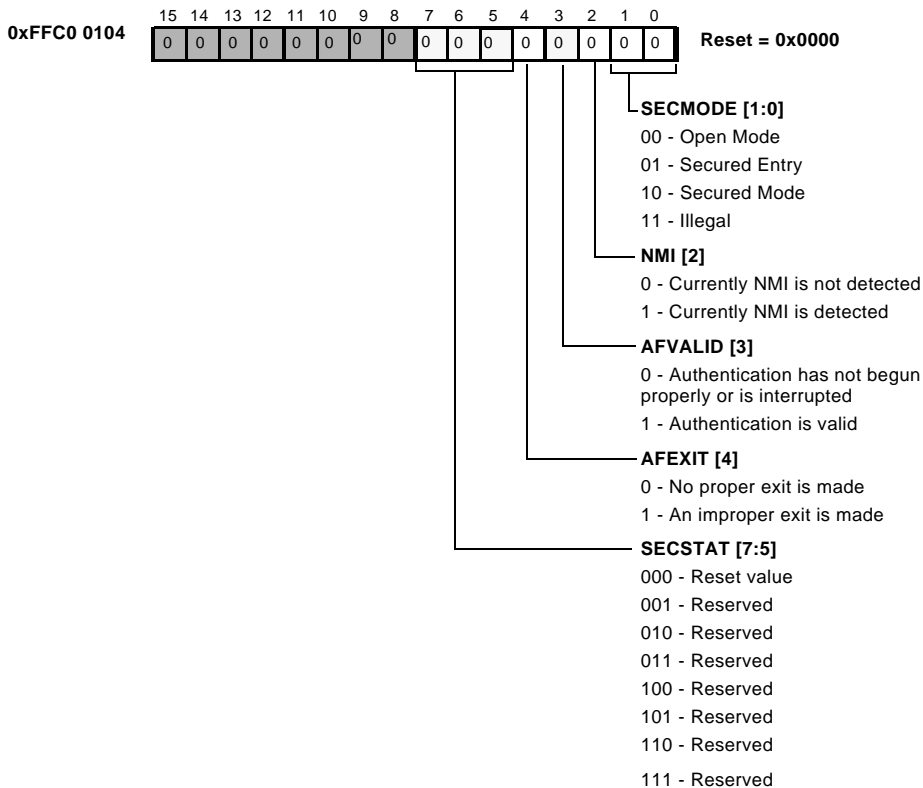


Figure 15-9. SECURE_SYSSWT, Bits 31:16

Security Registers

16 ONE-TIME PROGRAMMABLE MEMORY

This chapter describes One-Time-Programmable (OTP) memory features of the ADSP-BF54x Blackfin processor.

This chapter includes the following sections:

- [“OTP Memory Map” on page 16-2](#)
- [“Error Correction” on page 16-5](#)
- [“OTP Access” on page 16-9](#)
- [“OTP Timing Parameters” on page 16-11](#)
- [“Callable ROM Functions for OTP ACCESS” on page 16-14](#)
- [“Programming and Reading OTP” on page 16-17](#)
- [“Write-protecting OTP Memory” on page 16-25](#)
- [“Accessing Private OTP Memory” on page 16-27](#)
- [“OTP Programming Examples” on page 16-28](#)

OTP Memory Overview


The ADSP-BF54x processors include an on-chip, one-time-programmable (OTP) memory array which provides 64k-bits of non-volatile memory. This includes the array and logic to support read access and programming. A mechanism for error correction is also provided. Additionally, pages can be write protected.

OTP Memory Map

OTP memory can be programmed through various methods including software running on the Blackfin processor. The ADSP-BF54x processors provide C and assembly callable functions in the on-chip ROM to help the developer access the OTP memory.

The One Time Programmable Memory (OTP) is divided into two main regions. A 32k bit “public” unsecured region which has no access restrictions and a 32k bit “private” secured region with access restricted to authenticated code when operating in Secure Mode (For information about these modes, see the section “[Secure State Machine](#)” in [Chapter 15, Security](#) in this volume of the ADSP-BF54x Blackfin Processor Hardware Reference.)

OTP enables developers to store both public and private data on-chip. A 64Kx1bit array is available as shown by [Figure 16-2](#). In addition to storing public and private data, it allows developers to store completely user-definable data such as customer ID, product ID, MAC address, etc.

 The public portion of OTP memory contains many “factory set only” values. Users are urged to exercise caution when writing to OTP memory and to consult the OTP memory map for details of Customer Programmable Settings (CPS) and factory reserved areas of this memory. See also Factory Page Settings (FPS) and Preboot Page Settings (PBS) in [Chapter 17, “System Reset and Booting”](#) in this volume of the ADSP-BF54x Blackfin Processor Hardware Reference.

OTP Memory Map

The OTP is not part of the Blackfin linear memory map. It has a separate memory map that is shown in [Figure 16-2](#). OTP memory is not accessed directly using the Blackfin memory map, rather, it is accessed via four 32-bit wide registers (OTP_DATA3-0) which act as the OTP memory read/write buffer.

One-Time Programmable Memory

In the case of an OTP memory read, the `OTP_DATAx` registers will contain the 16 byte result of the OTP memory access. In the case of an OTP memory write, the `OTP_DATAx` registers will contain 16 bytes of data to be written to the OTP memory.

`OTP_DATA3-0` registers are organized into a 128 bit page as shown in [Figure 16-1](#).

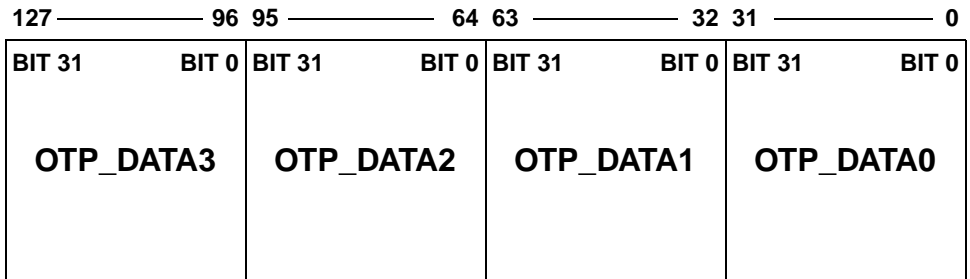


Figure 16-1. `OTP_DATAx` Registers

OTP memory ranges marked as Factory Reserved, Reserved and Error Correction Code Space, in [Figure 16-2](#), must not be programmed by the user. Customer Programmable Settings are optionally programmed by the developer.

Page-Protection bits provide protection for each 128-bit page within the OTP. By default, the OTP array bits are not set and will read back as zero values if left unprogrammed. Programmed data values consist of zeroes and ones, therefore, after programming OTP memory, some bits will intentionally remain as zero values. The write-protect bits provide protection for the zero value bits to remain as zeroes and prevent future programming (inadvertent or malicious) from changing bit values from zero to one.

Pages 0x10, 0x11 and 0x12 hold the customer public key which is used for Lockbox digital signature authentication. Please refer to [Chapter 15, “Security”](#) for more information on Lockbox and how the public key is used.

OTP memory is logically arranged in a sequential set of 128-bit pages. Each OTP memory address refers to a 128-bit page. The ADSP-BF54x thus provides 512 pages of OTP memory.

In order to read or program the OTP memory, a set of functions are provided in the on-chip ROM. These functions include `bfrom_OtpRead()`, `bfrom_OtpWrite()` and `bfrom_OtpCommand()`.

Error Correction

To meet strict quality goals, error correction is used to ensure data integrity. `bfrom_OtpRead()` and `bfrom_OtpWrite()`, provided in the on-chip ROM, support error correction.

Error Correction

Error correction works by calculating an 8-bit Error Correction Code (ECC) for each 64-bit data word (half page) when it is programmed into the OTP. When this word is later read from OTP, its corresponding ECC is also read and a data integrity check is performed. If the check fails, error correction on the data word can be attempted using the ECC. Depending on the type of error, the error correction algorithm will perform as shown in [Table 16-1](#).

Table 16-1. Hamming Code Single Error Corrections, Double Error Detection

No. of bad bits in data word	Error(s) Detected?	Error(s) Corrected?
0	N/A	N/A
1	Yes	Yes
2	Yes	No
3 or more	No	No

Error Correction Policy

1. Error correction requires that OTP space is written and read in 64-bit widths. Firmware will only support writing or reading half of an OTP page.
2. Error Correction is used to correct data in all pages of OTP space except the protection pages (0x0 to 0x3) and ECC pages themselves. See exceptions detailed in the OTP Access section following.
3. Firmware will generate and program the 8-bit ECC fields as mapped in [Table 16-2](#) and [Table 16-3](#).

One-Time Programmable Memory

4. The developer is responsible for locking both the data page(s) AND the ECC page(s) after all programming is complete.
5. Pages 0x04 to 0x0F are reserved for ADI factory use. Therefore, pages 0x004 to 0x00F, 0x0E0, and 0x0E1 will be locked coming out of the Analog Devices factory.

Table 16-2. Mapping for Storage of Error Correction Codes for Unsecured OTP Space

Page	Byte							
	15	14	13	12	11	10	9	8
0x0E0	0x007U	0x007L	0x006U	0x006L	0x005U	0x005L	0x004U	0x004L
0x0E1	0x00FU	0x00FL	0x00EU	0x00EL	0x00DU	0x00DL	0x00CU	0x00CL
0x0E2	0x017U	0x017L	0x016U	0x016L	0x015U	0x015L	0x014U	0x014L
...								
0x0FB	0x0DFU	0x0DFL	0x0DEU	0x0DEL	0x0DDU	0x0DDL	0x0DCU	0x0DCL
Page	7	6	5	4	3	2	1	0
0x0E0	Unused	Unused	Unused	Unused	Unused	Unused	Unused	Unused
0x0E1	0x00BU	0x00BL	0x00AU	0x00AL	0x009U	0x009L	0x008U	0x008L
0x0E2	0x013U	0x013L	0x012U	0x012L	0x011U	0x011L	0x010U	0x010L
...								
0x0FB	0x0DBU	0x0DBL	0x0DAU	0x0DAL	0x0D9U	0x0D9L	0x0D8U	0x0D8L


Error Correction

Table 16-3. Mapping for Storage of Error Correction Codes for Secured OTP Space


Page	Byte							
	15	14	13	12	11	10	9	8
0x1E0	0x107U	0x107L	0x106U	0x106L	0x105U	0x105L	0x104U	0x104L
0x1E1	0x10FU	0x10FL	0x10EU	0x10EL	0x10DU	0x10DL	0x10CU	0x10CL
0x1E2	0x117U	0x117L	0x116U	0x116L	0x115U	0x115L	0x114U	0x114L
....								
0x1FB	0x1DFU	0x1DFL	0x1DEU	0x1DEL	0x1DDU	0x1DDL	0x1DCU	0x1DCL
Page	7	6	5	4	3	2	1	0
0x1E0	0x103U	0x103L	0x102U	0x102L	0x101U	0x101L	0x100U	0x100L
0x1E1	0x10BU	0x10BL	0x10AU	0x10AL	0x109U	0x109L	0x108U	0x108L
0x1E2	0x113U	0x113L	0x112U	0x112L	0x111U	0x111L	0x110U	0x110L
....								
0x1FB	0x1DBU	0x1DBL	0x1DAU	0x1DAL	0x1D9U	0x1D9L	0x1D8U	0x1D8L

OTP Access

The ADSP-BF54x on-chip ROM contains functions for initializing OTP timing parameters, reading and programming the OTP memory. These functions include `bfrom_OtpRead()`, `bfrom_OtpWrite()` and `bfrom_OtpCommand()`.

 These functions are callable from C or assembly application code. Use only these functions for accessing OTP memory. Directly accessing memory locations within OTP memory by other means is not supported.

The existing ECC in ROM is known as “Hamming [72,64]” - This is specifically a 64-bit Data, +8-bit ECC Field, for 1-bit correction and 2-bit error detection scheme.

 The ROM-based OTP read/write API MUST be used for all OTP data accesses (see limited exceptions below). The ROM code incorporates the ONLY ECC method supported by Analog Devices. Analog Devices does not support direct access of OTP data without using error correction.

Exceptions: The only bits that do not use ECC are page lock bits (1st 4 pages) and the preboot invalidate bits. See the Preboot section in [Chapter 17, “System Reset and Booting”](#).

ADI does not support any ECC other than the ECC provided by ADI within the ROM API. All attempts to implement other schemes are not guaranteed or supported by Analog Devices.

OTP memory programming is done serially under software control. Since the unprogrammed OTP memory value defaults to zero, only bits whose value is intended to be “1” have to be programmed. In order to protect areas of OTP memory that have been programmed or areas which have intentionally been left unprogrammed which end users wish to remain unchanged, write-protect bits can be set for each 128-bit page within

OTP Access

OTP memory. Each write-protect bit, when set, will prevent further programming attempts to OTP memory on a per page basis. Please refer to the OTP memory map for more details.


The ADSP-BF54x Blackfin processor can program OTP through software code executing directly on the Blackfin processor. A charge pump residing on-chip is used to apply the voltage levels appropriate for programming OTP memory. OTP programming code can be loaded into the processor via JTAG emulation, DMA, and all supported boot methods.

OTP memory can only be written once (changing a bit from 0 to 1). Once a bit has been changed from a 0 to a 1, it cannot be changed back to 0. The write-protect bits prevent OTP memory that has already been programmed from having any bits that are meant to remain as 0 value later programmed to a value of 1.

Prior to accessing OTP memory, refer to the product data sheet for specifications on V_{DDINT} , V_{DDEXT} voltage levels to ensure reliable OTP programming. OTP timing parameter settings must be set prior to attempting any write accesses to OTP.

OTP Timing Parameters

In order to read and program the OTP memory reliably, the OTP timing parameters are required to be set correctly prior to accessing OTP memory. All the timing parameters are bitfields within the `OTP_TIMING` register shown below. The function, `bfrom_OtpCommand()`, provided in the on-chip ROM, is used to program the timing parameters.

 OTP timing parameters must be set using the `bfrom_OtpCommand()` detailed below. OTP read accesses may use the OTP timing default reset value (Reset: `OTP_TIMING = 0x00001485`). Use of the OTP timing default reset value for writes will result in write errors as this timing value is not appropriate for performing write accesses.

Insufficient voltage/current provided to OTP during write access or incorrect OTP timing parameters may result in the following error returned during OTP writes: `0x11`: error code returned (multiple bad bits in 64 bit data), and subsequent reads from this page return 0.

The OTP timing parameters consist of several fields which are combined together to form one value which is then passed as an argument to the `bfrom_OtpCommand()` function. There are two fields for which the developer must calculate a value based upon the desired SCLK frequency of operation at which the OTP access will be performed. These calculated values are then combined with a third field whose value is provided by Analog Devices to arrive at the setting appropriate for the access.

The OTP timing parameters are comprised of three values as follows:

`OTP_TIMING[7:0] = OTP_TP1 = 1000 / sclk_period`

`OTP_TIMING[14:8] = OTP_TP2 = 400 / (2 * sclk_period)`

`OTP_TIMING[31:15] = OTP_TP3 = 0x0A008`

OTP Access

The OTP_TP3 field is specified by Analog Devices and must be used to ensure reliable OTP write accesses. The user calculated fields must be combined with the OTP_TP3 value as shown in the examples below.

Example calculations are shown [Listing 16-1](#) through [Listing 16-3](#) based upon voltages specified in the *ADSP-BF54x Blackfin Embedded Processor* and OTP timing parameter calculations dependent upon user-defined SCLK frequency of operation. (Please refer to *ADSP-BF54x Blackfin Embedded Processor* data sheet for actual specifications and do not rely on the specifications quoted in these examples.)

Listing 16-1. OTP Timing Calculations for SCLK = 100 MHz

For SCLK = 10ns (100 MHz), the following field calculations are needed to determine the OTP timing argument for the `bfrom_otpCommand()` call.

$OTP_TP1 = 1000 / \text{skl_period} = 1000 / 10 = 0x64$	0x00000064
$OTP_TP2 = 400 / (2 * \text{skl_period}) = 400 / (2 * 10) = 0x14$	0x00001400
OTP_TP3 = (constant)	0x0A008xxx
Calculated OTP timing parameter value:	0x0A009464

The code for API call (in C) is:

```
// Initialize OTP access settings
// Proper access settings for VDDINT = 1V, SCLK = 100 MHz
const u32 OTP_init_value = 0x0A009464;
return_code = bfrom_otpCommand( OTP_INIT, OTP_init_value);
```

Listing 16-2. OTP Timing Calculations for SCLK = 50 MHz

For SCLK = 20.0ns (50 MHz), the following field calculations are needed to determine the OTP timing argument for the `bfrom_otpCommand()` call.

The code for API call (in C) is:

One-Time Programmable Memory

$OTP_TP1 = 1000 / sclk_period = 1000 / 20.0 = 0x32$	0x00000032
$OTP_TP2 = 400/(2 * sclk_period) = 400 / (2 * 20.0) = 0xA$	0x00000A00
$OTP_TP3 = (\text{constant})$	0x0A008xxx
Calculated OTP timing parameter value:	0x0A008A32

```
// Initialize OTP access settings
// Proper access settings for VDDINT = 1V, SCLK = 50 MHz
const u32 OTP_init_value = 0x0A008A32;
return_code = bfrom_otpCommand( OTP_INIT, OTP_init_value);
```

Listing 16-3. OTP Timing Calculations for SCLK = 40 MHz

For SCLK = 25.0ns (40 MHz), the following field calculations are needed to determine the OTP timing argument for the `bfrom_otpCommand()` call.

$OTP_TP1 = 1000 / sclk_period = 1000 / 25.0 = 0x28$	0x00000028
$OTP_TP2 = 400/(2 * sclk_period) = 400 / (2 * 25.0) = 0x8$	0x00000800
$OTP_TP3 = (\text{constant})$	0x0A008xxx
Calculated OTP timing parameter value:	0x0A008828

The code for API call (in C) is:

```
// Initialize OTP access settings
// Proper access settings for VDDINT = 1V, SCLK = 40 MHz
const u32 OTP_init_value = 0x0A008828
return_code = bfrom_otpCommand( OTP_INIT, OTP_init_value);
```

OTP Access

OTP_TIMING Register

OTP_TIMING Register

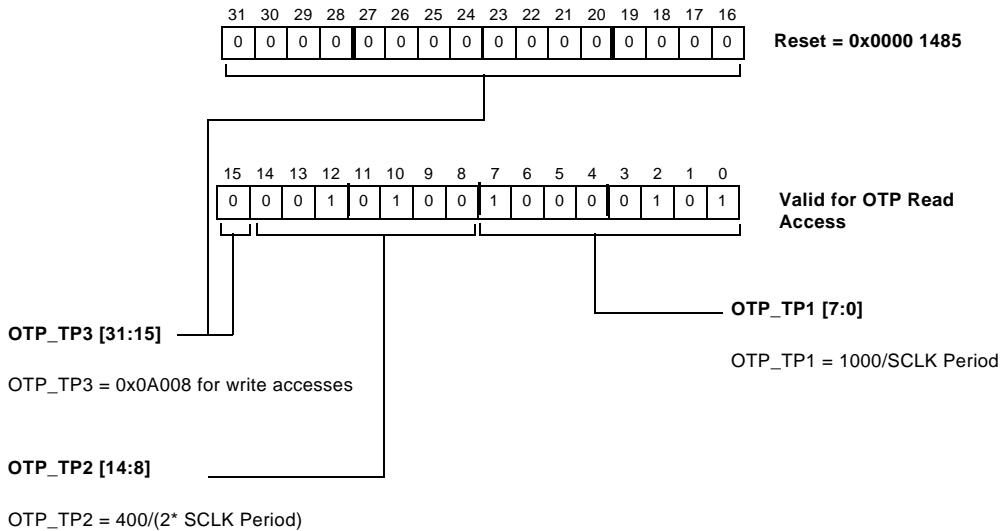


Figure 16-3. OTP_TIMING Register

Callable ROM Functions for OTP ACCESS

The following functions support OTP access.

Initializing OTP

This section describes the usage of `bfrom_otpCommand()` function for OTP memory controller setup provided in the ADSO-54x processor on-chip ROM. The prototype and macros to help decode the function's return

codes are supplied in the `bfrom.h` header file which is located in the VisualDSP++ installation directory. The meaning of the error code is described in section “[Error Codes](#)” on page 16-23.

bfrom_OtpCommand

This function is used to implement various “commands” to setup the OTP controller. The first input parameter is a mnemonic label specifying the command. The second parameter is a generic value that is passed as argument for the requested command. The second parameter is optional and it may be an integer value or (via opportune casting) a pointer or a pointer to an extension structure. There are two commands:

- **OTP_INIT**: sets the required timing value (register `OTP_TIMING`) to “value”.
- **OTP_CLOSE**: reinitializes the OTP controller. This can be called by the user before exiting Secure Mode if desired. The Value parameter may be specified as “0” or “NULL” with `OTP_CLOSE`.

Entry address: 0xEF00 0018

Arguments:

R0: command (dCommand)

`OTP_INIT`

`OTP_CLOSE`

R1: timing value to be programmed (dValue), not used for `OTP_CLOSE`

C Prototype: `u32 bfrom_otpCommand(u32 dCommand, u32 dValue);`

Return code:

`bfrom_otpCommand()` currently always returns with “0”.

OTP Access

From the examples above, the OTP timing parameter was calculated to be **0x0A009464**. Below shows a sample of C code that uses `bfrom_OtpCommand()` function to program this timing parameter.

```
# include <bfrom.h>
# define OTP_TIMING_PARAM (0x0A009464)

u32 Otp_Timing_Param_Init()
{
    u32 otp_timing_parameter;
    u32 = RetVal;
    otp_timing_parameter = OTP_TIMING_PARAM;
    RetVal = bfrom_OtpCommand(OTP_INIT, otp_timing_parameter);
    // (equivalently, with a variable):
    RetVal = bfrom_OtpCommand(OTP_INIT, OTP_TIMING_PARAM);
    return RetVal;
}
```

More examples:

```
// timing parameter
const u32 init_value = 0x0A009464;

// call sets OTP_TIMING register
RetVal = bfrom_OtpCommand(OTP_INIT, init_value);

// call sets OTP_TIMING register
RetVal = bfrom_OtpCommand(OTP_INIT, 0x0A009464);

// call clears OTP controller and data registers
RetVal = bfrom_OtpCommand(OTP_CLOSE, NULL);
```

The prototype of `bfrom_OtpCommand()` is also included in the `bfrom.h` header file installed with VisualDSP++ 5.0 and later releases. Also, the macro `OTP_INIT` is defined in `bfrom.h` as well.

Programming and Reading OTP

This section describes the usage of `bfrom_otpRead()` and `bfrom_otpWrite()` read and write functions for OTP memory provided in the ADSP-54x on-chip ROM. The prototypes and macros to help decode their return codes are supplied in the `bfrom.h` header file which is located in the VisualDSP++ installation directory. The meaning of the error code is described in section “[Error Codes](#)” on page 16-23.

`bfrom_otpRead`

This function is used to read 64-bit OTP half-pages using error correction.

Entry address: 0xEF00 001A

Arguments:

R0: OTP page address (`dPage`)

R1: Flags (`dFlags`)

`OTP_LOWER_HALF`

`OTP_UPPER_HALF`

`OTP_NO_ECC`

R2: Pointer to 64-bit memory struct (long long), to put read data (`*pPageContent`)

C prototype: `u32 bfrom_otpRead (u32 dPage, u32 dFlags, u64 *pPageContent);`

Return code:

R0: error or warning code, see [Table 16-4](#).

OTP Access

This function reads a half-page and stores the content in the 64-bit variable pointed to by its last parameter. The page parameter defines the address. The flags parameter defines whether the upper or the lower half page is to be read. The default reset `OTP_TIMING` value may be used for all read accesses without requiring any new setting value to be programmed prior to performing read accesses. Programming a valid value suitable for write accesses will also allow read accesses.

The use of flag parameter `OTP_NO_ECC` is not recommended for use with any OTP read access as it will bypass error correction code support. It is available only for diagnostic purposes.

bfrom_OtpWrite

This function attempts to write to (program) a half-page with the content in the 64-bit variable pointed to by its last parameter. The page parameter defines the address.

Entry address: 0xEF00 001C

Arguments:

R0: OTP page address (dFlag)

R1: Flags (dFlags)

OTP_LOWER_HALF

OTP_UPPER_HALF

OTP_NO_ECC

OTP_LOCK

OTP_CHECK_FOR_PREV_WRITE

R2: Pointer to 64-bit memory struct (long long) that contains the data to be written to OTP memory (*pPageContent)

C Prototype: u32 bfrom_otpWrite (u32 dPage, u32 dFlags, u64 *pPageContent);

Return code:

R0: error or warning code, see [Table 16-4](#).

OTP Access

The `dFlags` parameter defines whether the upper or the lower half page is to be written to and also if the target half page should be checked for a previously written value before any write attempt is made. Additionally, a page can optionally be locked (permanently protected against further writes).

When performing pure lock operations, the half-page parameter is not required and it makes no difference which half-page is specified if this parameter is included in the function call.

In order to reduce the probability of inadvertent writes to OTP pages, this function checks for a valid OTP write timing setting in the `OTP_TIMING` register. More specifically, bits [31:15] must not be equal to zero. Calls to the write routine when this field is equal to zero cause an access violation error and the requested action is not performed. The user can use this mechanism to protect against inadvertent writes by calling the `bfrom_OtpCommand (OTP_init, ...)` function with appropriate values for reads only and for read/write accesses. He is also free to ignore this mechanism by calling `bfrom_OtpCommand (OTP_init, ...)` only once for read/write access.

When the flag `OTP_CHECK_FOR_PREV_WRITE` is NOT specified, a previously written value will be overwritten, both in the ECC and data fields for any unlocked page where a write access is performed. Of course, once a bit was set to “1” it cannot be reset to “0” by the new write operation. This means

that, in all likelihood, if the new value is different from the previous one, the result will have multiple bit errors, in either or both the ECC and data fields.



Since the ECC field is written first by the ROM function, a multiple bit error will abort the operation without writing the new data value to the OTP data page.

Note also that multiple bit errors have a statistical chance of not being detected as such. So this default mode of operation is not recommended to be used, or used with appropriate caution.

The flag, `OTP_CHECK_FOR_PREV_WRITE`, should always be used by default when performing write accesses to OTP with the `bfrom_otpWrite()` function.

If the flag `OTP_CHECK_FOR_PREV_WRITE` is specified in the call, a write to a previously programmed page causes dedicated error messages and will not be undertaken. More specifically, the criterion for generating errors is as follows: the 64-bit data and the 8-bit ECC field are read and the total number of “1”s is counted. If this number is equal to or greater than 2, the error flag `OTP_PREV_WR_ERR0` is returned and the write operation is not performed. If the number is 0, the page is certainly blank and the write is performed. If the number is one, a more thorough check is performed. If the “1” is in the ECC field, an error flag `OTP_SB_DEFECT_ERROR` is returned and the write is not performed. If the “1” is in the data field, it is determined whether the value to be written contains a “1” in the same position. If so, the write is performed. If not, the error flag `OTP_SB_DEFECT_ERROR` is returned and the write is not performed. This error code warns the user that it could be a single-bit defect in the page. The user can then decide whether to use this page regardless (by repeating the call without the `OTP_CHECK_FOR_PREV_WRITE` flag) or skip this page.


The `OTP_CHECK_FOR_PREV_WRITE` flag is ignored when a pure lock operation is requested (for example, a `OTP_LOCK` flag is set and `&DATA = NULL`). It is therefore unnecessary and harmless to specify this flag. The

OTP Access

`OTP_CHECK_FOR_PREV_WRITE` flag is not ignored when doing a lock operation after a write (for example, `OTP_LOCK + write` in the same call and `&DATA = NULL`).

If the flag parameter for the write operation is augmented by the OR with `OTP_LOCK` flag, the write operation, if successful, will be immediately followed by setting the protection bit for the requested full 128-bit page.

A special case is the following (`OTP_LOCK`): if the third parameter is `NULL`, this call will lock a page without writing any data value to it (pure lock function). Note that in this case, “page” can span all pages from `0x000` to `0x1FF`. **This is the only way to lock the ECC pages themselves.**

 The use of flag parameter `OTP_NO_ECC` is only supported in write operations when used to implement write-protection/ page-locking (use of `OTP_LOCK` parameter in `bfrom_Otp_Write` function is preferred method of locking pages, see Write Protecting OTP Memory section below) or to set the preboot invalidate bits (see the Preboot section in [Chapter 17, “System Reset and Booting”](#)). Bypassing error correction in OTP writes may result in loss of OTP data integrity and is not supported for any other OTP access.

The use of ECC in all OTP accesses other than the limited exceptions described previously is mandatory.

Error Codes

This section describes the returned error codes from the API functions. [Figure 16-4](#) and [Table 16-4](#) demonstrate and list the returned error codes from API functions.

Returned Error Codes from API Functions

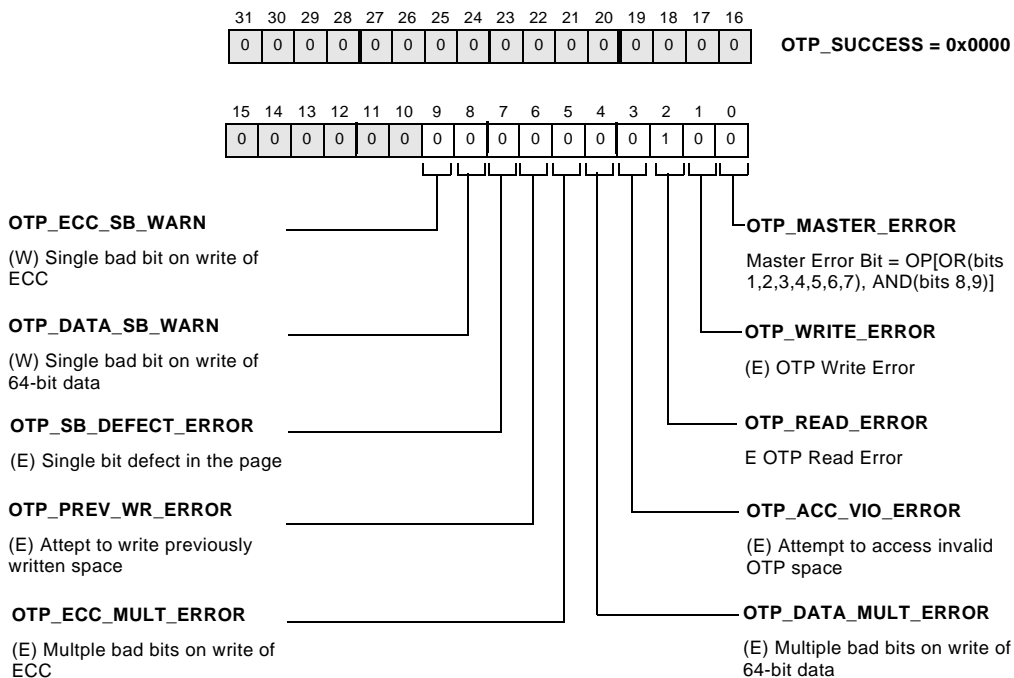


Figure 16-4. Returned Error Codes from API Functions

`bfrom_otpCommand()` currently always returns with “0”.

OTP Access

Table 16-4. Returned Error Codes from API Functions

Bit Position	Name	Example Return Value	Definition
N/A	OTP_SUCCESS	0x0	No Error
0	OTP_MASTER_ERROR	0x1	Master Error Bit = OR[OR(bits 1,2,3,4,5,6,7), AND(bits 8,9)]
1	OTP_WRITE_ERROR	0x3	(E) OTP Write Error
2	OTP_READ_ERROR	0x5	(E) OTP Read Error
3	OTP_ACC_VIO_ERROR	0x9	(E) Attempt to access invalid OTP space
4	OTP_DATA_MULT_ERROR	0x11	(E) Multiple bad bits on write of 64 bit data
5	OTP_ECC_MULT_ERROR	0x21	(E) Multiple bad bits on write of ECC
6	OTP_PREV_WR_ERROR	0x41	(E) Attempt to write previously written space
7	OTP_SB_DEFECT_ERROR	0x81	(E) Single-bit defect in the page
8	OTP_DATA_SB_WARN	0x100	(W) Single bad bit on write of 64 bit data
9	OTP_ECC_SB_WARN	0x200	(W) Single bad bit on write of ECC


`bfrom_OtpRead()` returns with an error when any of the bits [6:2] are set or both bits[9:8] are set. In this case, the `OTP_MASTER_ERROR` bit is also set. It returns with a warning, if only one of the bits [9:8] is set.

`bfrom_OtpWrite()` returns with an error when any of the bits [7:1] are set or both bits[9:8] are set. In this case, the `OTP_MASTER_ERROR` bit is also set. It returns with a warning, if only one of the bits [9:8] is set.

Write-protecting OTP Memory

As shown in [Figure 16-2](#), a small portion of OTP memory is reserved for write-protect bits (“write-protect” is synonymous with “page-protect” in the context of this discussion). After programming OTP memory, the programmer can use these protection bits to “lock” the page that was just programmed by setting the write-protect bit corresponding to the OTP data page. Once the write-protect bit is set and the lock is in place, further attempts to write to that page will not be allowed, resulting in an error. Page protect bits can also be set in order to prevent programming of unwritten OTP pages as well. Once an OTP page is page-protected, the write protection can not be reversed and no further write accesses can be made to the protected page(s).

There are four pages reserved for the write-protection bits. Pages 0x0 through 0x3 contain the 512 write-protect bits, one bit for each of the 512 data pages within OTP memory. The first two write-protect bit pages (pages 0x0 and 0x1) correspond to the public (non-secure) regions of the OTP map. The other two write-protect bit pages (0x2 and 0x3) correspond to the protection of private (secure) regions of the OTP map. The processor does not need to be operating in Secure Mode in order to be able to program protection pages associated with secure OTP regions. All protection bits can be written in any security state including Open Mode.

 Note that while reads and writes access a half-page at a time, setting a protection bit for a page will effectively lock an entire page for future write accesses (lower and upper half page). The programmer must ensure that all required programming is completed on a full 128-bit OTP data page prior to setting the write-protect bit for that page. In other words, the programmer must make sure that a full 128-bit OTP page is programmed, or that no future programming is required to be performed to the unprogrammed portion of the page before locking the page.

OTP Access

If P is the OTP page that is needed to be write-protected, the write-protect bit and its page can be calculated as follows:

Let WPP be the write-protect page where the write-protect bit resides and let WPB be the write-protect bit that needs to be set in order to lock page P .

The write-protect page can be calculated by:

$$WPP = P \gg 7$$

and the write-protect bit can be calculated by:

$$WPB = P \& 0x7f.$$

Manual calculation is largely unnecessary due to the fact that the `bfrom_OtpWrite()` function can be used to lock pages (see Programming Examples section below for more details).

```
// lock page (note third parameter equals NULL)
return_code = bfrom_OtpWrite( 0x01C, OTP_LOCK, NULL);
```


Locking a single ECC (error correction code) page results in locking the correction codes which correspond to eight OTP data pages (16 half pages). This is due to the fact that a 64-bit half-page access must be performed when write protecting the ECC page and every 8-bits within an ECC page is a parity correction code which corresponds to a 64-bit half-page of data in OTP. Therefore, a full 128-bit ECC page holds the correction codes for eight full 128-bit pages of data in OTP, or 16 half-pages. Pages can only be locked as full 128-bit pages even though read/write accesses may occur at 64-bit half-page granularity. Locking a single ECC page will prevent further write access to the corresponding eight OTP data pages.

ECC (error correction code) space is not permitted to be written to directly.

For example, locking ECC page 0xFB will result in locking the error correction parity data associated with the 16 data pages in the range of 0x0D8 – 0x0DF.

```
// Only Lock ECC code page
return_code = bfrom_OtpWrite(0xFB, OTP_LOCK, NULL);
```

No further write accesses to the ECC page 0xFB or corresponding data pages 0x0D8 – 0x0DF will be allowed following write protection of the ECC page in this example.

 Bits [3:0] of OTP page 0 are the write-protect bits for the first four OTP pages, which contain the write-protect bits. If these bits are set, it will prevent the other write-protect bits from being set, thus disabling the write protection mechanism. But, this does not prevent the user from programming the other user-programmable OTP pages.

Accessing Private OTP Memory

In order to read or write to the private area of OTP memory, the processor must be operating in Secure Mode and the `OTPSSEN` bit within the `SECURE_SYSSWT` register must be set to a value of 1 to enable secured OTP access. (For information about Security, Secure Mode and the Secure State Machine, see the Secure State Machine section of Chapter 15, Security).

OTP Programming Examples

To enable access to private OTP memory space while operating in Secure Mode, use the code shown in [Listing 16-4](#).

Listing 16-4. Enable access to private OTP

```
// Enable private OTP access
*pSECURE_SYSSWT = *pSECURE_SYSSWT | OTPSEN;
ssync();
...
```

To enable access to private OTP memory space via `OTPSSEN` while operating in Secure Mode, use the code shown in [Listing 16-5](#).

Listing 16-5. Enable access to private OTP and enable JTAG emulation in Secure Mode

```
// Enable JTAG and private OTP access
*pSECURE_SYSSWT = *pSECURE_SYSSWT & (~EMUABL) | OTPSEN;
SSYNC(0);
...
```

To read pages 0x4 through 0xDF in public OTP memory space and print results to VisualDSP++ console, use the code shown in [Listing 16-6](#).

Listing 16-6. Read pages 0x4 through 0xDF in public OTP memory space and print results to VisualDSP++ console

```
# include <blackfin.h>
# include <bfrom.h>

u32 return_code, i;
u64 value;

// initialize OTP timing parameter
// Proper timing for VDDINT = 1v, CCLK, SCLK = 100MHz
const u32 OTP_init_value = 0x0A009464;
return_code = bfrom_otpCommand( OTP_INIT, OTP_init_value);

...

for (i= 0x004; i,0x0xE0; i++)
{
return_code = bfrom_otpRead(i, OTP_LOWER_HALF, &value);
printf("page: 0x%03xL, Content ECC: 0x%01611x, returncode:
0x%03x \n", i, value, return_code);

return_code = bfrom_otpRead(i, OTP_UPPER_HALF, &value);
printf("page: 0x%03xH, Content ECC: 0x%01611x, returncode:
0x%03x \n", i, value, return_code);
}
```

OTP Programming Examples

To write and lock a single OTP page and return the results to the VisualDSP++ console via `printf`, use the code shown in [Listing 16-7](#).

Listing 16-7. Perform OTP write to a single page via two 64-bit (half-page) accesses

```
# include <blackfin.h>
# include <bfrom.h>

u64 value;
u32 return_code;

return_code = bfrom_OtpWrite(0x01C, OTP_LOWER_HALF |
OTP_CHECK_FOR_PREV_WRITE, &testdata);
printf("WRITE page: 0x%03xL, Content ECC: 0x%01611x,
returncode: 0x%03x \n", 0x1C, testdata, return_code);

return_code = bfrom_OtpWrite(0x01C, OTP_UPPER_HALF |
OTP_CHECK_FOR_PREV_WRITE | OTP_LOCK, &testdata);
printf("WRITE page: 0x%03xH, Content ECC: 0x%01611x,
returncode: 0x%03x \n", 0x1C, testdata, return_code);
}
```

Note that locking a page will lock the full 128-bit page whereas the examples shown above perform OTP access on a 64-bit half-page granularity. This is the finest level of granularity that is allowed due to the OTP error correction implementation. The page lock should occur only after both the lower and upper portion of the page have been written. Note that the page lock operation is performed on the second and final access to the page in the code in [Listing 16-7](#).

It may be desired to lock some specific OTP pages in a separate access after writing of data values is already complete.

OTP pages are typically locked in order to protect them from being overwritten or to prevent inadvertent or malicious tampering. This can be performed by the following instructions in [Listing 16-8](#):

Listing 16-8. Perform pure page lock operation without writing any data values:

```
#include <blackfin.h>
#include <bfrom.h>

u64 value;
u32 return_code;

// initialize OTP timing parameter
// Proper timing for VDDINT = 1V, CCLK, SCLK = 100MHz

const u32 OTP_init_value = 0x0A009464;

return_code = bfrom_OtpCommand( OTP_INIT, OTP_init_value);

return_code = bfrom_OtpWrite(0x01C, OTP_LOCK, NULL);
```

OTP Programming Examples

17 SYSTEM RESET AND BOOTING

This document contains material that is subject to change without notice. The content of the boot ROM as well as hardware behavior may change across silicon revisions. See the anomaly list for differences between silicon revisions. This document describes functionality of silicon revision 0.2 of the ADSP-BF542/ ADSP-BF544/ ADSP-BF547/ ADSP-BF548/ ADSP-BF549 (ADSP-BF54x) processors.

Overview

When the $\overline{\text{RESET}}$ input signal releases, the processor starts fetching and executing instructions from the on-chip boot ROM at address 0xEF00 0000.

The internal boot ROM includes a small boot kernel that loads application data from an external memory or host device. The application data is expected to be available in a well-defined format called the boot stream. A boot stream consists of multiple blocks of data and special commands that instruct the boot kernel how to initialize on-chip L1 and L2 SRAM memories as well as off-chip volatile memories.

The boot kernel processes the boot stream block-by-block until it is instructed by a special command to terminate the procedure and jump to the application's programmable start address, which traditionally is at 0xFFA0 0000 in on-chip L1 memory. This process is called "booting."

Overview

The processor features four dedicated input pins `BMODE[3:0]` that select the booting mode. The boot kernel evaluates the `BMODE` pins and performs booting from respective sources. [Table 17-1](#) describes the modes of the `BMODE` pins.

Table 17-1. Booting Modes

<code>BMODE[3:0]</code>	Boot Source	Description
0000	No boot - idle	The processor does not boot. Rather, the boot kernel executes an IDLE instruction.
0001	Boot from 8-bit or 16-bit external flash memory	The kernel boots from address 0x2000 0000 in asynchronous memory bank 0. The first byte of the boot stream contains further instructions whether the memory is eight or 16 bits wide.
0010	Boot from 16-bit asynchronous FIFO	By using the handshaked memory DMA (HMDMA1) feature through the DMAR1 input, the kernel boots from address 0x2030 0000 in asynchronous memory bank 3.
0011	Boot from serial SPI memory	After an initial device detection routine, the kernel boots from either 8-bit, 16-bit, 24-bit or 32-bit addressable SPI flash or EEPROM memory that connects to <code>SPI0_SSEL1</code> .
0100	Boot from SPI host	In this slave mode, the kernel expects the boot stream to be applied to SPI0 by an external host device.
0101	Boot from serial TWI memory	The kernel boots from TWI memory connected to TWI0. Memory is expected to respond to the unique slave identifier of 0xA0.
0110	Boot from TWI host	In this slave mode, the kernel expects the boot stream to be applied to TWI0 by an external host device. The Blackfin processor uses the slave identifier 0x5F.
0111	Boot from UART host	In this slave mode, the kernel expects the boot stream to be applied to UART1 by an external host device. The <code>UART1RTS</code> output is active and controlled by hardware. Prior to providing the boot stream, the host device is expected to send a 0x40 (ASCII '@') character that is examined by the kernel to adjust the bit rate.

Table 17-1. Booting Modes (Cont'd)

BMODE[3:0]	Boot Source	Description
1000	Reserved	
1001	Reserved	
1010	Boot from SDRAM memory ¹	This mode provides a quick warm boot option. It requires the SDRAM controller to be programmed by the preboot routine based on OTP settings. The kernel starts booting from address 0x0000 0010.
1011	Boot from on-chip OTP memory	This is the only stand-alone booting mode. It boots from the on-chip serial OTP memory. By default, the boot stream is expected to reside from OTP page 0x40 on. The start page can be altered by programming the OTP_START_PAGE field in OTP page PBS01H.
1100	Reserved	
1101	Boot from 8- and 16-bit NAND flash	The boot kernel automatically detects whether an 8-bit small page device or an 8-/16-bit large page device is connected to the NFC. The NAND flash may optionally contain further initialization code that enables some more advanced boot options.
1110	Boot from 16-bit Host DMA	The kernel initializes the Host DMA unit to 16-bit ACK mode. Boot stream parsing is up to the host device. An HIRQ command causes the kernel to issue a CALL to the address 0xFFA0 0000.
1111	Boot from 8-bit Host DMA	The kernel initializes the Host DMA unit 8-bit INT mode. Boot stream parsing is up to the host device. An HIRQ command causes the kernel to issue a CALL to the address 0xFFA0 0000.

¹ This chapter uses the term SDRAM as a synonym for off-chip synchronous dynamic memory. For the ADSP-BF54x products, SDRAM memory complies with either the DDR1 SDRAM or the Mobile DDR1 SDRAM standard.

Reset and Power-up

There is a subroutine in the boot kernel known as "preboot", which is executed prior to the boot mode being processed. This preboot routine can customize default values of MMR registers, such as the PLL and SDRAM controller registers. Furthermore, SPI and TWI master modes can be customized. The preboot behavior is controlled through OTP programming.

To enable booting into volatile memories such as SDRAM, the SDRAM controller must be programmed *before* data can be loaded into the memory. Either the preboot or the initialization code mechanism can be used for this purpose.

Table 17-2 describes the six types of resets.


 All resets described reset the core except for the System Software reset.

Table 17-2. Resets

Reset	Source	Result
Hardware reset	The RESET pin causes a hardware reset.	Resets both the core and the peripherals, including the dynamic power management controller (DPMC). Resets bits [15:4] of the SYSCR register. For more information, see “ System Reset Configuration (SYSCR) Register ” on page 17-109.
Wake up from hibernate state	Wake-up event as enabled in the VR_CTL register and reported by the PLL_STAT register.	Behaves as hardware reset except the WURESET bit in the SYSCR register is set. Booting can be performed conditionally on this event.
System software reset	Calling the <code>bfrom_SysControl()</code> routine with the <code>SYSCTRL_SYSRESET</code> option triggers a system reset.	Resets only the peripherals, excluding the RTC (real time clock) block and most of the DPMC. The system software reset clears bits [15:13] and bits [11:4] of the SYSCR register, but not the WURESET bit. The core is not reset and a boot sequence is not triggered. Sequencing continues at the instruction after <code>bfrom_SysControl()</code> returns.

Table 17-2. Resets (Cont'd)

Reset	Source	Result
Watchdog timer reset	Programming the watchdog timer causes a watchdog timer reset.	Resets both the core and the peripherals, excluding the RTC block and most of the DPMC (Because of the partial reset to the DPMC, the watchdog timer reset is not functional when the processor is in Sleep or Deep Sleep modes.). The SWRST or the SYSCR register can be read to determine whether the reset source was the watchdog timer.
Core double-fault reset	A core double fault occurs when an exception happens while the exception handler is executing. If the core enters a double-fault state, a reset can be caused by unmasking the DOUBLE_FAULT bit in the SWRST register.	Resets both the core and the peripherals, excluding the RTC block and most of the DPMC. The SWRST or SYSCR registers can be read to determine whether the reset source was a core double-fault.
Software reset	This reset is caused by executing a RAISE 1 instruction or by setting the software reset (SYSRST) bit in the core debug control register (DBGCTL) through emulation software through the JTAG port. The DBGCTL register is not visible to the memory map.	Program executions vector to the 0xEF00 0000 address. The boot code immediately a system reset to ensure system for consistency.

Hardware Reset

The processor chip reset is an asynchronous reset event. The $\overline{\text{RESET}}$ input pin must be deasserted after a specified asserted hold time to perform a hardware reset. For more information, see the product data sheet.

Reset and Power-up

A hardware-initiated reset results in a system-wide reset that includes both core and peripherals. After the $\overline{\text{RESET}}$ 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 state of the BMODE pins.

The BMODE 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 V_{DDEXT} or GND. The pins and the corresponding bits in the SYSCR register configure the boot mode that is employed after hardware reset or system software reset. See the *Blackfin Processor Programming Reference* for further information.

Software Resets

A software reset may be initiated in three ways.

- By the watchdog timer, if appropriately configured
- Calling the `bfrom_SysControl()` API function residing in the on-chip ROM. For further information, see [Chapter 18, “Dynamic Power Management”](#).
- By the `RAISE 1` instruction

The watchdog timer resets both the core and the peripherals, as long as the processor is in Active or Full-On mode. A system software reset results in a reset of the peripherals without resetting the core and without initiating a booting sequence.



In order to perform a system reset, the `bfrom_SysControl()` routine must be called while executing from L1 memory (either as cache or as SRAM). When L1 instruction memory is configured as cache, make sure the system reset sequence is read into the cache.

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 formatting the watchdog timer, the processor transitions into the boot mode sequence. The boot mode is configured by the state of the `BMODE` bit field in the `SYSCR` register.

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

A software reset only affects the state of the core. The boot kernel immediately issues a system reset to keep consistency with the system domain.

Reset Vector

When reset releases, the processor starts fetching and executing instructions from address `0xEF00 0000`. This is the address where the on-chip boot ROM resides.

On a hardware reset, the boot kernel initializes the `EVT1` register to `0xFFA0 0000`. When the booting process completes, the boot kernel jumps to the location provided by the `EVT1` vector register. With the exception of the `HOSTDP` boot modes, the content of the `EVT1` register is overwritten by the target address field of the first block of the applied boot stream. If the `BCODE` field of the `SYSCR` register is set to 3 (no boot option), the `EVT1` register is not modified by the boot kernel on software resets. Therefore, programs can control the reset vector for software resets through the `EVT1` register. This process is illustrated by the flow chart in [Figure 17-1](#).

Reset and Power-up

The content of the EVT1 register may be undefined in emulator sessions.

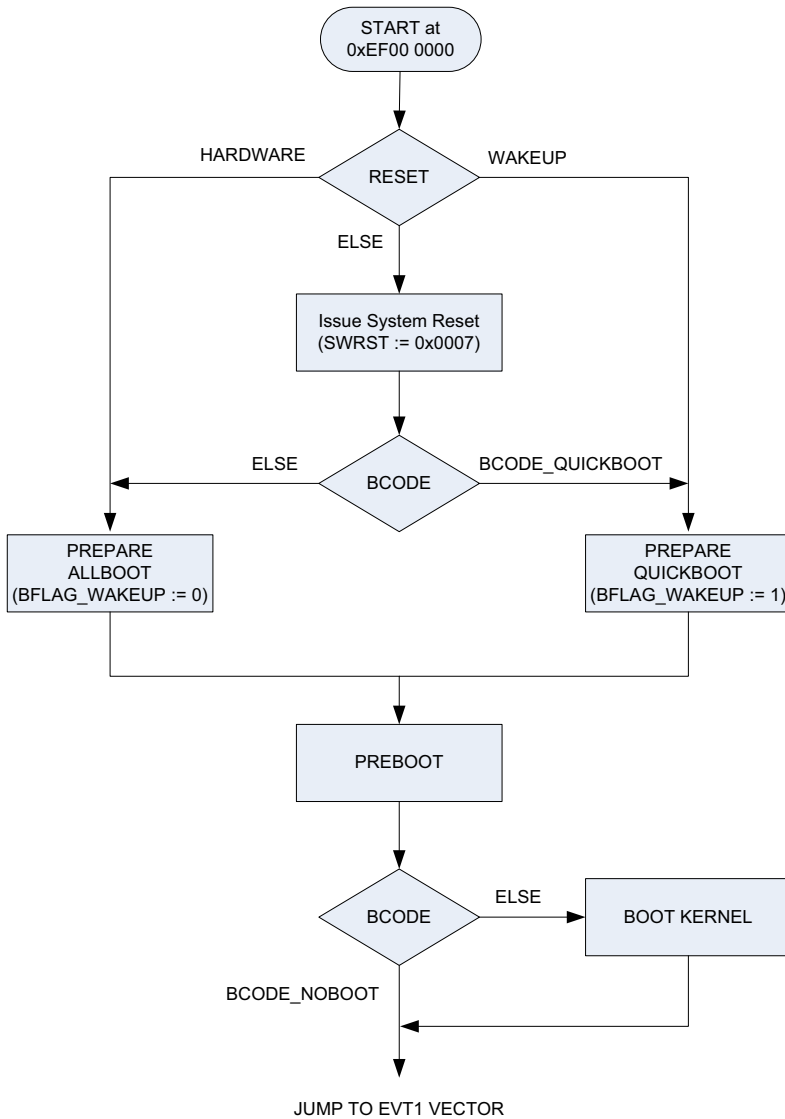


Figure 17-1. Global Boot Flow

Servicing Reset Interrupts


The processor services a reset event like other interrupts. The reset interrupt has top priority. Only emulation events have higher priority. When coming out of reset, the processor is in supervisor mode and has full access to all system resources. The boot kernel can be seen as part of the reset service routine. It runs at the top interrupt priority level.

Even when the boot process has finished and the boot kernel passes control to the user application, the processor is still in the reset interrupt. To enter user mode, the reset service routine must initialize the `RETI` register and terminate with an `RTI` instruction.

For an example, see “[System Reset](#)” in the “[Programming Examples](#)” on [page 17-154](#).

The code examples in [Listing 17-4](#) and [Listing 17-3 on page 17-155](#) show the instructions required to handle the reset event. See the *Blackfin Processor Programming Reference* for details on user and supervisor modes.

Systems that do not work in an OS environment may not enter user mode. Typically, the interrupt level needs to be degraded down to `IVG15`. [Listing 17-4](#) and [Listing 17-3](#) show how this is accomplished.

 As the boot kernel is running at reset interrupt priority, NMI events, hardware errors and exceptions are not served at boot time. As soon as the reset service routine returns, the processor may service the events that occurred during the boot sequence. It is recommended that programs install NMI, hardware error, and exception handlers before leaving the reset service routine. This includes proper initialization of the respective event vector registers, `EVTx`.

Preboot

After reset, the boot kernel residing in the on-chip boot ROM does not immediately start processing the boot stream. Rather, it first calls a subroutine called preboot, as shown in [Figure 17-2 on page 17-16](#) and [Figure 17-3 on page 17-17](#). The preboot routine customizes the default values of several system MMR registers based on user-configurable OTP (one-time programmable) memory. The following modules can be customized in this way.

- PLL and voltage regulator settings
- SDRAM controller settings
- Asynchronous EBIU settings

Some OTP bits customize the boot process:

- Bit rate of SPI and TWI master boot modes
- TWI master boot addressing scheme
- Activation of SPI fast read mode
- Boot host wait (HWAIT) signal

Further OTP bits let the user disable certain features of the processor:

- Individual boot modes (for security reasons)

Finally, certain bits are already preset in the factory:

- USB voltage trim
- Individual boot modes

Factory Page Settings (FPS)

The content of the boot ROM is identical across all ADSP-BF54x Blackfin processors. The factory settings prevent the boot ROM from accidentally accessing resources that are not present on a given processor, which would result in unpredictable behavior and/or hardware errors. The boot kernel goes to a safe idle state when the user configures the `BMODE` pins to a boot mode that is not available on a specific part.

For this purpose, the preboot routine always reads the `FPS01L` and `FPS01H` half pages from OTP memory. In addition, these half pages contain factory trim values for the USB PHY controller that are managed at preboot time, as required.

In addition, the `bfrom_SysControl()` routine reads the half page `FPS04H` and `FPS04L` to apply factory trim values to the voltage regulator and SDRAM controller.

Preboot Page Settings (PBS)

Four OTP pages optionally enable the user to customize the behavior of the processor immediately after reset. These four pages (eight half pages) can be seen as one contiguous pre-boot settings (PBS) block. By default, the block spans OTP pages 0x18 to 0x1B. The OTP pages serve the following purposes:

- PBS00L (by default, on half page 0x18L, see [“Lower PBS00 Half Page” on page 17-115](#) for details)
 - PLL and voltage regulator settings
 - Boot customization
 - Instruction whether to load further half pages
- PBS00H (by default, on half page 0x18H, see [“Upper PBS00 Half Page” on page 17-119](#) for details)
 - Asynchronous EBIU register settings
- PBS01L (by default on half page 0x19L)
 - Reserved
- PBS01H (by default, on half page 0x19H, see [“Upper PBS01 Half Page” on page 17-120](#) for details)
 - Disabling of boot modes
 - NFC controller register settings
 - OTP boot start page
- PBS02L (by default, on half page 0x1AL, see [“Lower PBS02 Half Page” on page 17-122](#) for details)
 - Synchronous EBIU register settings

- PBS02H (by default, on half page 0x1AH, see “[Upper PBS02 Half Page](#)” on page 17-124 for details)

Synchronous EBIU register settings

- PBS03L (by default, on half page 0x1BL, see “[Reserved Half Pages](#)” on page 17-126 for details)

Reserved in current silicon revision. Do not use.

- PBS03H (by default, on half page 0x1BH, see “[Reserved Half Pages](#)” on page 17-126 for details)

Reserved in current silicon revision. Do not use.

The preboot routine reads the main page `PBS00L` first. Since this page may instruct the preboot routine to alter the PLL settings, further pages may read more quickly. This page also instructs the preboot whether further OTP half pages have to be loaded and processed. By default, the `PBS00L` page reads all zeroes, and the preboot does not load further PBS pages.

Alternative PBS Pages


Especially during the development cycle, the user may fail to write the proper value to OTP memory and may make multiple attempts to get things right. Therefore, the `PBS00L` page provides a mechanism to invalidate the entire PBS block (consisting of pages (0x18, 0x19, 0x1A and 0x1B) and to use pages 0x1C to 0x1F instead. To do so, set the two `OTP_INVALID` bits (bits 62 and 63 on the `PBS00L` page). If both bits are set, the preboot routine disregards potential error codes returned by the `bfrom_OtpRead()` routine and continues processing from page 0x1C on. The active PBS block now spans the pages 0x1C to 0x1F. If the user wants to invalidate the second set of OTP pages as well, setting bits 62 and 63 on page 0x1C (which is the new `PBS00L` half page) instructs the preboot routine to continue at page 0x20, and so on.

Preboot

Theoretically, this can be repeated up to page 0xD8L, if the pages are not required for other purposes. There are 49 chances to get things right, before a device may become useless. Note that every page that needs to be read by the preboot routine causes additional delay to the boot process.

Programming PBS Pages

Due to the need for ECC error correction, a 64-bit OTP half page must be written all at once. It is recommended that PBS pages be programmed only through the API function `bfrom_OtpWrite()`.

 If it is anticipated that the user is customizing the boot-related OTP pages for safety or security reasons, it is recommended that all PBS blocks be locked at production time to protect these pages from being tampered with in the field.

Reading OTP memory is subject to a potential failure rate. Since the preboot only accesses OTP memory through the `bfrom_OtpRead()` function, the ECC error correction is applied and the statistical failure rate is very low. However, the way the `PBS00L` page is tested for being invalid may at some point reduce the ECC reliability. To keep failure rates at a minimum, it is a good idea to duplicate the content of pages 0x18–0x1B on pages 0x1C–0x1F. For production parts, the final block should be followed by its exact copy to maintain the lowest failure rates. Then, even the unlikely case where one of the `OTP_INVALID` bits is read incorrectly would not cause the boot to fail.

Recovering From Misprogrammed PBS Pages

The preboot mechanism provides a powerful method to customize the chip to the needs of the user. However, as a downside, there are chances that invalid values programmed to the PBS pages prevent the processor from operating within required operating conditions. There is specific risk when the PLL and the voltage regulator are programmed with meaningless values during the development cycle.

In such cases, the boot mode `BMODE = b#0000` helps. In this mode, the preboot routine does not attempt to read any of the user-programmable PBS pages, and the boot kernel does not try to boot any data. Rather, the processor is idled immediately after the FPS pages have been processed. Using the in-circuit emulator, the user then has the option to invalidate the actual PBS settings by overwriting both `OTP_INVALID` bits in the actual `PBS00L` with 1s.

For safety reasons, none of the boot modes, except the emulator, can get control over the processor when in this state.

Customizing Power Management

When the processor awakes with default PLL and voltage regulator settings, the preboot mechanism can be used to alter these settings to custom values before the boot process takes place. This is done by programming the OTP half page `PBS00L`.

If the `OTP_SET_PLL` bit is programmed to a 1, the value in the `OTP_PLL_DIV` bit field is copied into the `PLL_DIV` register, and the `OTP_PLL_CTL` bit field is copied into the `PLL_CTL` register, followed by the required `IDLE` instruction (if the contents of `PLL_CTL` are being altered).

If the `OTP_SET_VR` bit is programmed to a 1, the value in the `OTP_VR_CTL` bit field is copied into the `VR_CTL` register, followed by the required `IDLE` instruction (if the contents of `VR_CTL` are being altered).

The preboot mechanism invokes the `bfrom_SysControl()` routine to alter the PLL and the voltage regulator. The `bfrom_SysControl()` routine not only performs custom instructions, it also applies correction values from factory OTP pages `FPS01` and `FPS04`. See [Chapter 18, “Dynamic Power Management”](#) for details on the `bfrom_SysControl()` routine.

Preboot

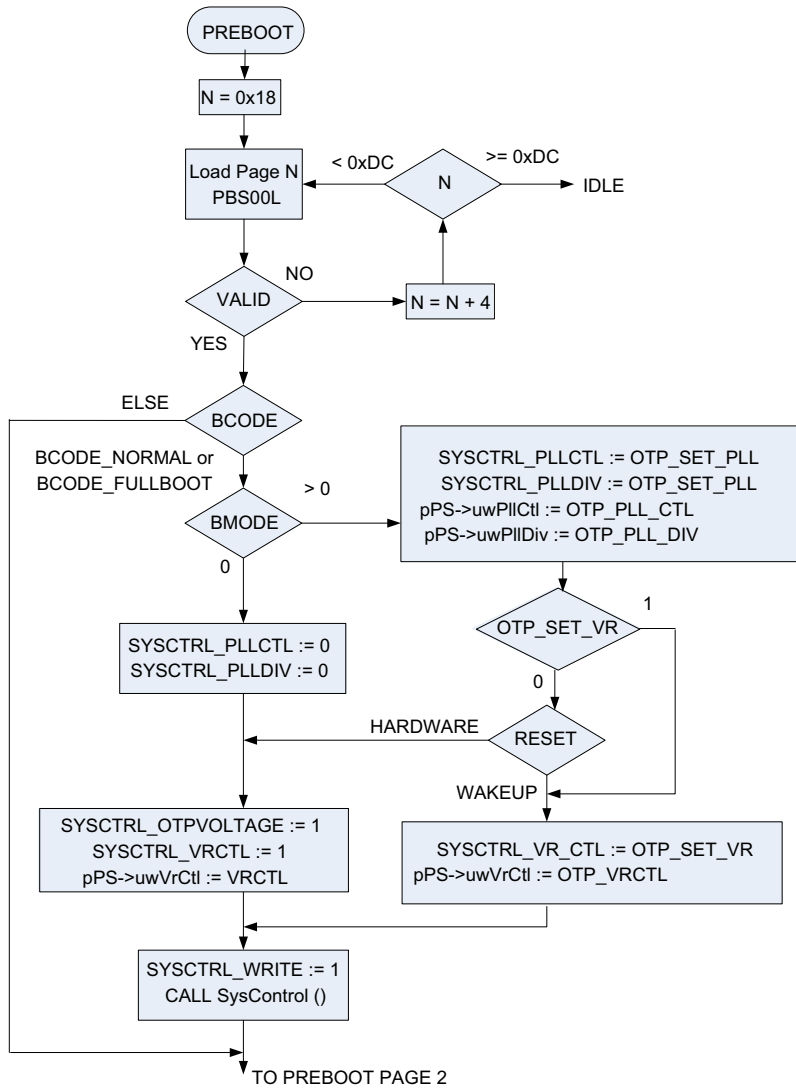


Figure 17-2. Preboot Flow 1 of 2

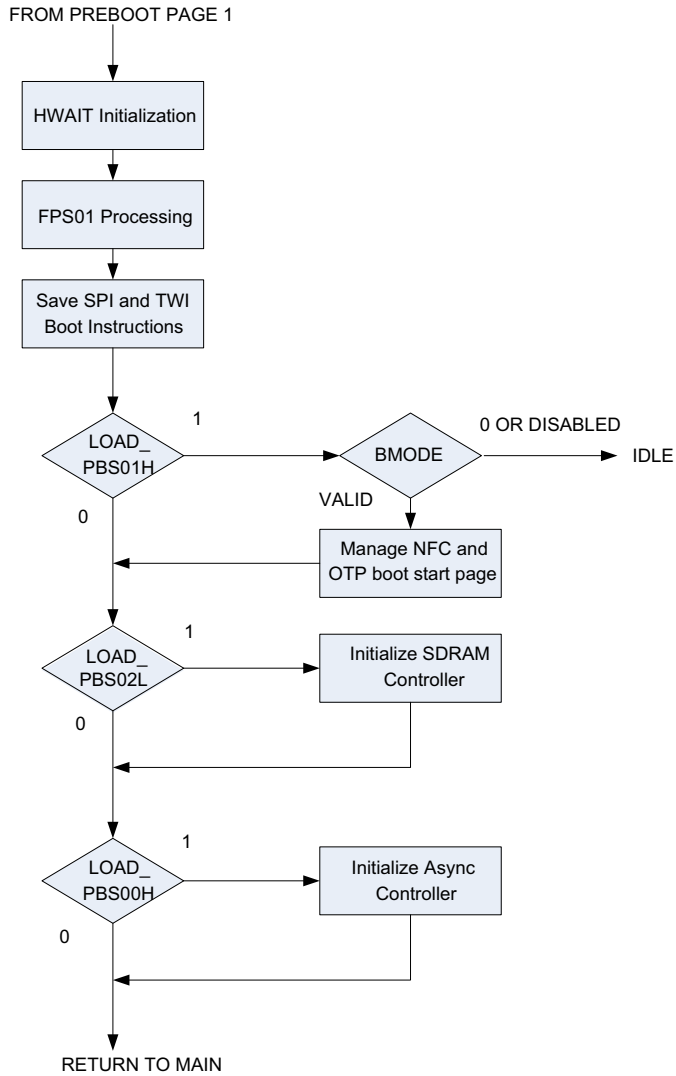


Figure 17-3. Preboot Flow 2 of 2

Customizing Booting Options

The OTP pages accessible by the preboot mechanism can also be used to customize some of the booting options. For example:

- TWI master boot mode operating frequency
- SPI master boot mode operating frequency
- SPI master boot mode read operation mode
- Start page for OTP boot mode
- HWAIT signal behavior
- Disabling of unwanted boot modes

In TWI master boot mode, the `OTP_TWI_PRESCALE` and `OTP_TWI_CLKDIV` values in the preboot half page `PBS00L` control the respective prescale and clock divider values written to the `TWIX_CONTROL` and `TWIX_CLKDIV` registers. The table of values can be found in [“TWI Master Boot Mode” on page 17-79](#). The bit field `OTP_TWI_TYPE` controls whether one, two, three or four address bytes are used to address the I²C memory device. By default, two address bytes are used. The address bits embedded in the read command are not counted.

In SPI master boot mode, the `OTP_SPI_BAUD` register in the preboot half page `PBS00L` controls the value written to the `SPIX_BAUD` registers. By default, the clock divider value of 133 can be reduced in power-of-two steps. The table of values can be found in [“SPI Master Boot Mode” on page 17-70](#). The `OTP_SPI_BAUD` bit instructs the boot kernel to use the 0x0B SPI read command instead of the normal 0x03 read command when accessing the SPI memory device.

In OTP boot mode, the boot kernel normally assumes that the boot stream starts at OTP page 0x40L. The user can change this start page by programming the `OTP_START_PAGE` bit field in the preboot half page `PBS01H`.

The boot host wait (HWAIT) signal is available in all boot modes. If the `OTP_RESETOUT_HWAIT` bit in the preboot half page `PBS00L` is set, the boot kernel does not toggle HWAIT. Rather, it simply drives it to simulate a reset output signal.

If the `OTP_ALTERNATE_HWAIT` bit in the same half page is set, the alternate GPIO pin (HWAITA) is used instead of HWAIT.

If safety or security of an application is impacted by the existence of certain boot modes, the boot mode disable bits in preboot half page `PBS01H` can be used to disable unwanted boot modes. For example, setting the `BMODE15_DIS` and `BMODE14_DIS` bits disables the two Host DMA boot modes. If a disabled boot mode is chosen by the `BMODE` pins, the boot kernel goes into a safe idle state and stops processing. The half page `PBS01H` is only loaded when the `OTP_LOAD_PBS01H` bit in the `PBS00L` page is set.

Customizing the Asynchronous Port


The preboot half page `PBS00H` contains instructions to customize the asynchronous portion of the EBIU controller. This half page is only loaded and processed when the `OTP_LOAD_PBS00H` bit in the `PBS00L` half page is programmed to a 1.

The `OTP_EBIU_AMG` field is copied into the `EBIU_AMGCTL` register. While the lower bit controls the `CLKOUT` signal, the upper three `AMBEN` bits control which of the four asynchronous banks are enabled. For the FIFO boot mode, the three `AMBEN` bits are overruled and are all always set.

The preboot routine analyzes the three `AMBEN` bits and initializes the 16-bit portions (this routine is similar to the enabled banks in the `EBIU_AMBCTL0` and `EBIU_AMBCTL1` registers) with the value provided in the 16-bit `OTP_EBIU_AMBCTL` field. In this way, the bus timing of the synchronous port can be customized prior to the boot process.

Preboot

Half page PBS00H also contains the 16-bit OTP_EBIU_FCTL field which is copied directly to the EBIU_FCTL register.

 Make sure that all bits in the OTP_EBIU_FCTL field that correspond with reserved bits in the EBIU_FCTL register are written with 0s.

The preboot routine ensures that a zero value is never written to the BCLK bit field.

The 8-bit value OTP_EBIU_MODE field is copied to the lower eight bits of the EBIU_MODE register. If any of the four memory banks has its BxMODE field set to a value of three, a device initialization sequence can be performed. All four banks are temporarily put into the asynchronous flash mode, and the four-bit OTP_EBIU_DEVSEQ field controls which sequence is performed. The 16-bit OTP_EBIU_DEVCFG word is part of the initialization sequence. Such a sequence is usually required to activate the bursting mode on multi-mode memories. Currently, the vendor-specific sequences shown in [Table 17-3](#) are supported.

Table 17-3. Burst NOR Flash Initialization Sequences

OTP_EBIU_DEVSEQ=2	OTP_EBIU_DEVSEQ=4	OTP_EBIU_DEVSEQ=6
Atmel, Intel, ST (16-bit)	Spansion (16-bit)	Samsung (16-bit)
w[OTP_EBIU_DEVCFG<<1] = 0x60	w[0x555<<1] = 0xAA	w[0x555<<1] = 0xAA
w[OTP_EBIU_DEVCFG<<1] = 0x03	w[0x2AA<<1] = 0x55	w[0x2AA<<1] = 0x55
w[0] = 0xFF	w[0x555<<1] = 0xD0	w[(OTP_EBIU_DEVCFG[6:0]<<12 0x555)<<1] = 0xC0
	w[0x000] = OTP_EBIU_DEVCFG	w[0x000] = 0xF0
	w[0x000] = 0xF0	

Whenever the `PBS00H` half page is processed, all EBIU signals that belong to the interface are enabled at the port muxing level. This includes the address pins on port H and port J, as well as the `ARDY` and bus request signals on port J. In flash boot mode, these signals are activated regardless of the OTP programming.

Finally, the 8-bit `OTP_NFC_CTL` field in the `PBS01H` half page initializes the eight least significant bits of the `NFC_CTL` register.

Customizing the Synchronous Port

Since many Blackfin applications require data and/or instruction code to be loaded into the SDRAM memory at boot time, the SDRAM controller must be initialized beforehand. This can be done by using either the [“Initialization Code” on page 17-39](#) or the preboot mechanism described here. For the SDRAM boot mode, only the preboot mechanism is valid.

If the `OTP_LOAD_PBS02L` and `OTP_LOAD_PBS02H` bits in the `PBS00L` half page have been programmed to a 1, then the two preboot half pages `PBS02L` and `PBS02H` are also loaded and processed. These half pages initialize the SDRAM controller.

Half page `PBS02L` contains the two 32-bit values `OTP_EBIU_DDRCTL0` and `OTP_EBIU_DDRCTL1` that are directly copied into the `EBIU_DDRCTL0` and `EBIU_DDRCTL1` SDRAM control registers, respectively.

Half page `PBS02H` contains the three 16-bit values `OTP_EBIU_DDRCTL2L`, `OTP_EBIU_DDRCTL3L` and `OTP_EBIU_DDRQUEL` that are copied into the lower 16 bits of the respective `EBIU_DDRCTL2`, `EBIU_DDRCTL3` and `EBIU_DDRQUE` registers.

Basic Booting Process

Once the preboot routine returns, the boot kernel residing in the on-chip boot ROM starts processing the boot stream. The boot stream is either read from memory or received from a host processor. A boot stream represents the application data and is formatted in a special manner. The application data is segmented into multiple blocks of data. Each block begins with a block header. The header contains control words such as the destination address and data length information.

As [Figure 17-4](#) illustrates, the VisualDSP++ tools suite features a loader utility (`elfloader.exe`). The loader utility parses the input executable file (`.DXE`), segments the application data into multiple blocks, and creates the header information for each block. The output is stored in a loader file (`.LDR`). The loader file contains the boot stream and is made available to

hardware by programming or burning it into non-volatile external memory. Refer to the *VisualDSP++ Loader Manual* for information on switches for loader files.

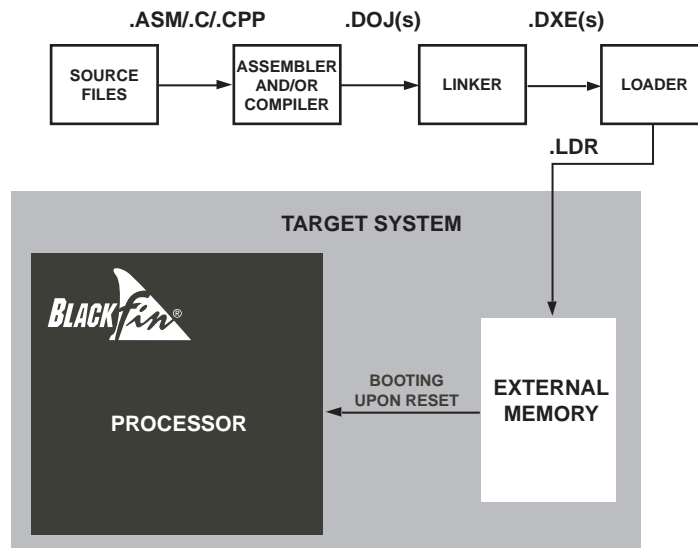


Figure 17-4. Project Flow for a Standalone System

Figure 17-5 shows the parallel or serial boot stream contained in a flash memory device. In host boot scenarios, the non-volatile memory more likely connects to the host processor rather than directly to the Blackfin processor. After reset, the headers are read and parsed by the on-chip boot

Basic Booting Process

ROM, and processed block-by-block. Payload data is copied to destination addresses, either in on-chip L1 and L2 memory or off-chip SRAM/SDRAM.

i Booting into scratchpad memory (0xFFB0 0000–0xFFB0 0FFF) is not supported. If booting to scratchpad memory is attempted, the processor hangs within the on-chip boot ROM. Similarly, booting into the upper 16 bytes of L1 data bank A (0xFF80 7FF0–0xFF80 7FFF by default) is not supported. These memory locations are used by the boot kernel for intermediate storage of block header information. These memory regions cannot be initialized at boot time. After booting, they can be used by the application during run time.

When the `BFLAG_INDIRECT` flag for any block is set, as in TWI boot modes, the boot kernel uses another memory block in L1 data bank B (by default, 0xFF90 7E00–0xFF90 7FFF) for intermediate data storage. To avoid conflicts, the VisualDSP++ `elfloader` utility ensures this region is booted last.

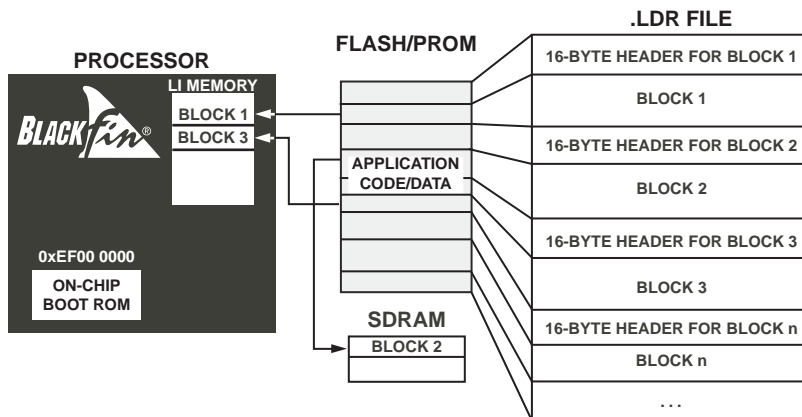


Figure 17-5. Booting Process

The entire source code of the boot ROM is shipped with the VisualDSP++ tools installation. Refer to the source code for any additional questions not covered in this manual. Note that minor maintenance work may be done to the content of the boot ROM when silicon is updated.

Block Headers

A boot stream consists of multiple boot blocks, see [Figure 17-6](#). Every block is headed by a 16-byte block header. However every block does not necessarily have a payload, as shown in [Figure 17-32 on page 17-93](#).

The 16 bytes are functionally grouped into four 32-bit words, the block code, the target address, the byte count, and the argument fields.

Basic Booting Process

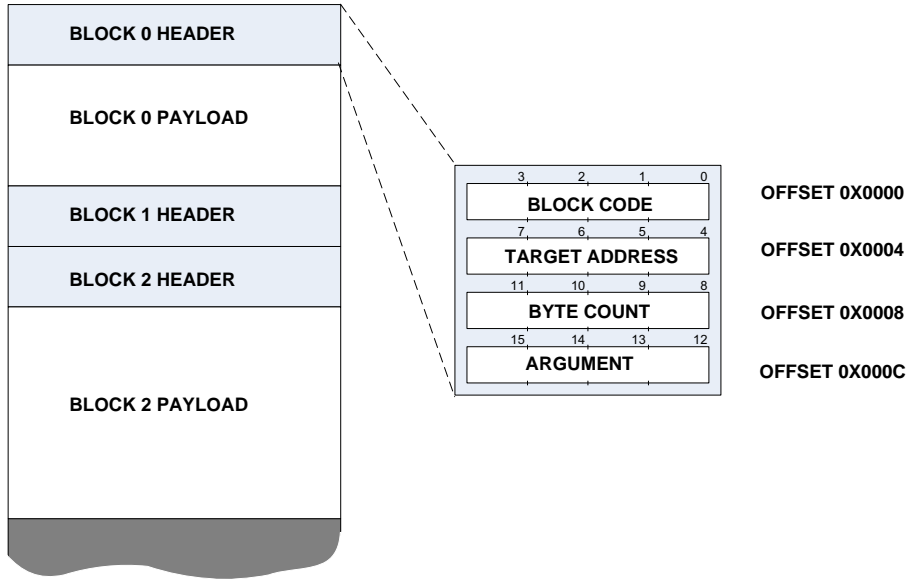


Figure 17-6. Boot Stream Headers

Block Code

The first 32-bit word is the block code field. See [Figure 17-7](#).

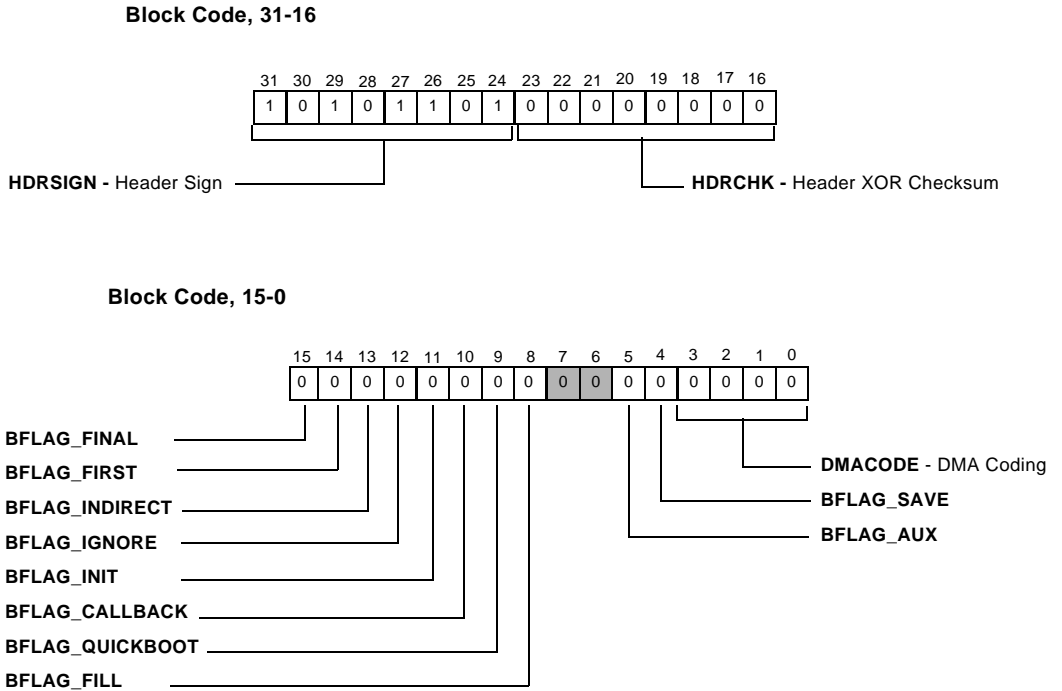


Figure 17-7. Block Code, 31-0

The DMA code (**DMACODE**) field instructs the boot kernel whether to use 8-bit, 16-bit or 32-bit DMA and how to program the source modifier of a memory DMA. Particularly in case of memory boot modes, this field is interrogated by the boot kernel to differentiate the 8-bit, 16-bit, and 32-bit cases.

Basic Booting Process

The boot kernel tests this field only on the first block and ignores the field in further blocks (See [Table 17-4](#)).

Table 17-4. Bus and DMA Width Coding

DMA Code	DMA Width	Source DMA Modify	Application
0	reserved ¹		
1	8-bit	1	Default 8-bit boot from 8-bit source ²
2	8-bit	2	Zero-padded 8-bit boot from 16-bit EBIU
3	8-bit	4	Zero-padded 8-bit boot from 32-bit EBIU ³
4	8-bit	8	Zero-padded 8-bit boot from 64-bit EBIU ⁴
5	8-bit	16	Zero-padded 8-bit boot from 128-bit EBIU ⁴
6	16-bit	2	Default 16-bit boot from 16-bit source ⁵
7	16-bit	4	Zero-padded 16-bit boot from 32-bit EBIU ³
8	16-bit	8	Zero-padded 16-bit boot from 64-bit EBIU ⁴
9	16-bit	16	Zero-padded 16-bit boot from 128-bit EBIU ⁴
10	32-bit	4	Default 32-bit boot from 32-bit source ^{3,5}
11	32-bit	8	Zero-padded 32-bit boot from 64-bit EBIU ⁴
12	32-bit	16	Zero-padded 32-bit boot from 128-bit EBIU ⁴
13	64-bit	8	Default 64-bit boot from 64-bit source ⁴
14	64-bit	16	Zero-padded 64-bit boot from 128-bit EBIU ⁴
15	128-bit	16	Default 128-bit boot from 128-bit source ⁴

- 1 Reserved to differentiate from ADSP-BF53x boot streams.
- 2 Used by all byte-wise serial boot modes.
- 3 Applicable only for boot from SDRAM and boot from internal ROM.
- 4 Not supported by ADSP-BF54x Blackfin products.
- 5 This is the only code supported by NAND flash boot.

Table 17-5. Block Flags

Bit	Name	Description
4	BFLAG_SAVE	Saves the memory of this block to off-chip memory in case of power failure or a hibernate request. This flag is not used by the on-chip boot kernel.
5	BFLAG_AUX	Nests special block types as required by special-purpose second-stage loaders. This flag is not used by the on-chip boot kernel.
6	Reserved	
7	Reserved	
8	BFLAG_FILL	Tells the boot kernel to not process any payload data. Instead the target memory (specified by the TARGET ADDRESS and BYTE COUNT fields) is filled with the 32-bit value provided by the ARGUMENT word. The fill operation is always performed by 32-bit DMA. Therefore target address and byte count must be divisible by four.
9	BFLAG_QUICKBOOT	Processes the block for full boot only. Does not process this block for a quick boot (warm boot).
10	BFLAG_CALLBACK	Calls a subfunction that may reside in on-chip or off-chip ROM or is loaded by an initcode in advance. Often used with the BFLAG_INDIRECT switch. If BFLAG_CALLBACK is set for any block, an initcode must register the callback function first. The function is called when either the entire block is loaded or the intermediate storage memory is full. The callback function can do advanced processing such as CRC checksum.

Basic Booting Process

Table 17-5. Block Flags (Cont'd)

Bit	Name	Description
11	BFLAG_INIT	This flag causes the boot kernel to issue a CALL instruction to the target address of the boot block after the entire block is loaded. The initcode should return by an RTS instruction. It may or may not be overwritten by application data later in the boot process. If the code is loaded earlier or resides in ROM, the init block can be zero sized (no payload).
12	BFLAG_IGNORE	Indicates a block that is not booted into memory. It instructs the boot kernel to skip the number of bytes of the boot stream as specified by BYTE COUNT. In master boot modes, the boot kernel simply modifies its source address pointer. In this case the BYTE COUNT value can be seen as a 32-bit two's-complement offset value to be added to the source address pointer. In slave boot modes, the boot kernel actively loads and changes the payload of the block. In slave modes the BYTE COUNT must be a positive value.
13	BFLAG_INDIRECT	Boots to an intermediate storage place, allowing for calling an optional callback function, before booting to the destination. This flag is used when the boot source does not have DMA support (TWI for example) and either the destination cannot be accessed by the core (L1 instruction SRAM) or cannot be efficiently accessed by the core (SDRAM or RAM). This flag is also used when CALLBACK requires access to data to calculate a checksum, or when performing tasks such as decryption or decompression.
14	BFLAG_FIRST	This flag, which is only set on the first block of a DXE, tells the boot kernel about the special nature of the TARGET ADDRESS and the ARGUMENT fields. The TARGET ADDRESS field holds the start address of the application. The ARGUMENT field holds the offset to the next DXE.
15	BFLAG_FINAL	This flag causes the boot kernel to pass control over to the application after the final block is processed. This flag is usually set on the last block of a DXE unless multiple DXEs are merged.

The **BFLAG_FIRST** flag must not be combined with the **BFLAG_FILL** flag. The **BFLAG_FIRST** flag may be combined with the **BFLAG_IGNORE** flag to deposit special user data at the top of the boot stream. Note the special importance of the VisualDSP++ elfloader `-readall` switch.

The header checksum (`HDRCHK`) field holds a simple XOR checksum of the other 31 bytes in the boot block header. The header signature (`HDRSGN`) byte always reads as `0xAD` and is used to verify whether the block pointer actually points to a valid block. The boot kernel jumps to the error routine if the result of an XOR operation across all 32 header bytes (including the `HDRCHK` value) differs from zero. The default error routine is a simple `IDLE;` instruction. The user can overwrite the default error handler using the `initcode` mechanism.

The `HDRSGN` byte can also be used as a boot stream version control. For the ADSP-BF54x and ADSP-BF52x Blackfin processors, the byte always reads `0xAD`. The ADSP-BF53x boot streams always read `0xFF`. The ADSP-BF561 boot streams always read `0xA0`.

Target Address

This 32-bit field holds the target address where the boot kernel loads the block payload data. When the `BFLAG_FILL` flag is set, the boot kernel fills the memory with the value stored in the argument field starting at this address. If the `BFLAG_INIT` flag is set the kernel issues a `CALL(TARGET ADDRESS)` instruction after the optional payload is loaded.


If the `BFLAG_FIRST` flag is set, the target address field contains the start address of the application to which the boot kernel jumps at the end of the boot process. This address will also be stored in the `EVT1` register. By default the VisualDSP++ elfloader utility sets this value to `0xFFA0 0000` for compatibility with other Blackfin products.

The target address should be divisible by four, because the boot kernel uses 32-bit DMA for certain operations. The target address must point to valid on-chip or off-chip memory locations. When booting to external memories, the memory controller must first be set up by either the pre-boot or the `initcode` mechanism. When booting through peripherals that do not support DMA transfers, such as TWI boot modes, the

Basic Booting Process

`BFLAG_INDIRECT` flag must be set if the target address points to L1 instruction memory. For performance reasons this is also recommended when booting to off-chip memories.

For a TWI boot the VisualDSP++ elfloader utility manages the `BFLAG_INDIRECT` flag automatically. Refer to the *VisualDSP++ Loader and Utilities* reference guide for manual control of the flag.

 Booting to scratchpad memory is not supported. The scratchpad memory functions as a stack for the boot kernel. The L1 data memory locations 0xFF80 7FF0 to 0xFF80 7FFF are used by the boot kernel and should not be overwritten by the application. The memory range used for intermediate storage as controlled by the `BFLAG_INDIRECT` switch should only be booted after the last `BFLAG_INDIRECT` bit is processed. By default the address range 0xFF90 7E00–0xFF90 7FFF is used for intermediate storage.

For normal boot operation, the target address points to RAM memory. There are however a few exceptions where the target address can point to on-chip or off-chip ROM. For example a zero-sized `BFLAG_INIT` block would instruct the boot kernel to `CALL` a subroutine residing in ROM or flash memory. This method is used to activate the CRC32 feature.

Byte Count

This 32-bit field tells the boot kernel how many bytes to process. Normally, this is the size of the payload data of a boot block. If the `BFLAG_FILL` flag is set there is no payload. In this case the byte count field uses the value in its argument field to tell the boot kernel how many bytes to process.

The byte count is a 32-bit value that should be divisible by four. Zero values are allowed in all block types. Most boot modes are based upon DMA operation which are only 16-bit words for Blackfin processors. The boot kernel may therefore start multiple DMA work units for large boot blocks.

This enables a single block to fill to zero the entire SDRAM memory, for example, resulting in compact boot streams. The `HWAIT` signal may toggle for each work unit.

If the `BFLAG_IGNORE` flag is set, the byte count is used to redirect the boot source pointer to another memory location. In master boot modes, the byte count is a two's-complement (signed long integer) value. In slave boot modes, the value must be positive.

Argument

This 32-bit field is a user variable for most block types. The value is accessible by the `initcode` or the callback routine and can therefore be used for optional instructions to these routines. When the `CRC32` feature is activated, the argument field holds the checksum over the payload of the block.

When the `BFLAG_FILL` flag is set there is no payload. The argument contains the 32-bit fill value, which is most likely a zero.

If the `BFLAG_FIRST` flag is set, the argument contains the relative next-DXE pointer for multi-DXE applications. For single-DXE applications the field points to the next free boot source address after the current DXE's boot stream.

Boot Host Wait (HWAIT) Feedback Strobe

The `HWAIT` feedback strobe is a handshake signal that is used to hold off the host device from sending further data while the boot kernel is busy.

On ADSP-BF54x processors this feature is implemented by a GPIO that is toggled by the boot kernel as required. By default the `PB11` GPIO is used for this purpose. If the `OTP_ALTERNATE_HWAIT` fuse in OTP memory page `PBS00L` is programmed, the boot kernel uses the `PH7` GPIO instead.

Basic Booting Process

The signal polarity of the H_{WAIT} strobe is programmable by an external resistor in the 10 k Ω range.

A pull-up resistor instructs the H_{WAIT} signal to be active high. In this case the host is permitted to send header and footer data when H_{WAIT} is low, but should pause while H_{WAIT} is high. This is the mode used in SPI slave boot on other Blackfin products.

Similarly, a pull-down resistor programs active-low behavior.



Note that the H_{WAIT} signal is implemented slightly differently than on ADSP-BF53x Blackfin processors. In the ADSP-BF54x processors, the meaning of the pulling resistor is inverted and H_{WAIT} is asserted by default during reset and preboot.

After preboot, the boot kernel first senses the polarity on the respective H_{WAIT} pin. Then it enables the output driver but keeps the signal in its asserted state. The signal is not released until the boot kernel is ready for data, or when a receive DMA is started. As soon as the DMA completes, H_{WAIT} becomes active again.

The boot host wait signal holds the host from booting in any slave boot mode and prevents it from being overrun with data. The H_{WAIT} signal is, however, available in all boot modes with the exception of the NAND boot mode. In some cases it is redundant to other handshake mechanisms, such as the UART \overline{RTS} signal.

In general the host device must interrogate the H_{WAIT} signal before every word that is sent. This requirement can be relaxed for boot modes using on-chip peripherals that feature larger receive FIFOs. However, the host must not rely on the DMA FIFO since its content is cleared at the end of a DMA work unit.

While the `HWAIT` signal is only used for boot purposes, it may also play a significant role after booting. In slave boot modes, for example, the host device does not necessarily know whether the Blackfin processor is in an active mode or a power-down mode. For example, the `HWAIT` signal can be used to signal when the processor is in hibernate mode.

Using `HWAIT` as `RESETOUT` Indicator

While the `HWAIT` signal is mandatory in some boot modes, it is optional in others. When not required for booting, the behavior of the `HWAIT` signal (or alternate `HWAIT` signal) can be changed by programming the `OTP_RESETOUT_HWAIT` bit in OTP page `PBS00L`.

If this bit is set, `HWAIT` does not toggle during the boot process. Rather, after page `PBS00L` is processed (and therefore the PLL has settled) the pre-boot routine first enables the `HWAIT` GPIO as an input and senses its state. Then `HWAIT` becomes an output and is driven to the invert of the state that is sensed. An external pulling resistor is required. If using a pull-up resistor, the `HWAIT` signal is driven low for the rest of the boot process (and beyond). If using a pull-down resistor, `HWAIT` is driven high.

With a pull-down resistor, this feature can be used to simulate a active-low reset output. When the processor is reset, or in hibernate, the GPIO is in a high impedance state and `HWAIT` is pulled low by the resistor. As soon as the processor recovers and has settled the PLL again, the `HWAIT` is driven high and can alert external circuits.

Boot Termination

After the successful download of the application into the bootable memory, the boot kernel passes control to the user application. By default this is performed by jumping to the vector stored in the `EVT1` register. The boot kernel provides options to execute an `RTS` instruction or a `RAISE 1` instruction instead. The default behavior can be changed by an initcode

Basic Booting Process

routine. The EVT1 register is updated by the boot kernel when processing the BFLAG_FIRST block. See “[Servicing Reset Interrupts](#)” on page 17-9 to learn how the application can take control.

Before the boot kernel passes program control to the application it does some housekeeping. Most of the registers that were used are changed back to their default state but some register values may differ for individual boot modes. DMA configuration registers and primary register control registers (UARTx_LCR, SPIx_CTL, HOST_CONTROL, etc.) are restored, while others are purposely not restored. For example SPIx_BAUD, UARTx_DLH and UARTx_DLL remain unchanged so that settings obtained during the booting process are not lost.

Single Block Boot Streams

The simplest boot stream consists of a single block header and one contiguous block of instructions. With appropriate flag instructions the boot kernel loads the block to the target address and immediately terminates by executing the loaded block.

[Table 17-6](#) shows an example of a single block boot stream header that could be loaded from any serial boot mode. It places a 256-byte block of instructions at L1 instruction SRAM address 0xFFA1 0000. The flags BFLAG_FIRST and BFLAG_FINAL are both set at the same time. Advanced flags, such as BFLAG_IGNORE, BFLAG_INIT, BFLAG_CALLBACK and BFLAG_FILL, do not make sense in this context and should not be used.

Table 17-6. Header for a Single Block Boot Stream

Field	Value	Comments
BLOCK CODE (DMACODE & 0x1)	0xAD32 0001	0xAD00 0000 XORSUM BFLAG_FINAL BFLAG_FIRST
TARGET ADDRESS	0xFFA0 0000	Start address of block and application code
BYTE COUNT	0x0000 0100	256 bytes of code
ARGUMENT	0x0000 0100	Functions as next-DXE pointer in multi-DXE boot streams

With the `BFLAG_FIRST` flag set, the `ARGUMENT` field functions as the next-DXE pointer. This is a relative pointer to the next free source address or to the next DXE start address in a multi-DXE stream.

Direct Code Execution

Applications may want to avoid long booting times and start code execution directly from 16-bit flash or SDRAM memory. This feature is called direct code execution. This is a special case of boot termination that replaces the no-boot/bypass mode in the ADSP-BF53x Blackfin processors.

An initial boot block header is needed for the processor to fetch and execute program code from the boot device as early as possible. The safety mechanisms of the block, such as the header signature and the XOR checksum, avoid unpredictable processor behavior due to the boot memory not being programmed with valid data yet. Rather than blindly executing code, the boot kernel first executes the preboot routine for system customization, then loads the first block header and checks it for consistency. If the block header is corrupted, the boot kernel goes into a safe idle state and does not start code execution.

If the initial block header checks good, the boot kernel interrogates the block flags. If the block has the `BFLAG_FINAL` flag set, the boot kernel immediately terminates and jumps directly to the address stored in the `EVT1` register. To cause the boot kernel to customize the `EVT1` register in advance, the initial blocks must also have the `BFLAG_FIRST` flag set. The target address field is then copied to the `EVT1` register. In this way, the target address field of the initial block defines the start address of the application.

For example in `BMODE = 1`, when the block header described in [Table 17-7 on page 17-38](#) is placed at address `0x2000 0000`, the boot kernel is instructed to issue a `JUMP` command to address `0x2000 0020`.

Basic Booting Process

The development tools must be instructed to link the above block to address 0x2000 0000 and the application code to address 0x2000 0020. An example shown in [“Direct Code Execution” on page 17-164](#) illustrates how this is accomplished using the VisualDSP++ tools suite.

Table 17-7. Initial Header for Direct Code Execution in BMODE = 1

Field	Value	Comments
BLOCK CODE	0xAD7B D006	0xAD00 0000 XORSUM BFLAG_FINAL BFLAG_FIRST BFLAG_IGNORE (DMACODE & 0x6)
TARGET ADDRESS	0x2000 0020	Start address of application code
BYTE COUNT	0x0000 0010	Ignores 16 bytes to provide space for control data such as version code and build data. This is optional and can be zero.
ARGUMENT	0x0000 0010	Functions as next-DXE pointer in multi-DXE boot streams

Similarly for direct code execution in the SDRAM boot mode (BMODE = 10), an initial block as shown in [Table 17-8](#) has to be linked to address 0x0000 0010.

Table 17-8. Initial Header for Direct Code Execution in BMODE = 10

Field	Value	Comments
BLOCK CODE	0xAD5B D006	0xAD000000 XORSUM BFLAG_FINAL BFLAG_FIRST BFLAG_IGNORE (DMACODE & 0x6)
TARGET ADDRESS	0x0000 0020	Start address of application code
BYTE COUNT	0x0000 0000	No bubble for control data
ARGUMENT	0x0000 0000	Functions as next-DXE pointer in multi-DXE boot streams

For multi-DXE boot streams, [Figure 17-11 on page 17-60](#) shows a linked list of initial blocks that represent different applications.

Advanced Boot Techniques

Initialization Code

Initcode routines are subroutines that the boot kernel calls during the booting process. The user can customize and speed up the booting mechanisms using this feature. Traditionally, an initcode is used to set up system PLL, bit rates, wait states and the SDRAM controller. If executed early in the boot process, the boot time can be significantly reduced.

After the payload data is loaded for a specific boot block, if the `BFLAG_INIT` flag is set, the boot kernel issues a `CALL` instruction to the `TARGET ADDRESS` of the block.

On ADSP-BF54x Blackfin processors, initcode routines follow the C language calling convention so they can be coded in C language or assembly.

The expected prototype is

```
void initcode(ADI_BOOT_DATA* pBootStruct);
```

The VisualDSP++ header files define the `ADI_BOOT_INITCODE_FUNC` type:

```
typedef void ADI_BOOT_INITCODE_FUNC (ADI_BOOT_DATA* ) ;
```

Optionally, the initcode routine can interrogate the formatting structure and customize its own behavior or even manipulate the regular boot process. A pointer to the structure is passed in the `R0` register. Assembly coders must ensure that the routine returns to the boot kernel by a terminating `RTS` instruction.

Initcodes can rely on the validity of the stack, which resides in scratchpad memory. The `ADI_BOOT_DATA` structure resides on the stack. Rules for register usage conform to the compiler conventions. See the *VisualDSP++ C/C++ Compiler and Library Manual* for more information.

Advanced Boot Techniques

In the simple case, initcodes consist of a single instruction section and are represented by a single block within the boot stream. This block has the `BFLAG_INIT` bit set.

An init block can consist of multiple sections where multiple boot blocks represent the initcode within the boot stream. Only the last block has the `BFLAG_INIT` bit set.

The VisualDSP++ elfloader utility ensures that the last of these blocks vector to the initcode entry address. The utility instructs the on-chip boot ROM to execute a `CALL` instruction to the given `TARGET ADDRESS`.

When the on-chip boot ROM detects a block with the `INIT` bit set, it boots the block into Blackfin memory and then executes it by issuing a `CALL` to its target address. For this reason, every initcode must be terminated by an `RTS` instruction to ensure that the processor vectors back to the on-chip boot ROM for the rest of the boot process.

Sometimes initcode boot blocks have no payload and the `BYTE COUNT` field is set to zero. Then the only purpose of the block may be to instruct the boot kernel to issue the `CALL` instruction.

Initcode routines can be very different in nature. They might reside in ROM or SRAM. They might be called once during the booting process or multiple times. They might be volatile and be overwritten by other boot blocks after executing, or they might be permanently available after boot time. The boot kernel has no knowledge of the nature of initcodes and has no restrictions in this regard. Refer to the *VisualDSP++ Loader and Utilities Manual* for how this feature is supported by the tools chain.

It is the user's responsibility to ensure that all code and data sections that are required by the initcode are present in memory by the time the initcode executes. Special attention is required if initcodes are written in C or C++ language. Ensure that the initcode does not contain calls to the run time libraries. Do not assume that parts of the run time environment, such as the heap are fully functional. Ensure that all run time components are loaded and initialized before the initcode executes.

The VisualDSP++ elfloader utility provides two different mechanisms to support the initcode feature.

- The `-init initcode.dxe` command line switch
- The `-initcall address/symbol` command line switch

If enabled by the VisualDSP++ elfloader `-init initcode.DXE` command line switch, the initcode is added to the beginning of the boot stream. Here, `initcode.DXE` refers to the user-provided custom initialization executable—a separate VisualDSP++ project. [Figure 17-8](#) shows a boot stream example that performs the following steps.

1. Boot initcode into L1 memory.
2. Execute initcode.
3. Initcode initializes the SDRAM controller and returns.
4. Overwrite initcode with final application code.
5. Boot data/code into SDRAM.
6. Continue program execution with block n.

Advanced Boot Techniques

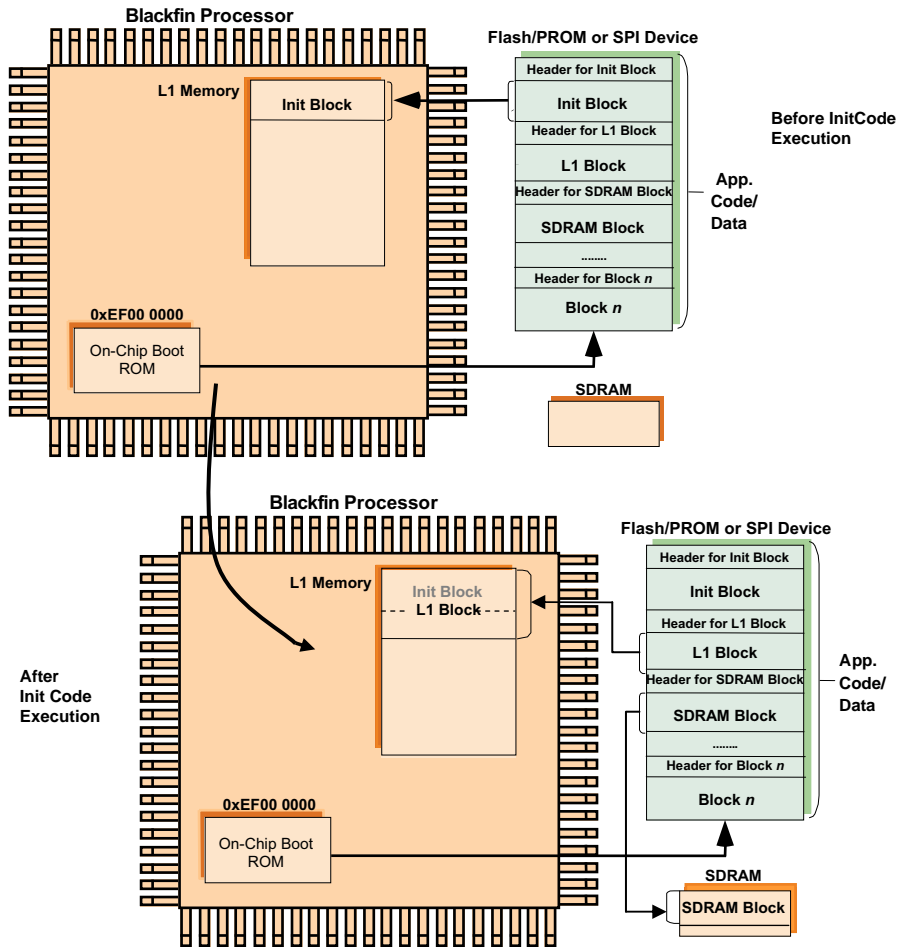


Figure 17-8. Initialization Code Execution/Boot

Although `initcode.DXE` files are built as VisualDSP++ projects, they differ from standard projects. Initcodes provide only a callable sub-function, so they look more like a library than an application. Nevertheless, unlike library files (`.DLB` file extension), the symbol addresses have already been resolved by the linker.

An initcode is always a heading for the regular application code. Consequently whether the initcode consists of one or multiple blocks, it is not terminated by a `BFLAG_FINAL` bit indicator—this would cause the boot ROM to terminate the boot process.

It is advantageous to have a clear separation between the initcode and the application by using the `-init` switch. If this separation is not needed, the `elfloader -initcall` command-line switch might be preferred. It enables fractions of the application code to be traded as initcode during the boot process. See the *VisualDSP++ Loader and Utilities Manual* for further details.

Initcode examples are shown in [“Programming Examples” on page 17-154](#).

Quick Boot

In some booting scenarios, not all memories need to be re-initialized. For example in a wake-up from hibernate state, off-chip SRAM might not be impacted if it was powered while the processor was in hibernate state. Dynamic RAM might also not be impacted if it was put into self-refresh mode before the processor powered down.

Advanced Boot Techniques

The ADSP-BF54x processor's boot kernel can conditionally process boot blocks. The normal scenario is all boot, the shortened version is quick boot. It relies on the following primitives.

- The `SYSCR` register is read to determine what kind of boot is expected from the boot kernel. The `WURESET` bit is compared against other reset bits to distinguish between cold boot and warm boot situations and to identify wake-up from hibernate situations. See [Figure 17-40 on page 17-110](#) for examples.
- The `BCODE` bit field in the `SYSCR` register can overrule the native decision of the boot kernel for a software boot. See the flowchart in [Figure 17-1 on page 17-8](#).
- The `BFLAG_WAKEUP` bit in the `dFlag` word of the `ADI_BOOT_DATA` structure indicates that the final decision was to perform a quick boot. If the boot kernel is called from the application, then the application can control the boot kernel behavior by setting the `BFLAG_WAKEUP` flag accordingly. See the `dFlags` variable on [Figure 17-55 on page 17-132](#).
- The `BFLAG_QUICKBOOT` flag in the block code word of the block header controls whether the current block is ignored for quick boot.

If both the global `BFLAG_WAKEUP` and the block-specific `BFLAG_QUICKBOOT` flags are set, the boot kernel ignores those blocks. But since the `BFLAG_INIT`, `BFLAG_CALLBACK`, `BFLAG_FINAL`, and `BFLAG_AUX` flags are internally cleared and the `BFLAG_IGNORE` flag is toggled, through double negation, the “ignore the ignore block” command instructs the boot kernel to process the block.

Although the `BFLAG_INIT` flag is suppressed in quick boot, the user may not want to combine the `BFLAG_INIT` flag with the `BFLAG_QUICKBOOT` flag. The initialization code can interrogate the `BFLAG_WAKEUP` flag and execute conditional instructions. For more information, see [“Quickboot With Restore From SDRAM” on page 17-162](#).

Indirect Booting

The ADSP-BF54x processor's boot kernel provides a control mechanism to let blocks either boot directly to their final destination or load to an intermediate storage place, then copy the data to the final destination in a second step. This feature is motivated by the following requirements.

- Some boot modes such as TWI modes do not use DMA. They load data by core instruction. The core cannot access some memories directly (for example L1 instruction SRAM), or is less efficient than the DMA in accessing some memories (for example, external SDRAM).
- In some advanced booting scenarios, the core needs to access the boot data during the booting process, for example in processing de-compression, decryption and checksum algorithms at boot time. The indirect booting option helps speed-up and simplify such scenarios. Software accesses off-chip memory less efficiently and cannot access data directly if it resides in L1 instruction SRAM.

Indirect booting is not a global setting. Every boot block can control its own processing by the `BFLAG_INDIRECT` flag in the block header.

In general a boot block may not fit into the temporary storage memory so the boot kernel processes the block in multiple steps. The larger the temporary buffer, the faster the boot process. By default the L1 data memory region between `0xFF907E00` and `0xFF907FFF` is used for intermediate storage. Initialization code can alter this region by modifying the `pTempBuffer` and `dTempByteCount` variables in the `ADI_BOOT_DATA` structure. The default region is at the upper end of a physical memory block. When increasing the `dTempByteCount` value, `pTempBuffer` also has to change.

Callback Routines

Callback routines, like initialization codes, are user-defined subroutines called by the boot kernel at boot time. The `BFLAG_CALLBACK` flag in the block header controls whether the callback routine is called for a specific block.

There are several differences between initcodes and callback routines. While the `BFLAG_INIT` flag causes the boot kernel to issue a `CALL` instruction to the `TARGET ADDRESS` of the specific boot block, the `BFLAG_CALLBACK` flag causes the boot kernel to issue a `CALL` instruction to the address held by the `pCallbackFunction` pointer in the `ADI_BOOT_DATA` structure. While a boot stream can have multiple individual initcodes, it can have just one callback routine. In the standard boot scenario, the callback routine has to be registered by an initcode prior to the first block that has the `BFLAG_CALLBACK` flag set.

The purpose of the callback routine is to apply standard processing to the block data. Typically, callback routines contain checksum, decryption, decompression, or hash algorithms. Checksum or hash words can be passed through the block header `ARGUMENT` field.

Since callback routines require access to the payload data of the boot blocks, the block data must be loaded before it can be processed. Unlike initcodes, a callback usually resides permanently in memory. If the block is loaded to L1 instruction memory or off-chip memory, the `BFLAG_CALLBACK` flag is likely combined with the `BFLAG_INDIRECT` bit. The boot kernel performs these steps in the following order.

1. Data is loaded into the temporary buffer defined by the `pTempBuffer` variable.
2. The `CALL` to the `pCallbackFunction` is issued.
3. After the callback routine returns, the memory DMA copies data to the destination.

If a block does not fit into the temporary buffer, for example when the `BLOCK_COUNT` is greater than the `dTempByteCount` variable, the three steps are executed multiple times until all payload data is loaded and processed. The boot kernel passes the parameter `dCbFlags` to the callback routine to tell it that it is being invoked the first or the last time for a specific block. To store intermediate results across multiple calls the callback routine can use the `uwUserShort` and `dUserLong` variables in the `ADI_BOOT_DATA` structure.

Callback routines meet C language calling conventions for subroutines. The prototype is as follows.

```
s32 CallbackFunction (ADI_BOOT_DATA* pBootStruct,  
ADI_BOOT_BUFFER* pCallbackStruct, s32 dCbFlags);
```

The VisualDSP++ header file defines the `ADI_BOOT_CALLBACK_FUNC` type the following way: `typedef s32 ADI_BOOT_CALLBACK_FUNC (ADI_BOOT_DATA*, ADI_BOOT_BUFFER*, s32) ;`

The `pBootStruct` argument is passed in `R0` and points to the `ADI_BOOT_DATA` structure used by the boot kernel. These are handled by the `pTempBuffer` and `dTempByteCount` variables as well as the `pHeader` pointer to the `ARGUMENT` field. The callback routine may process the block further by modifying the `pTempBuffer` and `dTempByteCount` variables.

The `pCallbackStruct` structure passed in `R1` provides the address and length of the data buffer. When the `BFLAG_INDIRECT` flag is not set, the `pCallbackStruct` contains the target address and byte count of the boot block. If the `BFLAG_INDIRECT` flag is set, the `pCallbackStruct` contains a copy of the `pTempBuffer`. Depending on the size of the boot block and processing progress, the byte count provided by `pCallbackStruct` equals either `dTempByteCount` or the remainder of the byte count.

When the `BFLAG_INDIRECT` flag is set along with the `BFLAG_CALLBACK` flag, memory DMA is invoked by the boot kernel after the callback routine returns. This memory DMA relies on the `pCallbackStruct` structure not the global `pTempBuffer` and `dTempByteCount` variables.

Advanced Boot Techniques

The callback routine can control the source of the memory DMA by altering the content of the `pCallbackStruct` structure, as may be required if the callback routine performs data manipulation such as decompression.

The `dCbFlags` parameter passed in `R2` tells the callback routine whether it is invoked the first time (`CBFLAG_FIRST`) or whether it is called the last time (`CBFLAG_FINAL`) for a specific block. The `CBFLAG_DIRECT` flag indicates that the `BFLAG_INDIRECT` bit is not active and so that the callback routine will only be called once per block. When the `CBFLAG_DIRECT` flag is set, the `CBFLAG_FIRST` and `CBFLAG_FINAL` flags are also set.

```
#define CBFLAG_FINAL      0x0008
#define CBFLAG_FIRST     0x0004
#define CBFLAG_DIRECT    0x0001
```

A callback routine also has a boolean return parameter in register `R0`. If the return value is non-zero, the subsequent memory DMA does not execute. When the `CBFLAG_DIRECT` flag is set, the return value has no effect.

Error Handler

While the default handler simply puts the processor into idle mode, an `initcode` routine can overwrite this pointer to create a customized error handler. The expected prototype is

```
void ErrorFunction (ADI_BOOT_DATA* pBootStruct, void
*pFailingAddress);
```

Use an `initcode` to write the entry address of the error routine to the `pErrorFunction` pointer in the `ADI_BOOT_DATA` structure. The error handler has access to the boot structure and receives the instruction address that triggered the error.

CRC Checksum Calculation

The ADSP-BF54x Blackfin processors provide an `initcode` and a callback routine in ROM that can be used for CRC32 checksum generation during boot time. The checksum routine only verifies the payload data of the blocks. The block headers are already protected by the native XOR checksum mechanism.

Before boot blocks can be tagged with the `BFLAG_CALLBACK` flag to enable checksum calculation on the blocks, the boot stream must contain an `initcode` block with no payload data and with the CRC32 polynomial in the block header `ARGUMENT` word.

The `initcode` registers a proper CRC32 wrapper to the `pCallbackFunction` pointer. The registration principle is similar to the XOR checksum example shown in [“Programming Examples” on page 17-154](#).

Load Functions

With the exception of the Host DMA boot modes, all boot modes are processed by a common boot kernel algorithm. The major customization is done by a subroutine that must be registered to the `pLoadFunction` pointer in the `ADI_BOOT_DATA` structure. Its simple prototype is as follows.

```
void LoadFunction (ADI_BOOT_DATA* pBootStruct);
```

The VisualDSP++ header files define the following type:

```
typedef void ADI_BOOT_LOAD_FUNC (ADI_BOOT_DATA* ) ;
```

For a few scenarios some of the flags in the `dFlags` word of the `ADI_BOOT_DATA` structure, such as `BFLAG_PERIPHERAL` and `BFLAG_SLAVE`, slightly modify the boot kernel algorithm.

Advanced Boot Techniques

The boot ROM contains several load functions. One performs a memory DMA for flash boot, another performs a peripheral DMA, and another loads data from the TWI port through a polling operation. The first is reused for fill operation and indirect booting as well.

In second-stage boot schemes, the user can create customized load functions or reuse the originals and modify the `pDmaControlRegister`, `pControlRegister` and `dControlValue` values in the `ADI_BOOT_DATA` structure. The `pDmaControlRegister` points to the `DMAX_CONFIG` or `MDMA_Dx_CONFIG` register. When the `BFLAG_SLAVE` flag is not set, the `pControlRegister` and `dControlValue` variables instruct the peripheral DMA routine to write the control value to the control register every time the DMA is started.

Load functions written by users must meet the following requirements.

- Protect against `dByteCount` values of zero.
- Multiple DMA work units are required if the `dByteCount` value is greater than 65536.
- The `pSource` and `pDestination` pointers must be properly updated.

In slave boot modes, the boot kernel uses the address of the `dArgument` field in the `pHeader` block as the destination for the required dummy DMAs when payload data is consumed from `BFLAG_IGNORE` blocks. If the load function requires access to the block's `ARGUMENT` word, it should be read early in the function.

Calling the Boot Kernel at Run Time

The boot kernel's primary purpose is to boot data to memory after power-up and reset cycles. However some of the routines used by the boot kernel might be of general value to the application. The boot ROM supports reuse of these routines as C-callable subroutines. Programs such as second-stage boot kernels, boot managers, and firmware update tools may

call the function in the ROM at run time. This could load entirely different applications or a fraction of an application, such as a code overlay or a coefficient array.

To call these boot kernel subroutines, the boot ROM provides an API at address 0xEF00 0000 in the form of a jump table.

When calling functions in the boot ROM, the user must ensure the presence of a valid stack following C language conventions. See the VisualDSP++ Compiler documentation for details.

Debugging the Boot Process

If the boot process fails, very little information can be gained by watching the chip from outside. In master boot modes, the interface signals can be observed. In slave boot modes only the `HWAIT` or the `RTS` signals tell about the progress of the boot process.

However, by using the emulator, there are many possibilities in debugging the boot process. The entire source code of the boot kernel is provided with the VisualDSP++ installation. This includes the project executable (DXE) file. The `LOAD SYMBOLS` feature of the VisualDSP++ IDDE helps to navigate the program. Note that the content of the ROM might differ between silicon revisions. Hardware breakpoints and single-stepping capabilities are also available. Since the content of the L1 instruction ROM cannot be read out by the emulator (as this ROM is not supported by the `ITEST` feature), these instructions are not displayed in the disassembly window.

Advanced Boot Techniques

Table 17-9 shows a couple of program symbols that are of interest.

Table 17-9. Boot Kernel Symbols for Debug

Symbol	Comment
<code>_bootrom.assert.default</code>	If the program counter halts at the IDLE instruction at the <code>_bootrom.assert.default</code> address, either the boot kernel or the preboot has detected an error condition and will not continue the boot process. A misformatted boot stream, checksum errors, or invalid PBS settings are the most likely causes of such an error. The RETS register points to the failing routine. When stepping a couple of instructions further, there is a way to ignore the error and to continue the boot process by clearing the <code>>ASTAT</code> register while the emulator steps over the subsequent <code>IF CC JUMP 0</code> instruction.
<code>_bootrom.bootmenu</code>	If the emulator hits a hardware breakpoint at the <code>_bootrom.bootmenu</code> address, this indicates that the preboot returned properly. Otherwise the program may hang during preboot due to improper PBS settings or invalid boot modes.
<code>_bootrom.bootkernel.entry</code>	If the emulator hits a hardware breakpoint at the <code>_bootrom.bootkernel.entry</code> label, this indicates that device detection or autobaud returned properly.
<code>_bootrom.bootkernel.breakpoint</code>	This is a good address to place a hardware breakpoint. When hit the boot kernel has a new block header loaded in. The block header can be watched at address <code>0xFF80 7FF0</code> or wherever the <code>pHeader</code> points to.
<code>_bootrom.bootkernel.initcode</code>	All payload data of the current block is loaded by the time the program passes the <code>_bootrom.bootkernel.initcode</code> label. The boot kernel is about to interrogate the <code>BFLAG_INIT</code> flag. If set, the init code can be debugged.
<code>_bootrom.bootkernel.exit</code>	Once the boot kernel arrives at the <code>_bootrom.bootkernel.address</code> label, it detects a <code>BFLAG_FINAL</code> flag. After some housekeeping, it jumps to the <code>EVT1</code> vector.

The boot kernel also generates a circular log file in scratch pad memory. While the `pLogBuffer` and the `dLogByteCount` variables describe the location and dimension of the log buffer, the `pLogCurrent` points to the next free location in the buffer. The log file is updated whenever the kernel passes the `_bootrom.bootkernel.breakpoint` label.

At each pass, nine 32-bit words are written to the log file, as follows.

- The block code word (`dBlockCode`) of the block header
- The target address (`pTargetAddress`) of the block header
- The byte count (`dByteCount`) of the block header
- The argument word (`dArgument`) of the block header
- The source pointer (`pSource`) of the boot stream
- The block count (`dBlockCount`)
- An internal copy of the `dBlockCode` word OR'ed with `dFlags`
- The content of the `SEQSTAT` register
- A `0xFFFF FFFA (-6)` constant

The ninth word is overwritten by the next entry set, so that `0xFFFF FFFA` always marks the last entry in the log file.

Most of the data structures used by the boot kernel reside on the stack in scratchpad memory. While executing the boot kernel routine (excluding subroutines), the `P5` points to the `ADI_BOOT_DATA` structure. Type “`(ADI_BOOT_DATA*) $P5`” in the VisualDSP++ expression window to see the structure content.

Boot Management

Blackfin processor hardware platforms may be required to run different software at different times. An example might be a system with at least one application and one in-the-field firmware upgrade utility. Other systems may have multiple applications, one starting then terminating, to be replaced by another application. Conditional booting is called boot management. Some applications may self-manage their booting rules, while others may have a separate application that controls the process, namely a boot manager.

In a master boot mode where the on-chip boot kernel loads the boot stream from memory, the boot manager is a piece of Blackfin software which decides at run time what application is booted next. This may simply be based on the state of a GPIO input pin interrogated by the boot manager, or it may be the conclusion of complex system behavior.

Slave boot scenarios are different from master boot scenarios. In slave boot modes, the host masters boot management by setting the Blackfin processor to reset and then applying alternate boot data. Optionally, the host could alter the `BMODE` configuration pins, resulting in little impact to the Blackfin processor since the intelligence is provided by the host device.

Booting a Different Application

The boot ROM provides a set of user-callable functions that help to boot a new application (or a fraction of an application). Usually there is no need for the boot manager to deal with the format details of the boot stream.

These functions are:

- `BFROM_MEMBOOT` discussed in “Flash Boot Modes” on page 17-64 and “SDRAM Boot Mode” on page 17-69
- `BFROM_TWIBOOT` discussed in “TWI Master Boot Mode” on page 17-79
- `BFROM_SPIBOOT` discussed in “SPI Master Boot Mode” on page 17-70

The user application, the boot manager application, or an `initcode` can call these functions to load the requested boot data. Using the `BFLAG_RETURN` flag the user can control whether the routine simply returns to the calling function or executes the loaded application immediately.

These ROM functions expect the start address of the requested boot stream as an argument. For `BFROM_MEMBOOT`, this is a Blackfin memory address, for `BFROM_TWIBOOT` and `BFROM_SPIBOOT` it is a serial address. The SPI function can also accept the code for the GPIO pin that controls the device select strobe of the SPI memory.

Multi-DXE Boot Streams

If the start addresses of all the boot streams are predefined, the boot manager needs only to call the ROM functions directly. However since the addresses tend to vary from build to build they may have to be calculated at run time.

In the world of the VisualDSP++ elfloader, a boot stream is always generated from a DXE file. It is therefore common to talk about multi-DXE or multi-application booting. When the elfloader utility accepts multiple DXE files on its command line, it generates a contiguous boot image by default. The second boot stream is appended immediately to the first one. Since the utility updates the `ARGUMENT` field of all `BFLAG_FIRST` blocks, the `ARGUMENT` field of a `BFLAG_FIRST` block is called next-DXE pointer (NDP).

Boot Management

The next-DXE pointer of the first DXE boot stream points relatively to the start address of the second DXE boot stream. A multi-DXE boot image can be seen as a linked list of boot streams. The next-DXE pointer of the last DXE boot stream points relatively to the next free address. This is illustrated by an example shown in the next two figures. [Figure 17-9 on page 17-57](#) shows a commented sketch as an example. [Figure 17-10 on page 17-58](#) shows a screenshot of the Blackfin loader file viewer utility for the same example. The `LdrViewer` utility is not part of the VisualDSP++ tools suite. It is a third-party freeware product available on www.dolomitics.com.

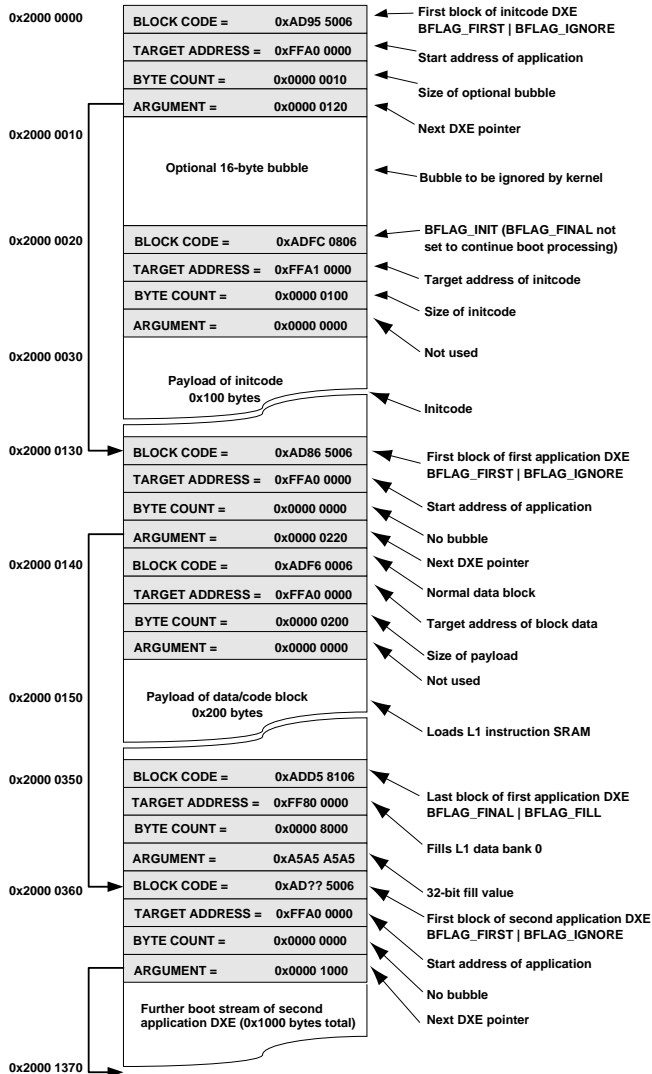


Figure 17-9. Multi-DXE Boot Stream Example for Flash Boot

Boot Management

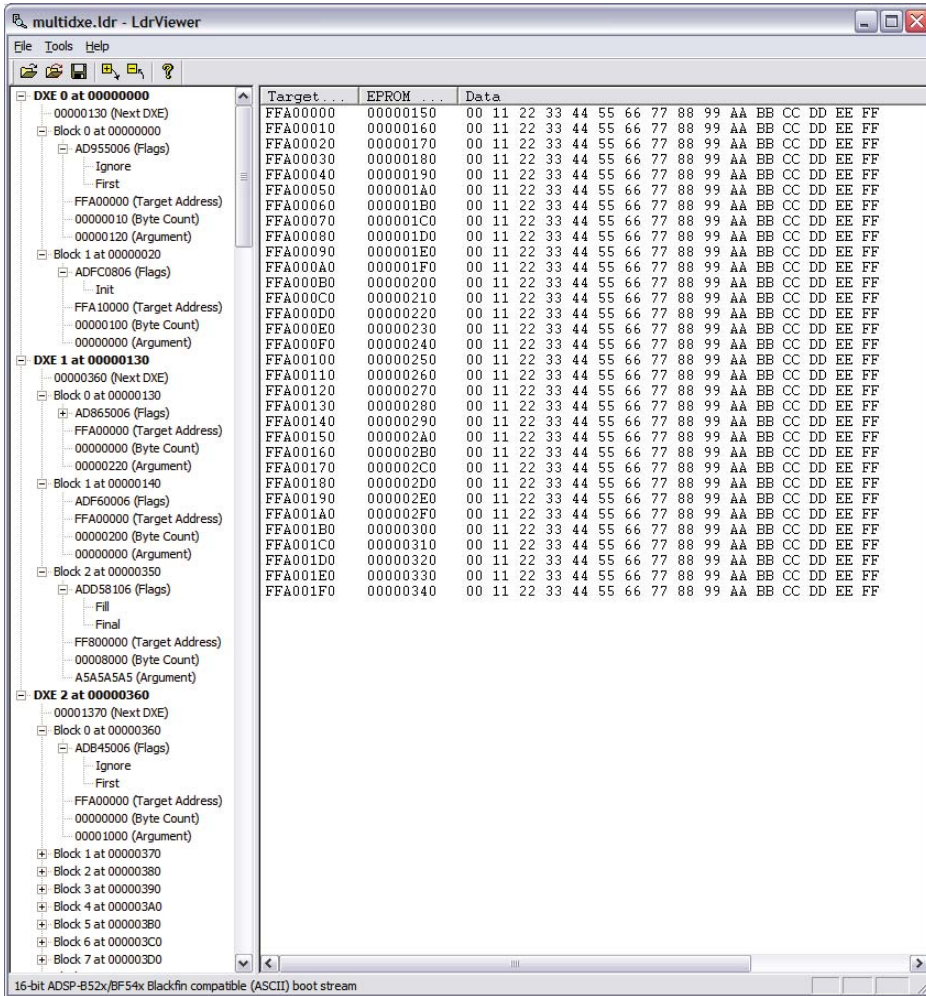


Figure 17-10. LdrViewer Screen Shot

Boot management principles are not only applicable to multi-DXE boot streams. The same scheme, as shown in [Figure 17-11 on page 17-60](#), can be applied to direct code executions of multiple applications. See [“Direct Code Execution” on page 17-37](#) for more information. The example shows a linked list of initial block headers that instruct the boot kernel to terminate immediately and to start code execution at the address provided by the target address field of the individual blocks. There is nothing in the boot ROM that prevents multi-DXE applications from mixing regular boot streams and direct code execution blocks.

Boot Management

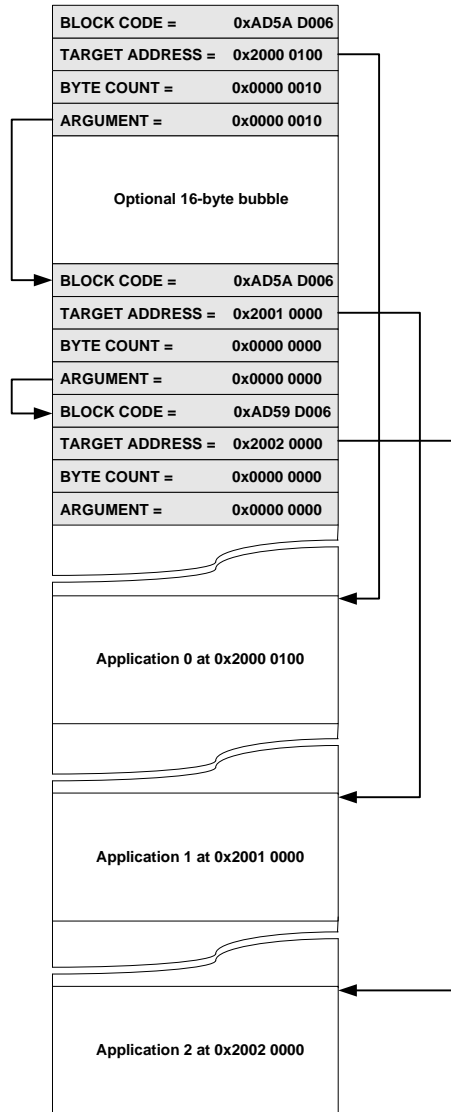


Figure 17-11. Multi-DXE Direct Code Execution Arrangement Example

Determining Boot Stream Start Addresses

The ROM functions `BFROM_MEMBOOT`, `BFROM_TWIBOOT`, and `BFROM_SPIBOOT` not only allow the application to boot a subroutine residing at a given start address, they also assist in walking through linked multi-DXE streams.

When the `BFLAG_NEXTDXE` bit in `dFlags` is set and these functions are called, the system does not boot but instead walks through the boot stream following the next-DXE pointers. The `dBlockCount` parameter can be used to specify the DXE of interest. The routines then return the start address of the requested DXE's boot stream.

Initialization Hook Routine

When the ROM functions `BFROM_MEMBOOT`, `BFROM_TWIBOOT` and `BFROM_SPIBOOT` are called, they create an instance of the `ADI_BOOT_DATA` structure on the stack and fill the items with default values. If the `BFLAG_HOOK` is set, the boot kernel invokes a callback routine which was passed as fourth argument of the ROM routines, after the default values have been filled. The hook routine can be used to overwrite the default values. Every hook routine should meet the prototype:

```
void hook (ADI_BOOT_DATA* pBS);
```

The VisualDSP++ header files define the `ADI_BOOT_HOOK_FUNC` type the following way:

```
typedef void ADI_BOOT_HOOK_FUNC (ADI_BOOT_DATA* ) ;
```

Specific Boot Modes

This section discusses individual boot modes and the required hardware connections.

The boot modes differ in terms of the booting source— for example whether data is loaded through the SPI or the TWI interface. Boot modes can be grouped into slave boot modes and master boot modes.

In slave boot modes, the Blackfin processor functions as a slave to any host device, which is typically another embedded processor, an FPGA device or even a desktop computer. Likely, the Blackfin processor $\overline{\text{RESET}}$ input is controlled by the host device. So, usually the host sets $\overline{\text{RESET}}$ first, then waits until the preboot routine terminates by sensing the HWAIT output, and finally provides the boot data.

If a Blackfin processor, configured to operate in any of the slave boot modes, awakens from hibernate, it cannot boot by its own control. A feedback mechanism has to be implemented at the system level to inform the host device whether the processor is in hibernate state or not. The HWAIT strobe is an important primitive in such systems.

In the master boot modes, the Blackfin processor usually does not need to be synchronized and can load the boot data by itself. Master modes typically read from memory. This can be parallel memory such as flash devices, or serial memory that is read through SPI or TWI interfaces.

Memory boot modes should also be differentiated from peripheral boot modes. Boot modes that load boot streams through memory DMA are referred to as memory boot mode, reading data from regular memory. Peripheral modes load boot data through peripherals such as UART, TWI or SPI. With the exception of the FIFO boot, which is a hybrid, all memory boot modes are master modes. The boot source is typically non-volatile memory, such as a flash or EPROM device or even on-chip ROM. When supported by the system in warm boot scenarios, the boot source can also be SRAM or SDRAM.


Whether from the host (slave booting mode) or from memory (master booting mode), the boot source does not need to know about the structure of the boot stream. However in the case of Host DMA boot, the size (byte count) of the boot stream should be known. This is because, having much more control over the Blackfin processor, the host must know what data is to be loaded to specific addresses.

No Boot Mode

When the `BMODE` pins are all tied low (`BMODE = 0000`), the Blackfin processor does not boot. Instead it processes factory-programmed OTP pages, then executes an `IDLE` instruction, preventing it from executing any instructions provided by the regular boot source. The purpose of this mode is to bring the processor up to a clean state after reset.

This mode helps to recover from malicious OTP configuration since it prevents execution of the user-controllable portion of the preboot routine.

When connecting an emulator and starting a debug session, the processor awakens from an idle due to the emulation interrupt and can be debugged in the normal manner.

 The no boot mode is not the same as the bypass mode featured by the ADSP-BF53x Blackfin processor. To simulate that bypass mode feature using `BMODE = 0000`, see [“Direct Code Execution” on page 17-37](#) and [“Direct Code Execution” on page 17-164](#).

Flash Boot Modes

These booting modes are intended to boot from flash or EEPROM memories or even from battery-buffered SRAMs. The flash boot modes are activated by $BMODE = 0000$. Although this is a single $BMODE$ setting, the ADSP-BF54x Blackfin products support various configurations.

- Boot from 8-bit asynchronous flash memory
- Boot from 16-bit asynchronous flash memory
- Boot from 16-bit asynchronous page-mode NOR flash memory
- Boot from 16-bit asynchronous burst-mode NOR flash memory

By default, the boot kernel does not alter any EBIU registers. Therefore, traditional asynchronous flash is assumed and maximum wait states are applied. By programming OTP half pages $PBS00L$ and $PBS00H$, the user has the option to instruct the preboot routine to alter the EBIU registers as desired. In this way, the EBIU can be preset to access the flash device in either page mode or burst mode. There are also options to customize bus settings, such as wait states and $ARDY$ behavior.

After the preboot routine returns and `HWAIT` is deasserted the first time, the boot kernel loads an initial burst of four 16-bit words. Then it interrogates the `DMACODE` field in the byte loaded from the `0x2000 0000` address. For flash mode, the following DMA options, as shown in [Table 17-10](#) are supported.

Table 17-10. DMA Options

DMACODE	DMA Width	Source Modify	Comment
1	8	1	Not recommended Provides ADSP-BF533 style 8-bit boot from 16-bit flash memory
2	8	2	8-bit MDMA boots from 8-bit flash mapped to lower byte of address bus.
6	16	2	16-bit MDMA boots from 16-bit flash
10	32	4	32-bit MDMA boots from 16-bit flash

The `DMACODE` field is filled by the `elfloader` utility based on boot mode, `-width` and `-dmawidth` settings. See the *VisualDSP++ Loader and Utility Manual* for details.

After the boot kernel has loaded and interpreted the first four 16-bit words, it continues loading the rest of the first block header and processes the boot stream.

Most of the popular page-mode and burst-mode NOR flash devices default to traditional flash mode and are perfectly designed for altering the operating mode along the way. Theoretically, if the user hesitated to customize boot settings through OTP programming, there was still the option to start booting in traditional asynchronous mode and to alter EBIU settings through an `initcode` which is loaded and executed early in the boot process.



If the preboot features are not used and the NOR flash device is put into burst or page mode, it must be programmed back to the standard mode before the processor is reset. If the processor can

Specific Boot Modes

reset itself without software control (through watchdog or double-fault error), a mechanism must be installed that also resets the flash device back to default mode along with the processor. One method to address this is to set the `OTP_RESETOUT_HWAIT` bit in OTP half page `PBS00L` and to connect the `HWAIT` signal to the reset input pin of the NOR flash device.

Hardware configurations for the individual modes are shown in [Figure 17-12](#) to [Figure 17-13](#). The chip select is always controlled by the $\overline{\text{AMS0}}$ strobe. This maps the boot stream to the Blackfin processor's address `0x2000 0000`.

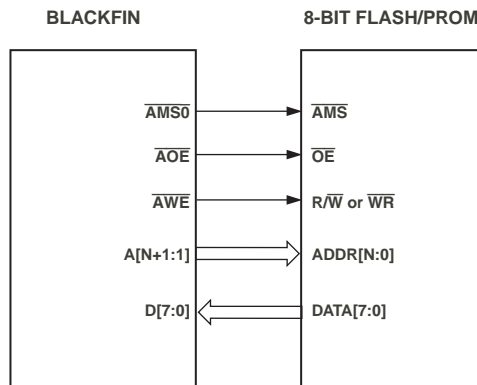


Figure 17-12. 8-Bit Flash Interconnection

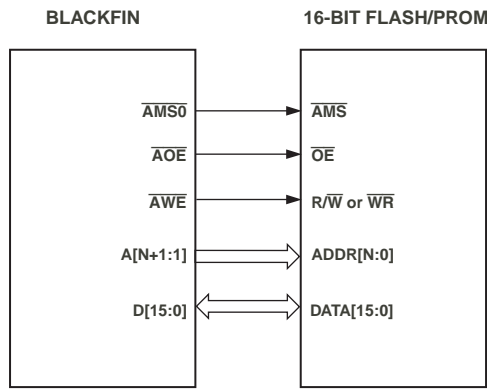


Figure 17-13. 16-Bit Flash Interconnection

See the chapter on System Design for connection of page-mode and burst-mode flash devices.

Some flash devices provide write protection mechanisms, which can be activated during the power-up and reset cycles of the Blackfin processor. In the absence of such mechanisms, a pull-up resistor on the $\overline{\text{AMS0}}$ strobe prevents the chip select from floating when the state of the processor is unknown.

i In flash mode all the muxed address lines (A4 to A9 on port H and A10 to A25 on port I) are activated by the boot kernel. When $\text{BMODE} = 0001$, none of these pins can function as an input without external hardware protection. Upper address pins are unlikely to toggle and can still be used for GPIO output purposes, with the limitation that the pins are driven low during boot time.

When the EBIU registers are configured to burst-flash mode by the pre-boot due to OTP programming, the boot kernel activates the NOR clock on the PI15 pin rather than the A25 line.

Specific Boot Modes

After $\overline{\text{RESET}}$ has released, the preboot processes a number of OTP pages. Then, the boot kernel starts reading data from the external flash memory. The initial cycles of the flash boot are shown in Figure 17-14 through Figure 17-23. The first 4-word burst loads half of the first boot block header in. After the DMACODE is evaluated the rest of the first block is loaded by the second 4-word burst. As settings are now known the next header is then loaded as an 8-word (16-byte) entity.

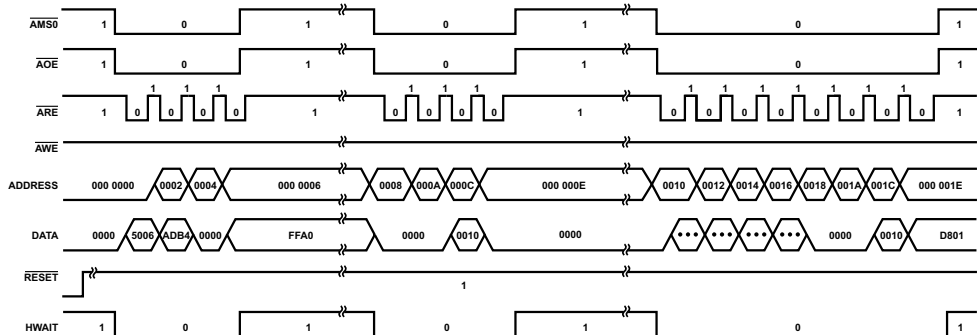


Figure 17-14. 16-bit Flash Mode Waveform (ADSP-BF54x)

The boot mode $\text{BMODE} = 0001$ can also be used to instruct the boot kernel to terminate immediately and directly execute code from the 16-bit flash memory instead. Code execution from 8-bit flash memory is not supported. See “[Direct Code Execution](#)” on page 17-37 for details.

SDRAM Boot Mode

From the boot kernel perspective, the SDRAM boot mode is just another memory boot mode like flash boot. The only differences are that the boot stream is expected at address 0x0000 0010 and the initial eight bytes are loaded by two 32-bit loads.

From the application point of view, SDRAM boot is a completely different scheme. Since SDRAM is volatile memory, $B_{MODE} = 1010$ is not a valid setting when the processor and the memories have just been powered up. This mode can only be used as a dynamically applied B_{MODE} setting to install warm boot scenarios.

OTP programming is required to boot from SDRAM. Other boot modes can configure the SDRAM controller by execution of an initcode. But in the case of SDRAM boot, the initcode cannot be loaded without having the SDRAM controller already configured.

SDRAM boot is meaningful when the Blackfin processor is in hibernate state or is completely shut off for power savings while the SDRAM is kept alive in self-refresh mode.

Users who prefer to execute code out of SDRAM, rather than performing a boot from it, may refer to [“Direct Code Execution” on page 17-37](#) for details.

FIFO Boot Mode

The FIFO boot mode ($B_{MODE} = 0010$) boots the Blackfin processor from another processor or FPGA system, referred to as the host device. The host is decoupled from the Blackfin bus by an asynchronous FIFO memory. When compared to the glue-less Host DMA boot modes, the FIFO mode requires less intelligence from the host. The host device is only expected to handshake with the FIFO and to load the entire boot stream in 16-bit portions. There is no need for the host to know about the content and format of the boot stream.

Specific Boot Modes

The hardware configuration for the FIFO boot mode is shown in [Figure 5-6 on page 5-50](#). The FIFO chip select connects to the $\overline{\text{AMS3}}$ strobe. Data read requests go to the $\overline{\text{DMAR1}}$ input on pin PH6. The host device controls the Blackfin processor's $\overline{\text{RESET}}$ input. As in all slave modes, the host device should not send requests to $\overline{\text{DMAR1}}$ unless the HWAIT signal goes inactive. The host device may optionally rely on HWAIT edges to continue or discontinue transmission of boot data in an interrupt controlled manner.


From the boot kernel perspective the FIFO boot mode ($\text{BMODE} = 0010$) is just another memory boot mode, the only exception being that the HMDMA1 block is enabled in advance. Activating this functionality makes the FIFO boot mode become a slave mode.

The bits set in the HMDMA1_CONTROL register are SND , REP and HMDMAEN . The SND bit is new to ADSP-BF54x Blackfin products. The ADSP-BF54x processor's FIFO boot mode differs slightly from the FIFO boot mode provided by the ADSP-BF52x and ADSP-BF53x Blackfin processors.

In the FIFO boot mode, the DMACODE field in the boot block headers must always be 0x06, which instructs the boot kernel to perform 16-bit DMA. The boot kernel increments the applied addresses as if reading from flash memory.

SPI Master Boot Mode

This mode ($\text{BMODE} = \text{b}\#0011$) boots from SPI memories connected to the SPI0 interface. 8-, 16-, 24-, and 32-bit address words are supported. Standard SPI memories are read using either the standard 0x03 SPI read command or the 0x0B SPI fast read command.

 Unlike other Blackfin processors, the ADSP-BF54x Blackfin processors have no special support for DataFlash devices from Atmel. Nevertheless, DataFlash devices can be used for booting and are

sold as standard 24-bit addressable SPI memories. They also support the fast read mode. If used for booting, DataFlash memory must be programmed in the power-of-2 page mode.

For booting, the SPI memory is connected as shown in [Figure 17-15](#).

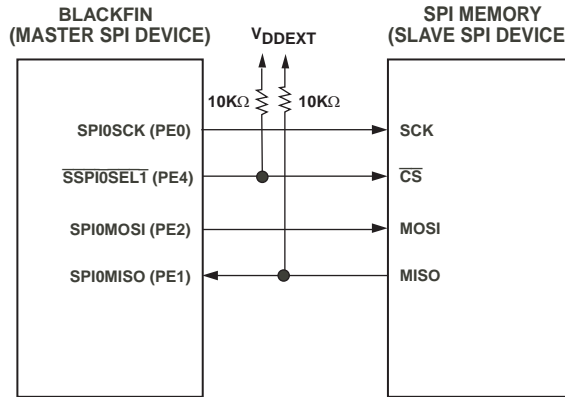


Figure 17-15. Blackfin to SPI Memory Connections

The pull-up resistor on the MISO line is required for automatic device detection. The pull-up resistor on the SSPI0SELT line ensures that the memory is in a known state when the Blackfin GPIO is in a high-impedance state (for example, during reset). A pull-down resistor on the SPI0SCK line displays cleaner oscilloscope plots during debugging.

For SPI master boot, the SPE, MSTR and SZ bits are set in the SPI0_CTL register (See [Table 17-11](#)). With TIMOD=2, the receive DMA mode is selected. Clearing both the CPOL and CPHA bits results in SPI mode 0. The boot kernel does not allow SPI0 hardware to control the SSPI0SELT pin. Instead, this pin is toggled in GPIO mode by software. Initialization code is allowed to manipulate the uwSsel variable in the ADI_BOOT_DATA structure to extend the boot mechanism to a second SPI memory connected to another GPIO pin.

Specific Boot Modes

By default, the boot kernel sets the `SPIO_BAUD` register to a value of 133, resulting in a bit rate of $SCLK/266$. This default value can be altered by programming the 4-bit `OTP_SPI_BAUD` field in OTP page `PBS00L`.

Table 17-11. Bit Rate

OTP_SPI_BAUD	SPI_BAUD	Bit Rate
b#0000	133	$SCLK/(2 \times 133)$
b#0001	Reserved	
b#0010	2	$SCLK/(2 \times 2)$
b#0011	4	$SCLK/(2 \times 4)$
b#0100	8	$SCLK/(2 \times 8)$
b#0101	16	$SCLK/(2 \times 16)$
b#0110	32	$SCLK/(2 \times 32)$
b#0111	64	$SCLK/(2 \times 64)$

Similarly, the boot kernel uses the standard `0x03` SPI read command, by default. Programming the `OTP_SPI_FASTREAD` bit in OTP page `PBS00L` enables the fast read mode where the boot kernel uses the `0x0B` read command instead and transmits a dummy zero byte after the address bytes.

SPI Device Detection Routine

Since `BMODE = 011` supports booting from various SPI memories, the boot kernel automatically detects what type of memory is connected. To determine whether the SPI memory device requires an 8-, 16-, 24- or 32-bit addressing scheme, the boot kernel performs a device detection sequence prior to booting. The `MISO` signal requires a pull-up resistor, since the routine relies on the fact that memories do not drive their data outputs unless the right number of address bytes are received.

Initially, the boot kernel transmits a read command (either 0x03 or 0x0B) on the `MOSI` line, which is immediately followed by two zero bytes. Once the transmission is finished, the boot kernel interrogates the data received on the `MISO` line. If it does not equal 0xFF (usually a `DMACODE` value of 0x01 is expected), then an 8-bit addressable device is assumed.

If the received value equals 0xFF, it is assumed that the memory device has not driven its data output yet and that the 0xFF value is due to the pull-up resistor. Thus, another zero byte is transmitted and the received data is tested again. If it differs from 0xFF, either a 16-bit addressable device (standard mode) or an 8-bit addressable device (fast read mode) is assumed.

If the value still equals 0xFF, device detection continues. Device detection aborts immediately if a byte different than 0xFF is received. The boot kernel continues with normal boot operation and it re-issues a read command to read from address 0 again. The first block header is loaded by two read sequences, further block headers and block payload fields are loaded by separate read sequences.

Specific Boot Modes

Figure 17-16 illustrates how individual devices would behave.

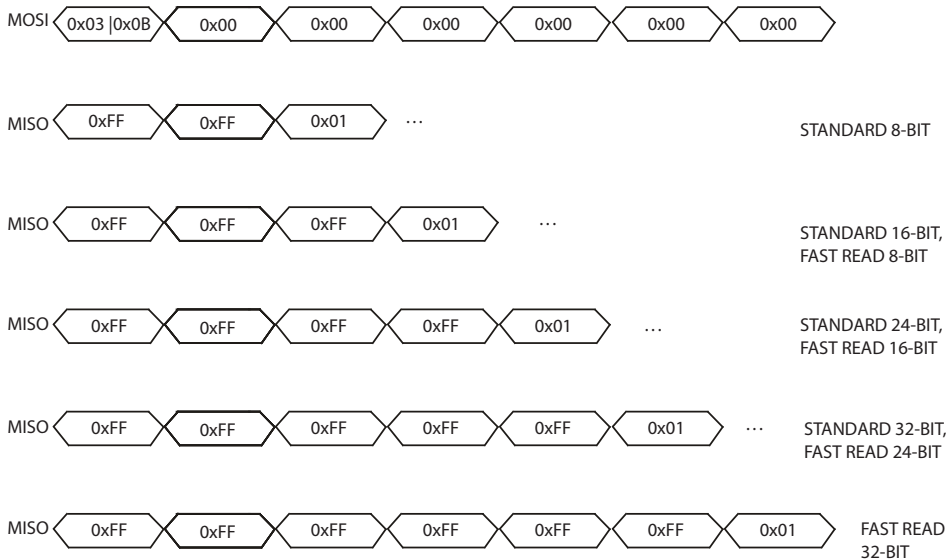


Figure 17-16. SPI Device Detection Principle

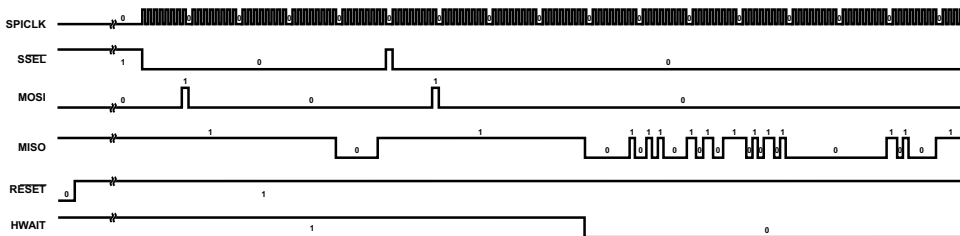


Figure 17-17. SPI Master Boot

Figure 17-17 on page 17-74 shows the initial signaling when a 24-bit addressable SPI memory is connected in SPI master boot mode. After $\overline{\text{RESET}}$ releases and preboot has processed relevant OTP pages, a 0x03 command is transmitted to the MOSI output, followed by a number of 0x00 bytes. The 24-bit addressable memory device returns a first data byte at the fourth zero byte. Then, the device detection has completed and the boot kernel re-issues a 0x00 address to load the boot stream.

SPI Slave Boot Mode

For SPI slave mode boot ($B_{MODE} = 100$), the Blackfin processor is consuming boot data from an external SPI host device. SPI0 is configured as an SPI slave device. The hardware configuration is shown in Figure 17-18. As in all slave boot modes, the host device controls the Blackfin processor's \overline{RESET} input.

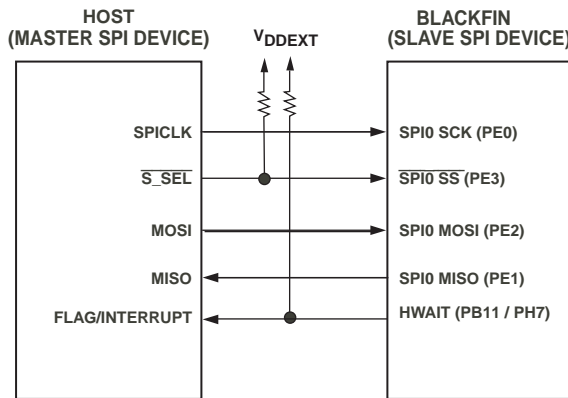


Figure 17-18. Connections Between Host (SPI Master) and Blackfin Processor (SPI Slave)

The host drives the SPI clock and is responsible for the timing. The host must provide an active-low chip select signal that connects to the $\overline{SPI0 SS}$ input of the Blackfin processor. It can toggle with each byte transferred or remain low during the entire procedure. 8-bit data is expected. The 16-bit mode is not supported.

In SPI slave boot mode, the boot kernel sets the $CPHA$ bit and clears the $CPOL$ bit in the $SPI0_CTL$ register. Therefore the $MISO$ pin is latched on the falling edge of the $MOSI$ pin. For details see the chapter on SPI-Compatible Port Controllers in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 2 of 2)*.

In SPI slave boot mode, H_{WAIT} functionality is critical. The H_{WAIT} handshake signal can operate on either the GPIO pin PB11 or on PH7 when the OTP_ALTERNATE_H_{WAIT} in OTP page PBS00L is programmed. When high, the resistor shown in [Figure 17-18 on page 17-76](#) programs H_{WAIT} to hold off the host. H_{WAIT} holds the host off while the Blackfin processor is in reset or executing the preboot. Once H_{WAIT} turns inactive, the host can send boot data. The SPI module does not provide very large receive FIFOs, so the host must test the H_{WAIT} signal for every byte. [Figure 17-19 on page 17-78](#) illustrates the required program flow on the host side.

[Figure 17-20 on page 17-79](#) shows the initial waveform for an SPI slave boot case. As soon as the Blackfin processor releases H_{WAIT} after reset, the host device pulls the S_{PISS} pin low and starts transmitting data. After the eight data word has been received, the boot kernel asserts H_{WAIT} again as it has to process the DMACODE field of the first block header. When the host detects the asserted H_{WAIT} it still finishes gracefully the transmission of the on-going word. Then, it pauses transmission until H_{WAIT} releases again.

Specific Boot Modes

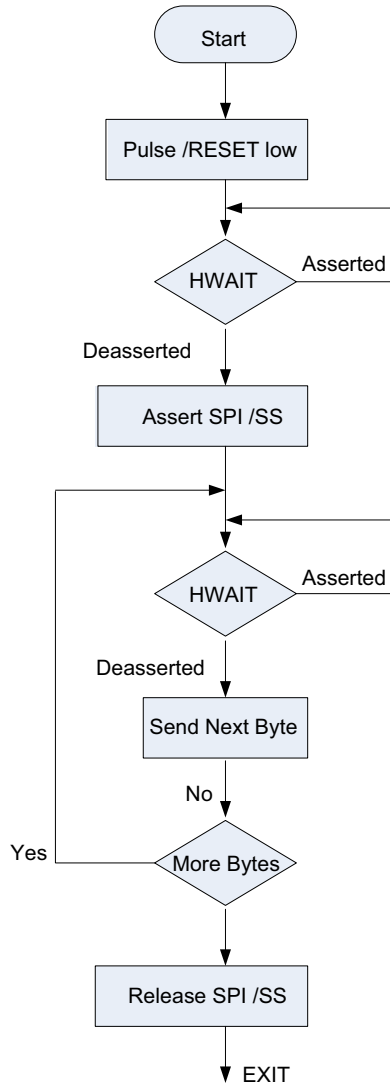


Figure 17-19. Program Flow on Host Device

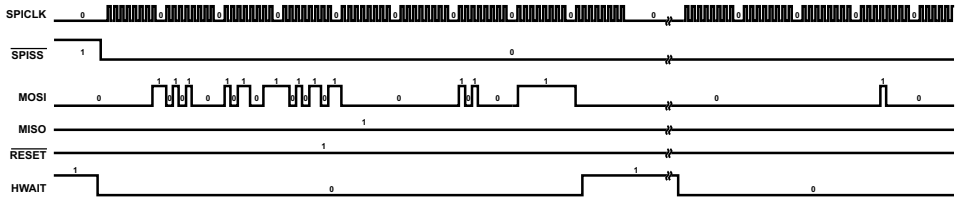


Figure 17-20. SPI Slave Boot

TWI Master Boot Mode

In TWI master boot mode ($BMODE = 0101$) the boot kernel reads boot data from I²C memory connected to the TWI0 interface. The Blackfin processor selects the slave EEPROM with the unique ID 0xA0, submits successive read commands to the device starting at internal address 0x0000, and begins clocking data to the processor. The EEPROM's device select bits A2–A0 must be 0s (tied low) when present. The I²C EPROM device should comply with Philips I²C Bus Specification version 2.1 and should have the capability to auto increment its internal address counter such that the contents of the memory device can be read sequentially. See [Figure 17-21](#).

i On the Blackfin processor, in both TWI master and slave boot modes, the upper 512 bytes starting at address 0xFF90 3E00 either must not be used or must be booted last. The boot ROM code uses this space for the TWI boot modes to temporarily hold the serial data which is then transferred to L1 instruction memory using DMA. All boot blocks that target the L1 instruction memory or external memories must have the `BFLAG_INDIRECT` bit set. Initcodes

Specific Boot Modes

can alter the placement of the temporary buffer by modifying the `pTempBuffer` and `dTempByteCount` variables in the `ADI_BOOT_DATA` structure.

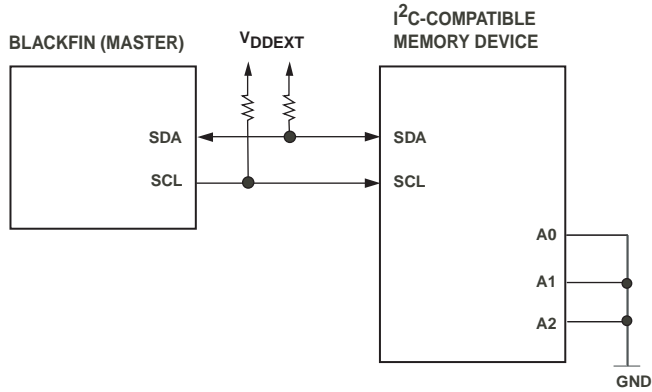


Figure 17-21. TWI Master Boot Mode

In [Figure 17-22](#), the Blackfin processor's TWI controller outputs the address of the I²C device to boot from, in this case 0xA0, where the least significant bit indicates the direction of the transfer. In this example, it is a write (0) to write the first two bytes of the internal address from which to start booting (0x00). [Figure 17-22](#) shows the TWI init and zero fill blocks.

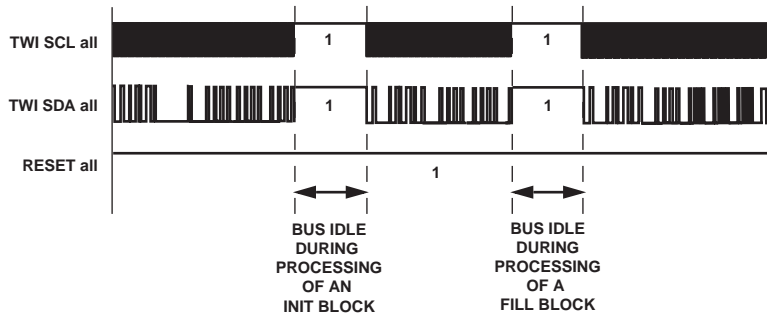


Figure 17-22. TWI Init and Fill Blocks

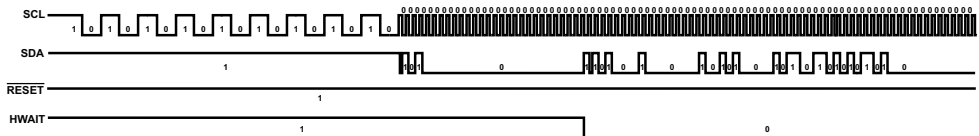


Figure 17-23. TWI Master Boot

[Figure 17-23](#) shows the initial waveforms in case of TWI master boot. After reset, the kernel generates nine slow pulses on the SCL output to ensure the TWI memory's state machine exits any pending state. Then a

Specific Boot Modes

start condition is issued and 0xA0 address command is issued, where the least significant bit indicates the direction of the write. In this case it is a write (0) in order to write two more 0x00 address bytes.

By default, it is assumed that the I²C memory device is two-byte addressable. This can be changed by programming the `OTP_TWI_TYPE` bit field in OTP page `PBS00L` as shown in [Table 17-12](#).

Table 17-12. Addressable Bytes

OTP_TWI_TYPE	Address Bytes
00	2
01	3
10	4
11	1

The TWI0 controller is programmed to generate a 30% duty cycle clock in accordance with the I²C clock specification for fast-mode operation (`PRESCALE = 0xA`, `CLKDIV = 0x811`) as shown in [Table 17-13](#). The default values can be altered by OTP programming. Setting the `OTP_TWI_CLKDIV` bit in OTP page `PBS00L` changes the `TWI0_CLKDIV` register value to `0x3232` as recommended for 100 kHz TWI operation. The `OTP_TWI_PRESCALE` field controls the prescale value written to the `TWI0_CONTROL` register.

Table 17-13. Prescale Value

OTP_TWI_PRESCALE	PRESCALE	Recommended
000	0x0A	SCLK = 100 MHz
001	0x0E	SCLK = 140 MHz (theoretical)
010	0x0C	SCLK = 120 MHz
011	0x0A	SCLK = 100 MHz
100	0x08	SCLK = 80 MHz
101	0x06	SCLK = 60 MHz
110	0x04	SCLK = 40 MHz
111	0x02	SCLK = 20 MHz

TWI Slave Boot Mode

In TWI slave boot mode ($B_{MODE} = 0110$) the Blackfin processor consumes data from a I²C host device that connects to the TWI0 interface. The I²C host selects the slave (Blackfin processor) with the 7-bit slave address 0x5F. When the Blackfin processor acknowledges, the host can download the boot stream. The I²C host should comply with Philips I²C Bus Specification version 2.1. The host supplies the serial clock.

Connections are shown in [Figure 17-24](#), [Figure 17-25](#) and [Figure 17-26](#).

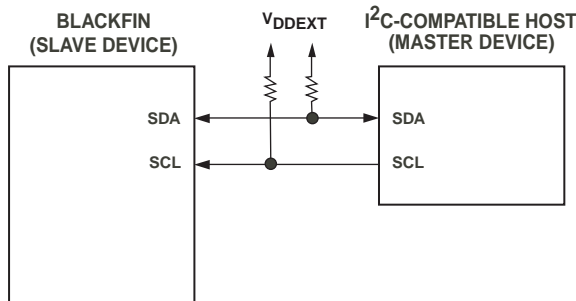


Figure 17-24. TWI Slave Boot Mode

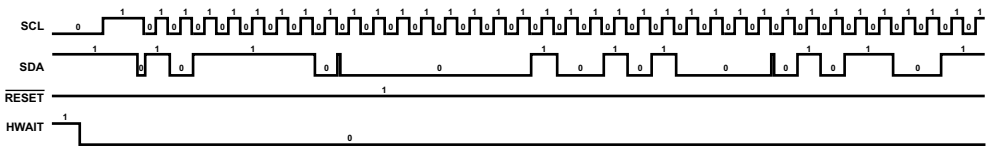


Figure 17-25. TWI Slave Boot

Specific Boot Modes

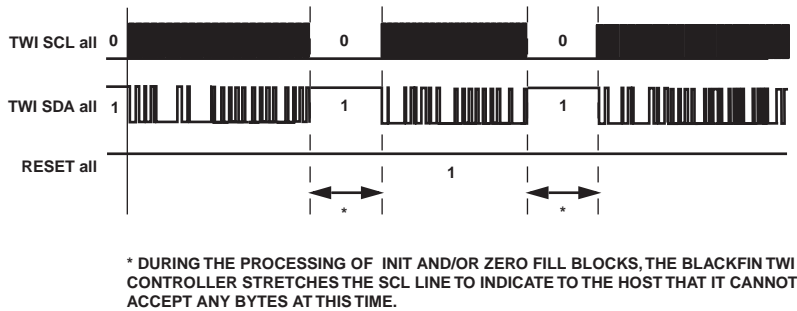


Figure 17-26. TWI Bit Stretching

Figure 17-25 on page 17-83 shows initial waveforms in case of TWI slave boot. As soon as `HWAIT` releases after reset the host starts transmitting the boot stream data. It starts by with a start conditions and a `0xBE` command, which is composite by the `0x5F` address and a trailing zero bit to indicate write direction.

i On the Blackfin processor, in both TWI master and slave boot modes, the upper 512 bytes starting at address `0xFF90 3E00` either must not be used or must be booted last. The boot ROM code uses this space for the TWI boot modes to temporarily hold the serial data which is then transferred to L1 instruction memory using DMA. All boot blocks that target the L1 instruction memory or external memories must have the `BFLAG_INDIRECT` bit set. Initcodes can alter the placement of the temporary buffer by modifying the `pTempBuffer` and `dTempByteCount` variables in the `ADI_BOOT_DATA` structure.

UART Slave Mode Boot

In the UART slave mode boot, the Blackfin processor consumes boot data from a UART host device connected to a UART interface.

For $BMODE = 0111$, the ADSP-BF54x processor consumes boot data from a UART host device connected to the UART1 interface. Automatic control of the \overline{RTS} output provides flow control.

The host downloads programs formatted as boot streams using an auto-baud detection sequence. The host selects a bit rate within the UART clocking capabilities. To determine the bit rate when performing the auto-baud, the boot kernel expects an “@” character (0x40, eight data bits, one start bit, one stop bit, no parity bit) on the UART RXD input. The boot kernel acknowledges, and the host then downloads the boot stream. The acknowledgement consists of four bytes: 0xBF, UARTx_DLL, UARTx_DLH, 0x00. The host is requested to not send further bytes until it has received the complete acknowledge string. Once the 0x00 byte is received, the host can send the entire boot stream. The host should know the total byte count of the boot stream, but it is not required to have any knowledge about the content of the boot stream. Further information regarding auto-baud detection is given in [“Autobaud Detection” on page 31-21](#).

When the boot kernel is processing fill or initcode blocks it might require extra processing time and needs to hold the host off from sending more data. This is signalled with the HWAIT output as well as by the RTS output. The host is not allowed to send data until HWAIT turns inactive after a reset cycle. Therefore a pulling resistor on the HWAIT signal is required.

Specific Boot Modes

If the resistor pulls to ground, the host must pause transmission when $HWAIT$ is low and is permitted to send when $HWAIT$ is high. A pull-up resistor inverts the signal polarity of $HWAIT$. The host should test $HWAIT$ at every transmitted byte.

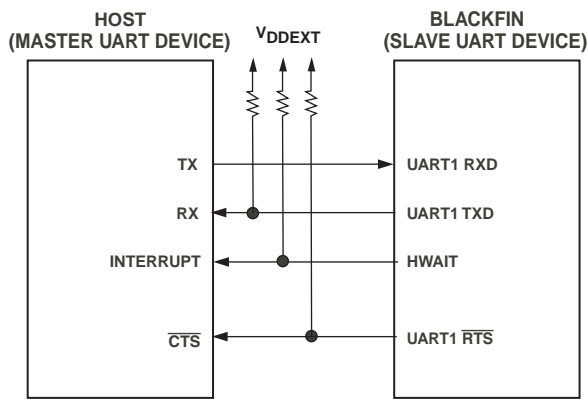


Figure 17-27. UART Slave Boot Mode

During ADSP-BF54x boot operation, the host device more likely relies on the RTS output of UART1. Then, the use of $HWAIT$ becomes optional. At boot time the Blackfin does not evaluate RTS signals driven by the host and the UART1 CTS input is inactive. Since the RTS is in a high impedance state when the Blackfin processor is in reset or while executing preboot, an external pull-up resistor to V_{DDEXT} is recommended.

Figure 17-27 shows the interconnection required for booting. The figure does not show physical line drivers and level shifters that are typically required to meet the individual UART-compatible standards.

Figure 17-28, Figure 17-29, and Figure 17-30 provide timing information for UART booting.

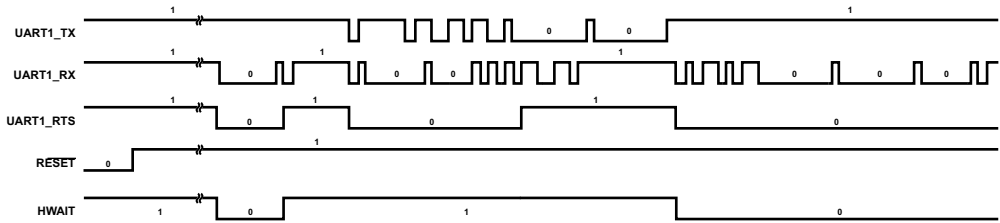


Figure 17-28. UART Autobaud Waveform

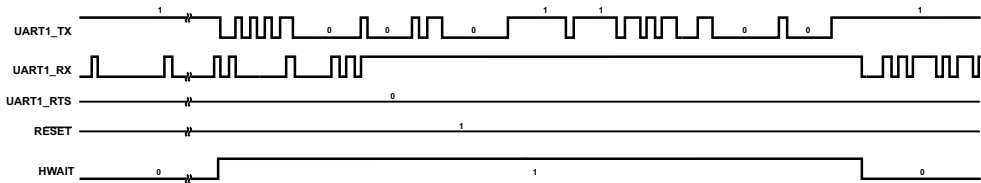


Figure 17-29. UART Boot - Host is relying on HWAIT

Specific Boot Modes

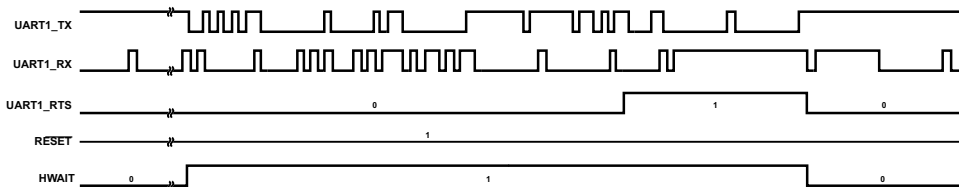


Figure 17-30. UART Boot - Host is relying on RTS

Figure 17-28 on page 17-87 shows the initial case of the UART boot mode. As soon as HWAIT releases after reset, the boot kernel expects to receive a 0x40 byte for bit rate detection. After the bit rate is known, the UART is enabled and the kernel transmits for bytes.

i When the UART is enabled, the $\overline{\text{RTS}}$ goes immediately low, encouraging the host to send first boot stream data as shown in the figure. In case of a half-duplex UART connection this must be avoided. The host should either rely on the HWAIT signal or wait until it has received the four bytes from the Blackfin processor, before sending any data.

Figure 17-29 and Figure 17-30 compare $\overline{\text{RTS}}$ and HWAIT timing in case an extended initcode executes. Since code execution is distracting from the data loading, the host devices has to be prevented to send more data. The HWAIT timing is much more conservative than the $\overline{\text{RTS}}$. If the host relies on HWAIT, the UART receive buffer may not be filled over watermark level and RTS might not be de-asserted at all. If, however, the host relies on RTS it will be stalled a couple of bytes later. Both methods are valid.

i In case of UART boot, it is not obvious on how to change the PLL by an initcode routine. This is because the UARTx_DLL and UARTx_DLH routines have to be updated to keep the required bit rate constant after the SCLK frequency has changed. It must be

ensured that the host does not send data while the PLL is changing. The initcode examples provided along with the VisualDSP++ tools installation demonstrate how this can be accomplished.

OTP Boot Mode

In the OTP boot mode ($B_{MODE} = 1011$), the boot kernel loads the boot stream from the on-chip OTP memory. OTP booting is a self-sufficient booting mechanism that does not require external boot memory or a host device.

By default the boot kernel starts loading the boot stream starting from OTP page 0x40. This is in the public OTP region. The boot stream can occupy all pages up to OTP page 0xDF, resulting in a boot stream length of up to 2560 bytes. The start address of the boot stream can be altered by programming the `OTP_START_PAGE` field in the `PBS01H` page. If there is no conflict with the alternate preboot pages feature, the `OTP_START_PAGE` field can be set to 0x20, resulting in a boot stream length of up to 3072 bytes.

In the current implementation, the OTP engine has no DMA support. Data is loaded and copied by core instructions. Nevertheless the `DMACODE` field should be set to 0xA, indicating 32-bit operation. The boot kernel ensures proper operation at 32-bit granularity, but 64-bit alignment may help to reduce the number of OTP pages that have to be read during boot processing. Byte 0 of the boot stream is expected to be byte 0 of the lower 32-bit word of the lower 0x40 half page.



In the OTP boot mode, the upper 512 bytes starting at address 0xFF90 3E00 either must not be used or must be booted last. The boot ROM code uses this space to temporarily hold the serial data which is then transferred to L1 instruction memory using DMA. All boot blocks that target the L1 instruction memory or external memories must have the `BFLAG_INDIRECT` bit set. Initcodes can alter

Specific Boot Modes

the placement of the temporary buffer by modifying the `pTempBuffer` and `dTempByteCount` variables in the `ADI_BOOT_DATA` structure.

Host DMA Boot Modes

The Host DMA boot modes differ completely from other boot modes because the boot kernel has no control over the DMA channels. The host device masters the DMA, so the host device must be able to parse the boot stream by itself.

The two host DMA boot modes (`BMODE = 1110` for 16-bit and `BMODE = 1111` for 8-bit) are almost identical. The differences are the port muxing control and the initial programming of the `HOST_CONTROL` register. The 16-bit boot mode uses the HOSTDTP's acknowledge mode while the 8-bit boot mode sets the `INT_MODE` bit in `HOST_CONTROL` to activate the interrupt mode.

Connection of a host device to the Blackfin processor is discussed in [Chapter 8, “Host DMA Port”](#). For booting the host device should control the `RESET` of the Blackfin processor. The host processor must poll `HOST_STATUS` register using a configuration read of the HOSTDTP until the `ALLOW_CNFG` bit is set indicating that the host may begin sending the 7 configuration words. This is necessary before each configuration of the HOSTDTP. The host processor may optionally sense the `HWAIT` signal to determine when it should begin polling the `ALLOW_CNFG` bit.

The HOSTDTP interface does not support the advanced boot kernel operations such as fill, CRC or callback. There is simple support to simulate the initcode functionality. Typically, this feature is not so important when the preboot's OTP memory pages can be programmed to configure the PLL and SDRAM controllers. However, if the user does not have the option to program OTP memory, the simulated initcode is the only

option to speed up the processor clocks and to enable the SDRAM controller for booting. One of these options must be used for the host device to boot into SDRAM memory.

In order to simulate initcodes the host device must send a valid initcode routine to the L1 instruction address 0xFFA0 0000. Additionally, the host is required to issue an `HIRQ` command after sending the 7 configuration words (but before sending any data) for the initcode block to the `HOSTDP`. Once the boot kernel detects an `HIRQ` command from the host and the DMA work unit is complete, the boot kernel will issue a `CALL` instruction to the address held in the `EVT1` register, and the C language initcode routine is called. `EVT1` defaults to 0xFFA0 0000, but it can be modified by user instructions during the boot process. When the initcode returns, the regular boot process continues. This can be repeated multiple times if necessary.

If the initcode routine has properly configured the SDRAM controller, subsequent Host DMA work units can write to SDRAM memory. Similarly, if the initcode has programmed the PLL, the Host DMA port can run at higher speed since it is `SCLK` dependent.

The same scheme is used to terminate the boot process. When the host is ready to send the final boot block of the application it needs to send the 7 configuration words required by the `HOSTDP`. The host device should then send an `HIRQ` command followed by the remaining data. Once all data has written, the boot kernel executes another `CALL` instruction and the application takes control of the system rather than returning to the boot kernel.

[Figure 17-31 on page 17-92](#) illustrates boot kernel processing in the Host DMA boot mode. [Figure 17-32 on page 17-93](#) illustrates host device flow.

Specific Boot Modes

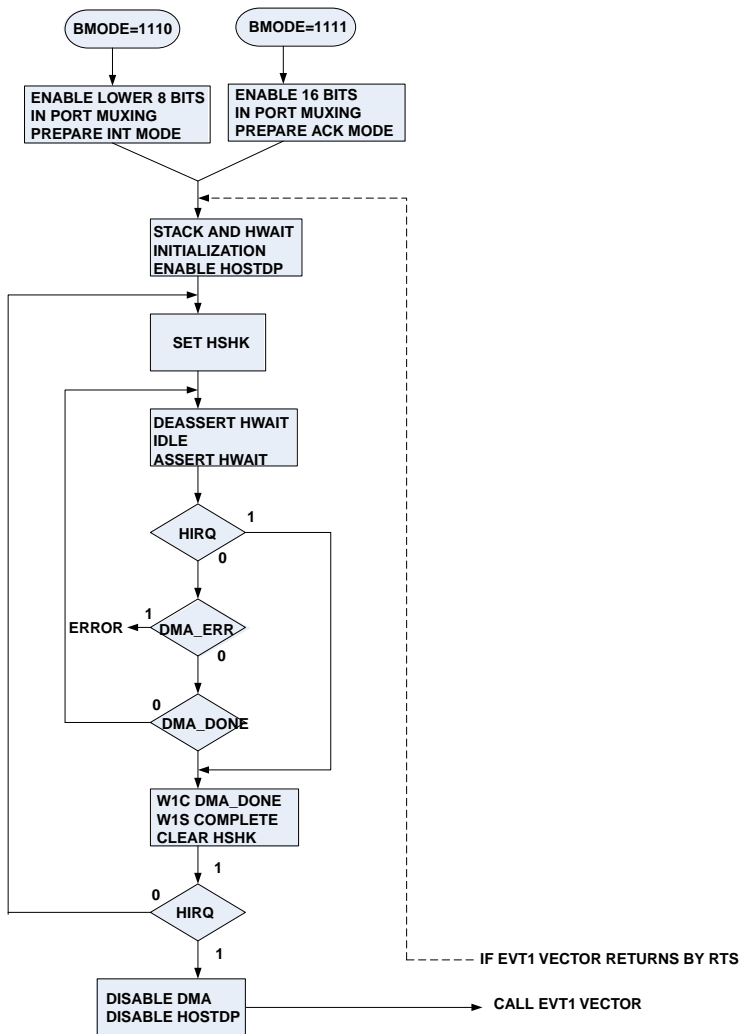


Figure 17-31. Boot Kernel Processing in Host DMA Boot Mode

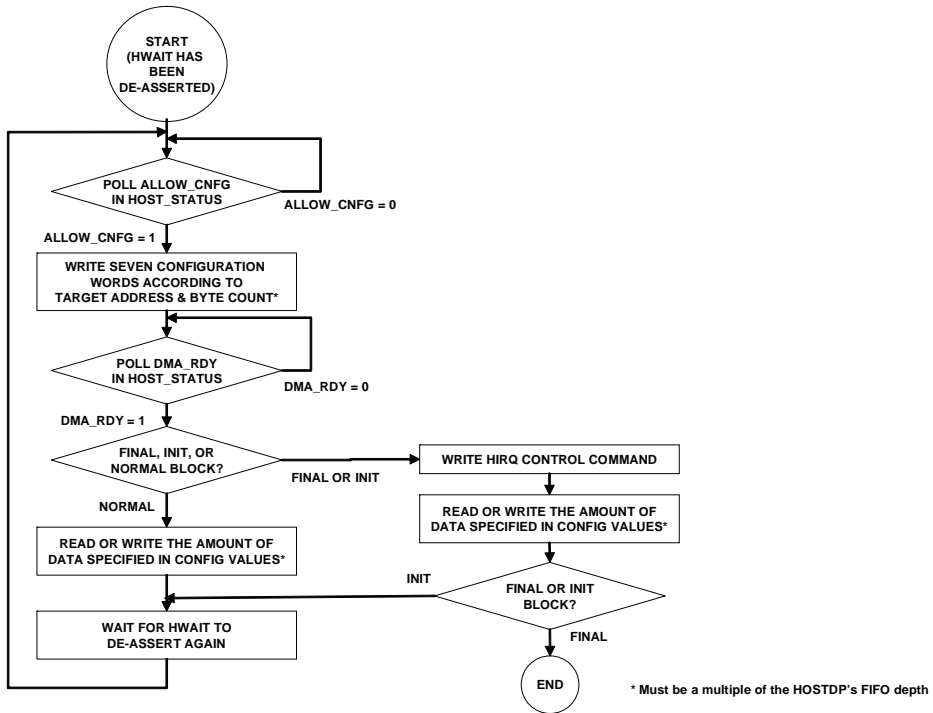


Figure 17-32. Host Device Flow

NAND Flash Boot Mode

NAND boot mode (BMODE = 1101) is intended to boot from SLC NAND flash memory devices connected directly to the NAND Flash Controller (NFC) of the ADSP-BF54x processors.

Specific Boot Modes

Although this is a single `BMODE` setting, the ADSP-BF54x family of processors supports booting from a number of various NAND flash device configurations:

- 8-bit Small Page SLC NAND flash
- 8-bit Large Page SLC NAND flash
- 16-bit Large Page SLC NAND flash

By default the NAND flash boot mode configures the read and write delay strobe timing parameters within the `NFC_CTL` register with `RD_DLY = 0x3` and `WR_DLY = 0x3`. This provides t_{RP} and t_{WP} timings of 4 SCLK cycles (30ns at 133 MHz) in order to provide maximum compatibility. By programming OTP half page `PBS01H`, the user has the option to instruct the preboot routine to provide alternate settings prior to accessing the NAND flash for the first access. In NAND boot mode, the `HWAIT` signal does not toggle. The respective GPIO pins remains in high-impedance mode.



Providing OTP configurations of `RD_DLY = 0x0` and `WR_DLY = 0x0` will result in the boot kernel using the default configuration of `RD_DLY = 0x3` and `WR_DLY = 0x3`. The highest performance settings that can be enabled for NAND boot are with `WR_DLY = 0x1` and `RD_DLY = 0x0`.

Supported Devices

NAND flash boot provides support for booting from a large number of NAND flash devices from a number of different manufacturers. There are two main classifications of single SLC NAND flash memories:

- Small Page NAND flash
- Large Page NAND flash

The small page NAND flash devices use a different addressing scheme for accessing the NAND flash array than that required by large page NAND flash devices. Additionally small page devices require a different command set for reading from different parts of the page.

Compatible small page devices supported for booting must comply with the array configuration as provided in [Table 17-14](#) and support the commands provided in [Table 17-15](#).

Table 17-14. Supported Small Page Device Array Configuration

Parameter	Size
Page Size	512 Bytes
Block Size (excluding spare area)	16384 Bytes (32 pages)
Spare Area	16 Bytes
1st half of page	256 Bytes
2nd half of page	256 Bytes
Maximum number of addressable blocks	524288

Table 17-15. Supported Small Page Commands

Operation	Command
Reset	0xFF
Read from 1st half of array	0x00
Read from 2nd half of array	0x01
Read from spare area	0x50



The NAND flash boot kernel, by default, issues 4 address cycles after issuing the read command. This redundancy can be removed by modifying the `uwNumCommands` parameter of the `ADI_BOOT_NAND_ADDRESS` structure from within an initialization routine that is executed before the main application boot stream is

Specific Boot Modes

processed. In order to load the initialization function however, 4 address cycles are always issued. The NAND flash device must be capable of ignoring the additional address cycles.

NAND flash boot provides support for a number of large page array configurations. The 4th byte of the NAND flash Electronic Signature is used to configure the boot kernel to allow correct access to the memory array. The boot kernel supports any large page NAND flash device configuration that is compliant with the format detailed in [Figure 17-33](#).

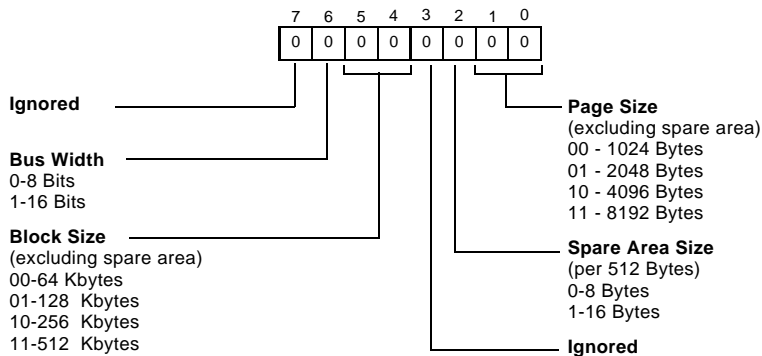





Figure 17-33. Supported 4th byte of NAND Flash Electronic Signature

The command set that must be supported is provided in [Table 17-16](#).

Table 17-16. Supported Large Page Commands

Operation	1st Command	2nd Command
Reset	0xFF	-
Read Electronic Signature	0x90	-
Read	0x00	0x30

-  Due to the auto detection method used. Large page NAND flash devices must not react to the issuing of command 0x50 followed by 4 address cycles by driving the $R\bar{B}$ signal low and then high again.
-  The NAND flash boot kernel, by default, issues 5 address cycles after issuing the read command. This redundancy can be removed by modifying the `uwNumCommands` parameter of the `ADI_BOOT_NAND_ADDRESS` structure from within an initialization routine that is executed before the main application boot stream is processed. In order to load the initialization function however, 5 address cycles are always issued. The NAND flash device must be capable of ignoring the additional address cycles.
-  Supported 16-Bit NAND flash memories must only use the lower 8 bits of the bus for the command and address cycles. 16-bit command and address cycles are not supported.

Specific Boot Modes

Hardware configuration for the NAND boot mode is shown in [Figure 17-34](#).

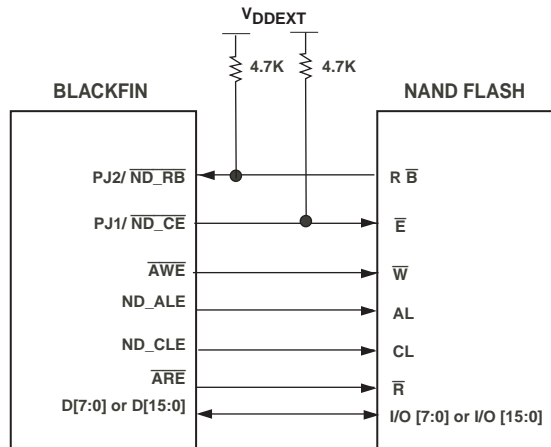


Figure 17-34. 8-Bit/16-Bit NAND Flash Interconnection

As the GPIO pins are originally configured as inputs, a pull-up resistor is required on PJ1\ND_CE. This ensures that the device is not selected after a reset until the required GPIO pins have been configured correctly.

Auto Detection

Once the boot kernel has detected the NAND boot option, the first operation to be performed is the auto detection procedure of the NAND flash device.

The boot kernel first of all issues a reset signal to the NAND flash device. The reset command brings the NAND flash out of the default read mode ready to accept a command. The NAND flash reacts by driving the R_B signal low and then high again.

The processor, after issuing the reset command, enters a nested loop that checks the status of the $\overline{R_B}$ signal every 100 CLK cycles. A maximum of 100 checks are performed. If the ready busy signal is not driven low and then high again after 100 attempts, then the boot kernel enters the safe idle mode as it assumed that no NAND flash device is present.

The routine terminates the first time the assertion of the signal is detected after which, the processor then proceeds to determine if the attached device is a small page NAND flash device.

The small page device detection consists of issuing the command to read from the spare area of the device, command 0x50, followed by 4 address cycles. Once again the processor then enters the nested loop routine waiting for detection of a rising edge of the $\overline{R_B}/\overline{ND_RB}$ signal. If the rising edge is detected then the boot kernel is configured to boot from the supported small page device. If no rising edge is detected by the time the loop terminates then the device is assumed to be a large page device. The processor issues a further reset command to reset the large page device then proceeds to read the electronic signature in order to configure the boot kernel appropriately.

Boot Stream Processing

In order to successfully boot from NAND flash, blocks of data must be first transferred to the processors internal memory in order to be processed by the boot kernel. This is achieved through the use of a 512 byte temporary storage space located at 0xFF907E00 - 0xFF907FFF.

This storage space is effectively split into two buffers each consisting of 256 bytes.

The 256 byte buffer at location 0xFF907E00 - 0xFF907EFF is what is referred to as the “MainBuffer”. The remaining 256 bytes from 0xFF907F00 - 0xFF907FFF is referred to as the “PrefetchBuffer”.

Specific Boot Modes

The NAND flash controller is configured for a 256 byte page size. During the boot phase, a single block transfer consists of 256 bytes. All block transfers from the NAND flash device go to the PrefetchBuffer. The boot kernel determines if the MainBuffer is empty, is partially processed or is fully processed. If the MainBuffer is empty or all data currently residing in the MainBuffer is processed, the boot kernel copies the contents of the PrefetchBuffer into the MainBuffer then requests another 256 block of data from the NAND flash. This process continues until the entire boot stream is processed.

An important requirement of NAND flash devices is the need for error checking and correction (ECC) to be performed on the received data. The NAND flash controller of the ADSP-BF54x devices employs a Hamming code algorithm to automatically generate 2 sets of parity data per 256 byte block transfer. The two sets of parity data each consist of 11 bits providing a total of 22 bits of parity data. This allows for the detection and correction of a single bit error within a 256 byte block, detection of a double error and detection of an error within the parity data itself. The boot kernel makes use of the embedded NFC ECC parity generation hardware and performs the error correction algorithm after every block transfer to the PrefetchBuffer providing greater reliability. The kernel is capable of detecting when the requested data resides in a new page. Before requesting the actual data, the kernel reads the data from the spare area section of the page, where the ECC parity data resides, to the PrefetchBuffer before storing internally on the stack to the EccParity structure. The parity data for the entire NAND flash page is stored allowing for ECC to be performed on all further data transfers from that page without requiring further access to the spare area. This allows for the kernel to adopt a more efficient access method by only issuing a single read command for sequential 256 byte block accesses to a page when requesting the actual data.



Due to the fact that the NAND boot procedure uses a prefetch mechanism, the 256 byte block following the end of the boot stream must have a correct ECC parity field programmed. Failure

to adhere to this will result in the boot kernel generating an uncorrectable error when fetching the block of data resulting in the boot process terminating.

Software Configurable NAND Boot Modes

The NAND boot mode provides support for three different boot methods with regards to handling errors and bad blocks that may be encountered:

- Sequential Block Mode (default)
- Block Skip Mode
- Multiple Image Mode

The three booting options provide users with great flexibility in how they wish to use a NAND flash for booting purposes.

The three boot modes are configured through the `uwBlockSkipFeature` variable of the `EccParity` structure. By default `uwBlockSkipFeature = 0`, configuring the device for Sequential Block Mode. The user can modify the boot mode by modifying the `uwBlockSkipFeature` variable from within an initialization routine that is loaded and executed before the main application boot stream is processed.

Access to the `ADI_BOOT_NAND` structure is provided by a pointer stored in the `dUserLong` parameter of the `ADI_BOOT_DATA` structure.

Sequential Block Mode

The default boot method is the Sequential Block Mode. In this mode no bad block detection is performed. The processor simply boots the boot stream starting from page 0 of block 0 until the end of the boot stream is reached. Error correction is always performed for greater reliability, however, if an uncorrectable error or error in the parity data is detected, the booting process terminates and the error handler is called.

Specific Boot Modes

This boot mode is ideally suited to applications that wish to adopt a second stage boot loader approach, where the second stage loader starts from the first byte in the NAND flash.

If the boot stream to be loaded expands a number of blocks then all blocks that the boot stream occupies must be good blocks. If a block in which the boot stream would occupy is known to be bad then this boot method should not be adopted for that particular device.

Figure 17-35 highlights some typical usage scenarios for this mode.

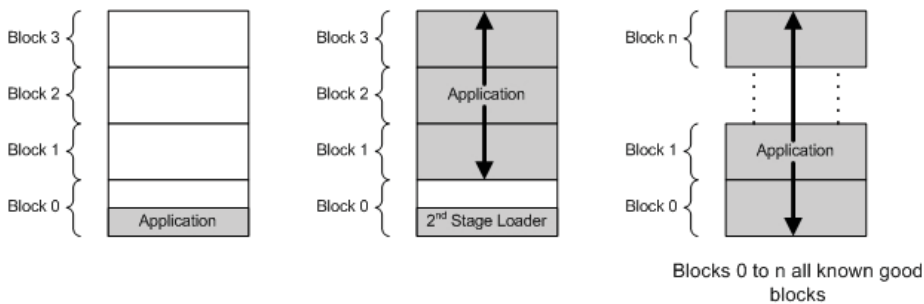


Figure 17-35. Sequential Block Mode Usage Scenarios

Block Skip Mode

This mode is enabled with `uwBlockSkipFeature = 1`. When enabling this mode the user must also set `uwBlockModifier = 1`. Failure to do so can result in the boot procedure failing.

This boot mode is ideally suited for larger applications not adopting the second stage loader approach. During the loading of the application to the NAND flash, upon detection of factory set bad block, the last byte of the spare area of the first and second page of the bad block is set to a non 0xFF value. The boot procedure works in a similar manner to the Sequen-

tial Block Mode except on detection of an access to a new block the spare area sections of the of the first two pages are loaded. The boot kernel checks the last byte of each. If either is not equal to 0xFF then the page is detected as bad. A byte offset of 1 block is then applied to all subsequent data requests thus skipping any bad blocks allowing for the booting of a single larger boot stream that is impeded by bad blocks in the area that the boot stream occupies. Each time a bad block is encountered the byte offset applied to the address of the requested data is incremented by 1 block. [Figure 17-36](#) highlights a typical usage scenario for this boot method.

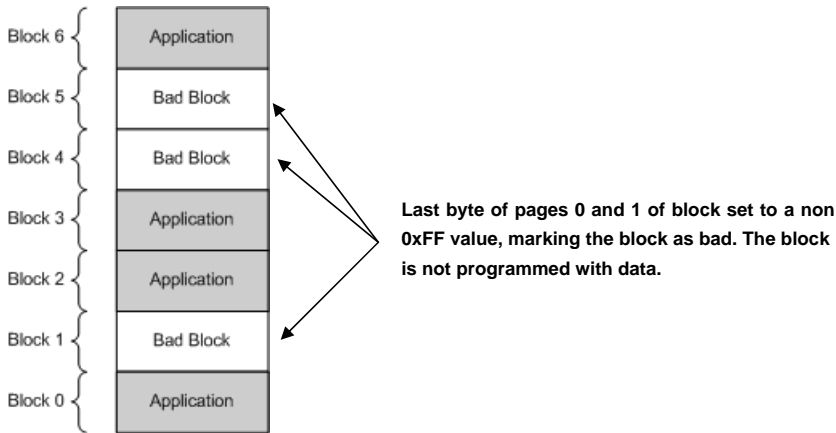


Figure 17-36. Block Skip Mode Typical Usage Scenario

Multiple Image Mode

This mode is enabled with `uwBlockSkipFeature = 2`. Multiple Image Mode allows for multiple copies of the boot stream to be loaded to the NAND flash providing maximum reliability. The number of blocks between each copy of the boot stream is defined by `uwBlockModifier`. Upon detection of an access to a new block, as in Block Skip Mode, the last byte of the spare area of the first and second page of the block are

Specific Boot Modes

checked to see if either indicate that the block is bad. If the block is bad, the block offset is applied to the requested data address to fetch from the next copy of the application. In addition, this mode is the only mode capable of handling uncorrectable errors as a result of error detection and correction. If an uncorrectable error is received in any block (including block 0) or an error is detected in the parity data the kernel will fetch the same block of data from the next copy of the application. The parameter `uwMaxCopies` specifies how many copies of the application are located in the NAND flash. If an uncorrectable error, error in the ECC parity data or a bad block is detected and the processor is booting from the final copy of the boot stream, processor then enters a safe idle state and the booting process is terminated. This boot method provides greater reliability that users may wish to adopt if regular boot stream updates are expected throughout the lifetime of a product.

Figure 17-37 shows a typical usage scenario.

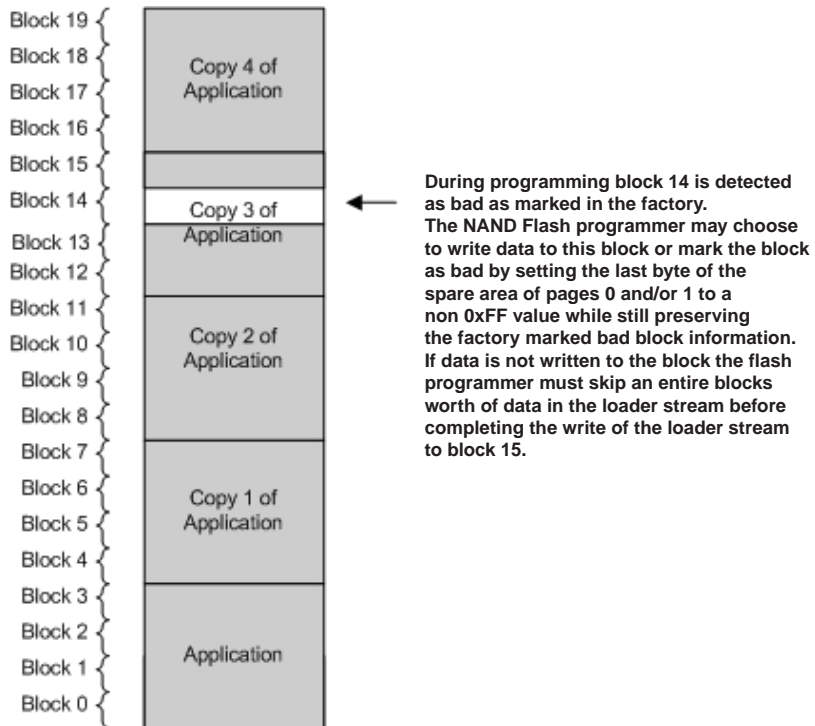


Figure 17-37. Mirror Image Mode Typical Usage Scenario

NAND Flash Page Structure

The NAND boot option transfers data contents from the main area of the NAND flash using a 256 byte DMA transfer on DMA channel 22. The spare area section at the end of the page contains the ECC parity data for each 256 sub block of the main data area. The spare area section is divided

Specific Boot Modes

into equal sizes corresponding to the number of 256 byte blocks contained within a page, so for a 512 Mbyte small page device the spare area is divided into two sections.

The first three bytes of each sub section of the spare area contains the 22-bit ECC parity data for the corresponding data block. The very last byte of the spare area is reserved in the first and second pages of each block for the bad block marker. [Figure 17-38](#) shows the page structure for a NAND flash device with a page size of 2048-Bytes. The 64-byte area is divided into eight 8-byte sections. The first three bytes of each section contains the parity data for the corresponding 256-byte block.

The last byte in the page is used as the bad-block marker in the event that the device only contains 8-bytes of spare area per 512-byte block instead of the more common 16-bytes per 512-byte block.

[Figure 17-26](#) on [page 17-84](#) shows an example of bit stretching.

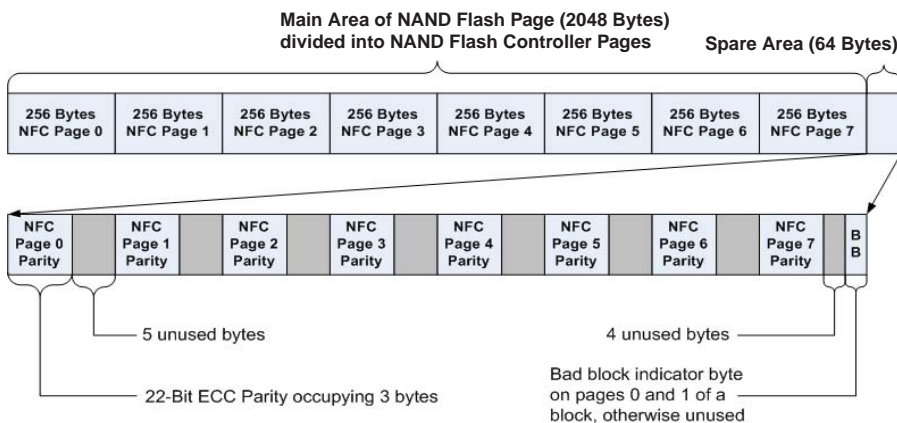


Figure 17-38. Page Layout of NAND Flash Device, 2048-Byte Page Size

Reset and Booting Registers

Two registers are used for reset and booting—the software reset register (SWRST) and the system reset configuration register (SYSCR).

Software Reset (SWRST) Register

A software reset can be initiated by setting bits [3:0] in the system software reset field in the software reset register (SWRST) shown in [Figure 17-39 on page 17-108](#). Bit 3 can be read to determine whether the reset source was core-double-fault. A core-double-fault resets both the core and the peripherals, but not the RTC block and most of the DPMC. Bit 15 indicates whether a software reset has occurred since the last time SWRST was read. Bit 14 indicates the software watchdog timer has generated the software reset. Bit 13 indicates the core-double-fault has generated the software reset. Bits [15:13] are read-only and cleared when the register is read. Reading the SWRST also clears bits [15:13] in the SYSCR register. Bits [3:0] are read/write.

Only writing to bits[2:0], resets only the modules in the SCLK domain. It does not clear the core. The program executes normally at the instruction after the MMR write to SWRST. The system is kept in the reset state as long as the bits[2:0] are set to b#111. To release reset, write a zero again. An example is shown in [Listing 17-2 on page 17-154](#). It is not recommended to use this functionality directly. Rather, call the ROM function `bfrom_SysControl()` to perform a system reset.

Reset and Booting Registers

Software Reset Register (SWRST)

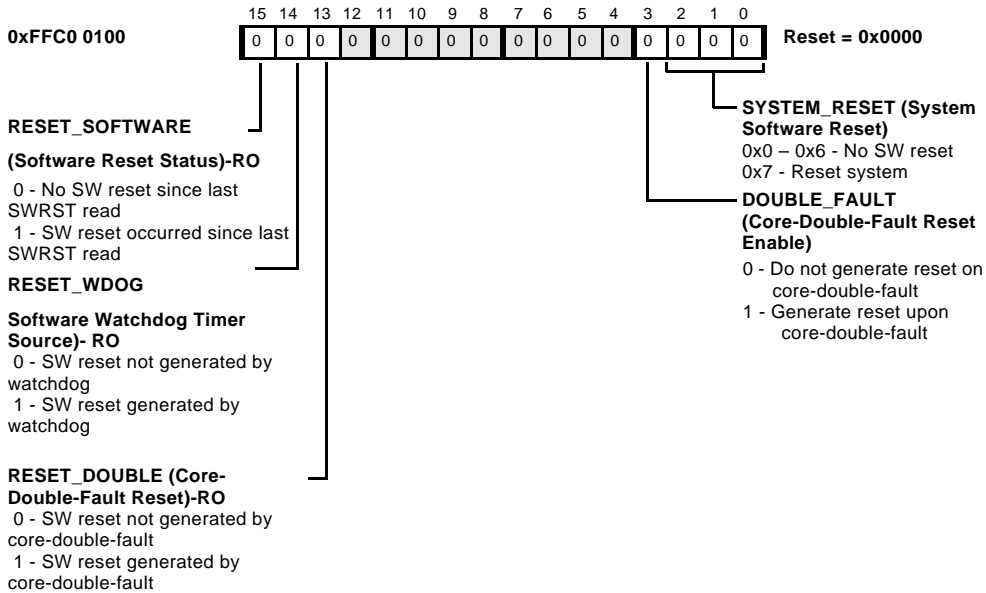


Figure 17-39. Software Reset Register

System Reset Configuration (SYSCR) Register

The values sensed from the `BMODE[3:0]` pins of the `SWRST` register are mirrored into the system reset configuration register (`SYSCR`). The values are available for software access and modification after the hardware reset sequence. Software can modify only bits[7:4] in this register to customize boot processing upon a software reset.

[Table 17-2 on page 17-4](#), and [Figure 17-1 on page 17-8](#) illustrate these booting sequences.

The bits [15:13] are exact copies of the same bits in the `SWRST` register. Unlike the `SWRST` register, `SYSCR` can be read without clearing these bits. Reading `SWRST` also causes `SYSCR[15:13]` to clear.

The `WURESET` indicates whether there was a wake up from hibernate since the last hardware reset. The bit cannot be cleared by software.

Bits [11:8] have no booting or reset purpose. These bits control the DMA arbitration.

The software reset configuration register (`SYSCR`) is shown in [Figure 17-40 on page 17-110](#).

Reset and Booting Registers

System Reset Configuration Register (SYSCR)

X - state is initialized from BMODE pins during hardware reset

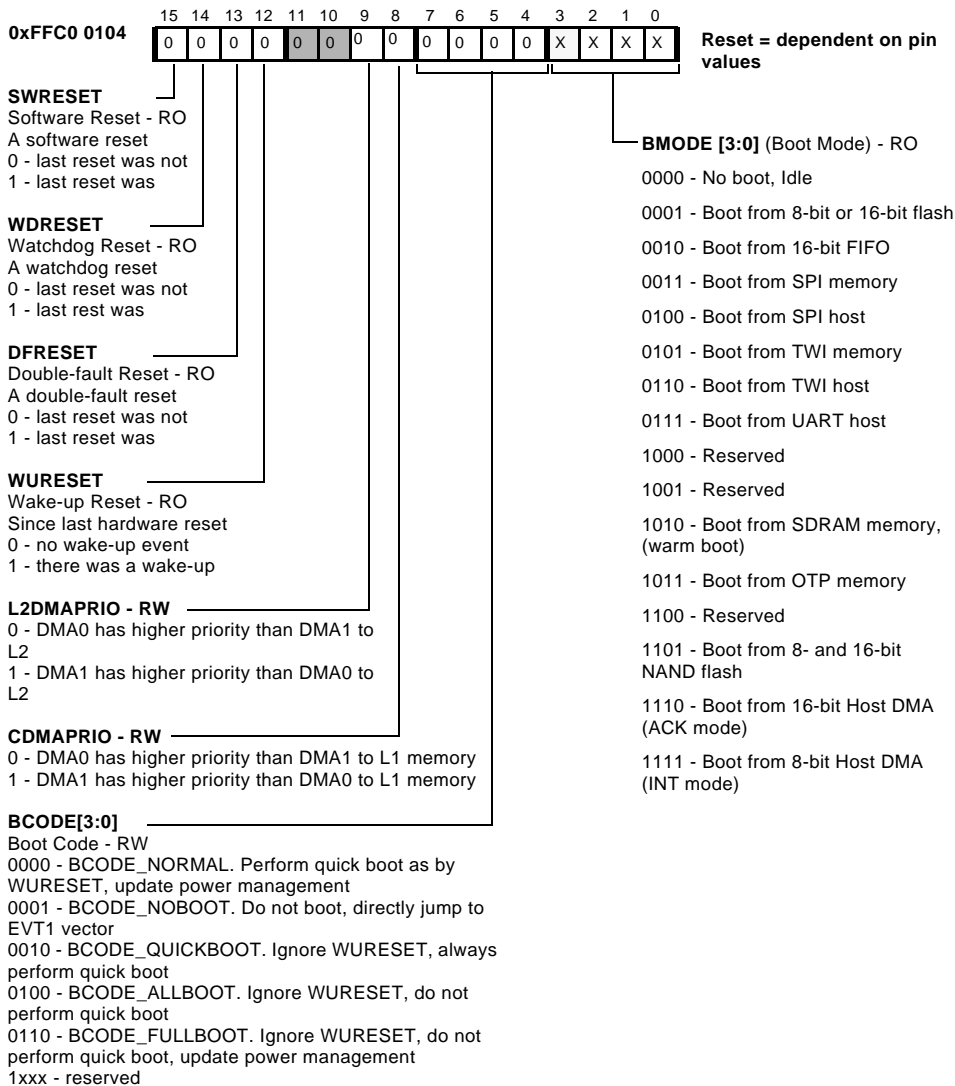
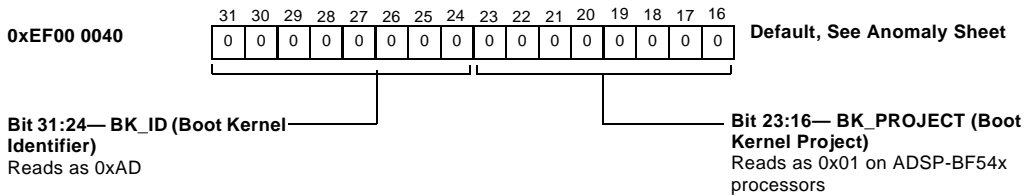


Figure 17-40. System Reset Configuration Register

Boot Code Revision Control (BK_REVISION)

The boot ROM reserves the 32-bit memory location at address 0xEF00 0040 for a version code consisting of four bytes as shown in [Figure 17-41 on page 17-111](#).

Boot Code Revision BK_REVISION Word, 31-16



Boot Code Revision BK_REVISION Word, 15-0

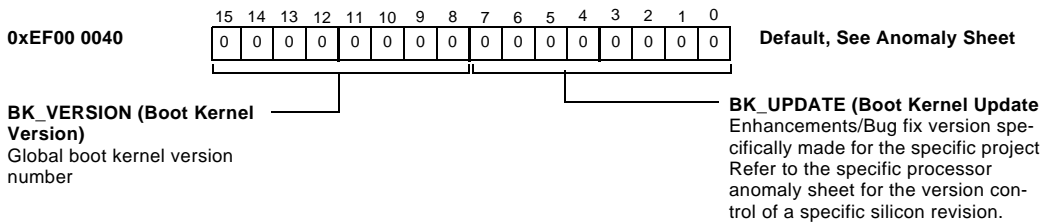


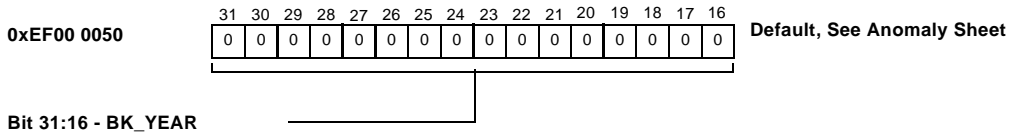
Figure 17-41. Boot Code Revision Code (BK_REVISION)

Reset and Booting Registers

Boot Code Date Code (BK_DATECODE)

The boot ROM reserves the 32-bit memory location at address 0xEF00 0050 to report the code of the build date as shown in [Figure 17-42 on page 17-112](#).

Boot Code Date Code BK_DATECODE Word, 31-16



Boot Code Date Code BK_DATECODE Word, 15-0

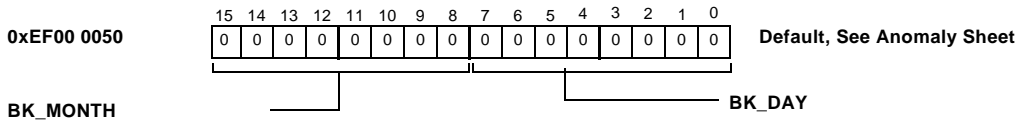
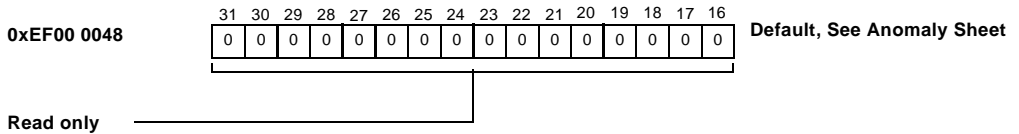


Figure 17-42. Boot Code Date Code (BK_DATECODE)

Zero Word (BK_ZEROS)

The boot ROM reserves the 32-bit memory location at address 0xEF00 0048 which always reads as 0x0000 000 as shown in [Figure 17-43](#) on page 17-113.

Zero Word BK_ZEROS, 31-16



Zero Word BK_ZEROS, 15-0

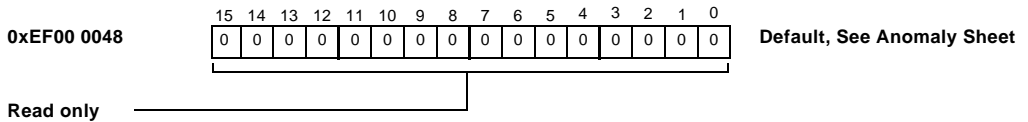


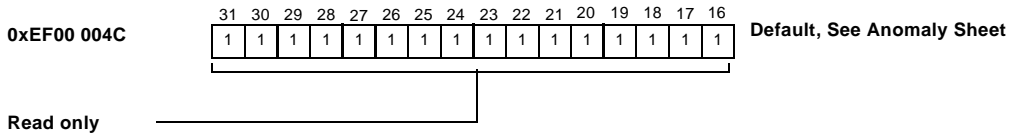
Figure 17-43. Zero Word (BK_ZEROS)

Reset and Booting Registers

Ones Word (BK_ONES)

The boot ROM reserves the 32-bit memory location at address 0xEF00 004C which always reads 0xFFFF FFFF as shown in [Figure 17-44](#) on page 17-114.

Ones Word BK_ONES, 31-16



Ones Word BK_ONES, 15-0

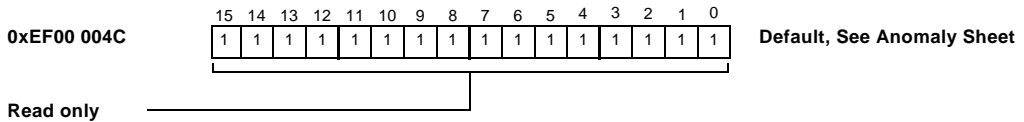


Figure 17-44. Ones Word (BK_ONES)

OTP Memory Pages for Booting

Lower PBS00 Half Page

The 64-bit lower half of page 0x18 is always read by the preboot routine. These control bits customize the boot process and instruct the preboot routine whether to process further pages and whether the PLL settings have to be changed. Other bits customize the SPI and TWI master boot speed.

OTP Memory Pages for Booting

Lower PBS00 Half Page (PBS00L, Upper 63-48)

One-Time Programmable

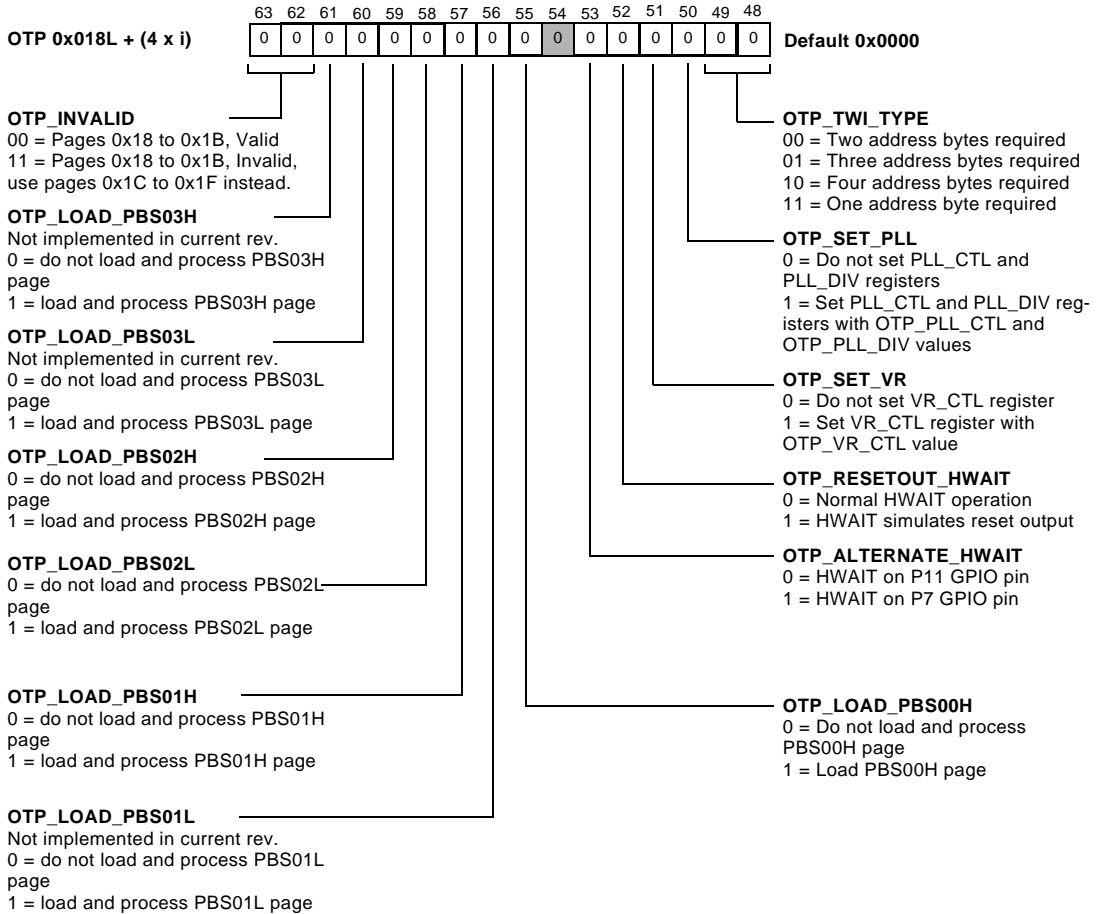


Figure 17-45. Lower PBS00 Half Page (PBS00L, Bits 63–48)

Lower PBS00 Half Page (PBS00L, Upper 47-32)

One-Time Programmable

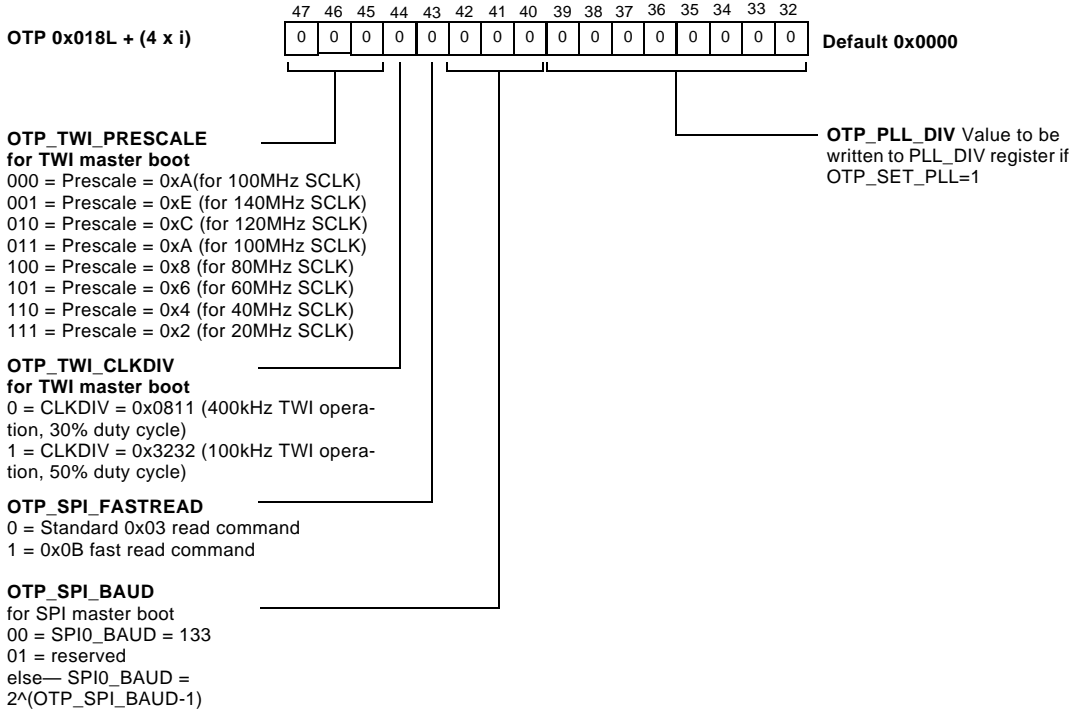
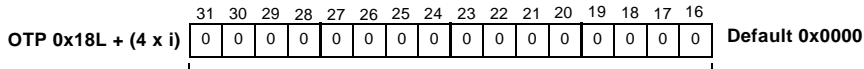


Figure 17-46. Lower PBS00 Half Page (PBS00L, Bits 47–32)

OTP Memory Pages for Booting

Lower PBS00 Half Page (PBS00L, Lower 31-16)

One-Time Programmable

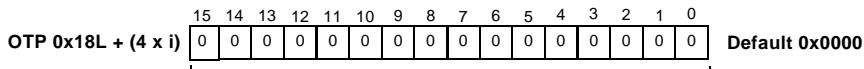


OTP_PLL_CTL

Value to be written to PLL_CTL register if
OTP_SET_PLL=1

Lower PBS00 Half Page (PBS00L, Lower 15-0)

One-Time Programmable



OTP_VR_CTL

Value to be written to VR_CTL register if
OTP_SET_VR=1

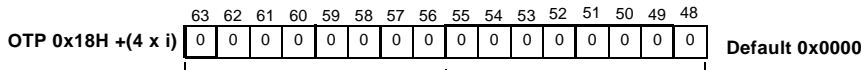
Figure 17-47. Lower PBS00 Half Page (PBS00L, Bits 31–0)

Upper PBS00 Half Page

The preboot routine loads the upper 64-bit half of page PBS00 only if the `OTP_LOAD_PBS00H` bit in the `PBS00L` page is set. Page `PBS00H` customizes the default setting of the asynchronous portion of the EBIU controller.

Upper PBS00 Half Page (PBS00H, Upper 63-48)

One-Time Programmable

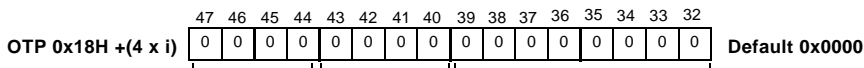


OTP_EBIU_DEVCFG

Device Configuration word to be used by device sequence.

Upper PBS00 Half Page (PBS00H, Upper 47-32)

One-Time Programmable



OTP_EBIU_DEVSEQ

0010 = perform 16-bit Atmel, Intel, ST sequence
 0100 = perform 16-bit Spansion sequence
 0110 = perform 16-bit Samsung sequence
 else: do not perform any device sequence

OTP_EBIU_AMG

Value to be written to `EBIU_AMGCTL` register

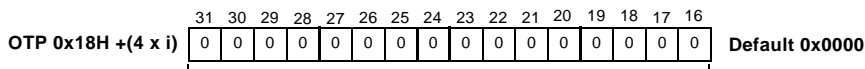
OTP_EBIU_MODE. Value to be written to the `EBIU_MODE` register

Figure 17-48. Upper PBS00 Half Page (PBS00H, Bits 63-32)

OTP Memory Pages for Booting

Upper PBS00 Half Page (PBS00H, Lower 31-16)

One-Time Programmable

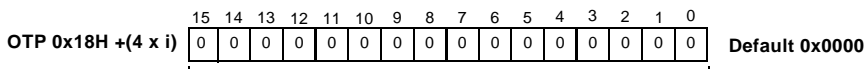


OTP_EBIU_FCTL

Value to be written to the EBIU_FCTL register if
OTP_SET_FCTL=1

Upper PBS00 Half Page (PBS00H, Lower 15-0)

One-Time Programmable



OTP_EBIU_A CTL

Value to be written to the EBIU_A CTL0 and
EBIU_A CTL1 registers. Applies only to banks
as enabled in the OTP_EBIU_AMG value.

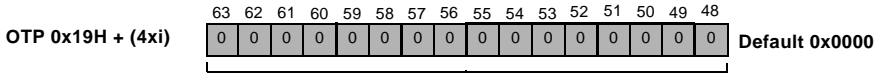
Figure 17-49. Upper PBS00 Half Page (PBS00H, Bits 31–0)

Upper PBS01 Half Page

The preboot routine loads the upper 64-bit half of page 0x19 only if either the `OTP_LOAD_PBS01H` bit in the `PBS00L` page is set. This page allows the user to disable boot modes. If a disabled boot mode configuration is chosen by the `BMODE[3:0]` pins, the boot kernel goes into idle state. This half pages also provides customization of the NAND flash controller. In OTP boot mode, this pages determines where in OTP memory the boot stream resides.

Upper PBS01 Half Page (PBS01H, Upper 63-48)

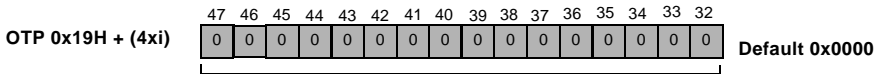
One-Time Programmable



Bits 63-48— Reserved

Upper PBS01 Half Page (PBS01H, Upper 47-32)

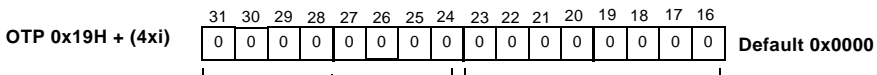
One-Time Programmable



Bits 47-32— Reserved

Upper PBS01 Half Page (PBS01H, Lower 31-16)

One-Time Programmable



OTP_START_PAGE
OTP start page for OTP boot mode. If 0x00, OTP boot starts at OTP page 0x40.

OTP_NFC_CTL
If non-zero value is written to lower eight bits of NFC_CTL register.

Figure 17-50. OTP Half Page (PBS01H, Bits 63–16)

OTP Memory Pages for Booting

Upper PBS01 Half Page (PBS01H, Lower 15-0)

One-Time Programmable

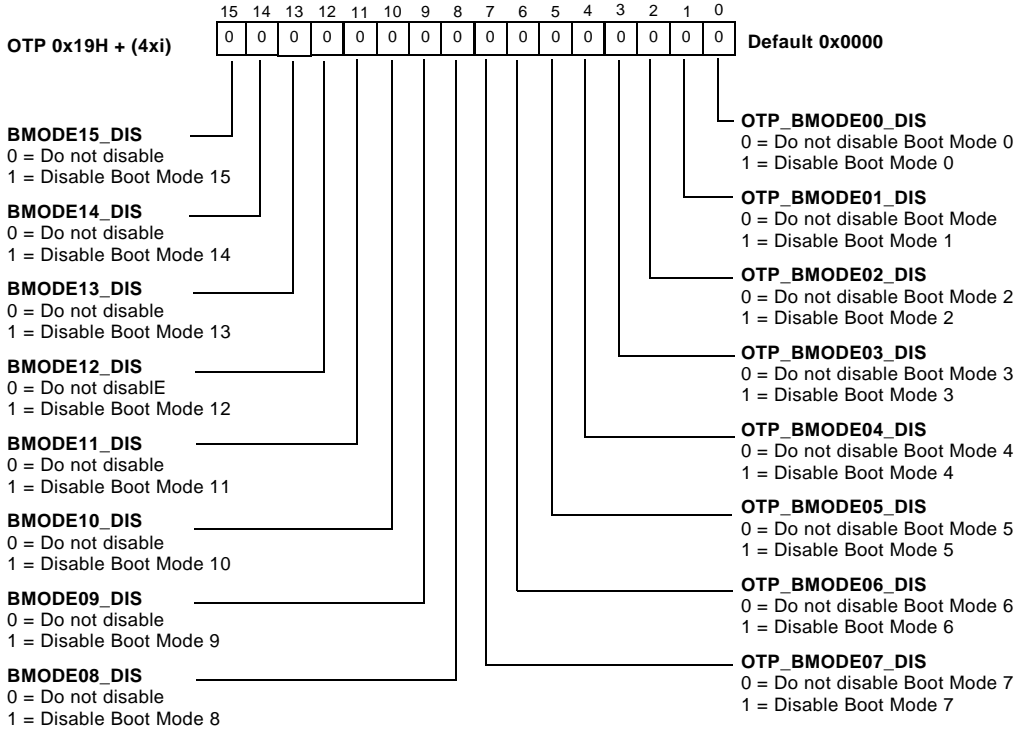


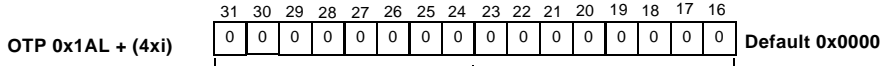
Figure 17-51. OTP Half Page PBS01H (PBS01H, Bits 15-0)

Lower PBS02 Half Page

The preboot routine loads the lower 64-bit half of page 0x1A only if the `OTP_LOAD_PBS02L` bit in half page `PBS00L` is set. Half pages `PBS02L` and `PBS02H` customize the SDRAM controller settings.

Lower PBS02 Half Page (PBS02L, Lower 31:16)

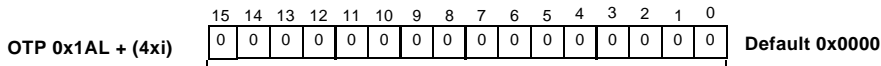
One-Time Programmable



OTP_EBIU_DDRCTL1[31:16] _____
 Values to be written to the EBIU_DDRCTL1 register

Lower PBS02 Half Page (PBS02L, Lower 15:0)

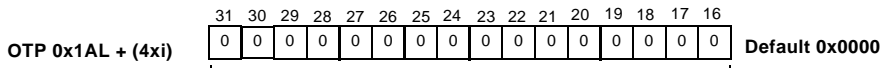
One-Time Programmable



OTP_EBIU_DDRCTL1[15:0] _____

Lower PBS02 Half Page (PBS02L, Lower 31:16)

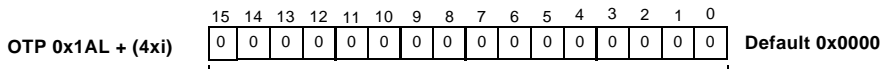
One-Time Programmable



OTP_EBIU_DDRCTL0[31:16] _____
 Values to be written to the EBIU_DDRCTL0 register

Lower PBS02 Half Page (PBS02L, Lower 15:0)

One-Time Programmable



OTP_EBIU_DDRCTL0[15:0] _____

Figure 17-52. ADSP-BF54x Lower PBS02 Half Page (PBS02L, Bits 63–16)

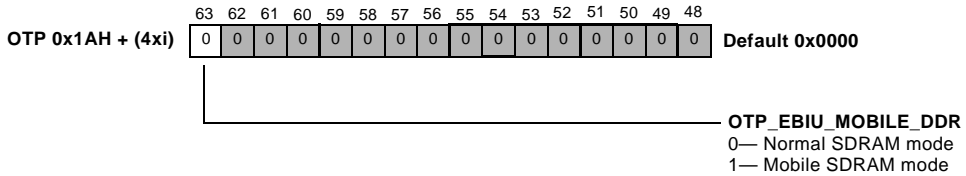
OTP Memory Pages for Booting

Upper PBS02 Half Page

The preboot routine loads the upper 64-bit half of page 0x16 only if the `OTP_LOAD_PBS02H` bit in the `PBS00L` page is set. Half pages `PBS02L` and `PBS02H` customize the SDRAM controller settings.

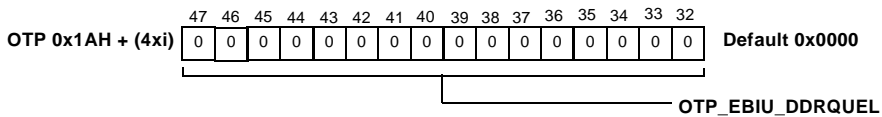
Upper PBS02 Half Page (PBS02H, Upper 63-48)

One-Time Programmable



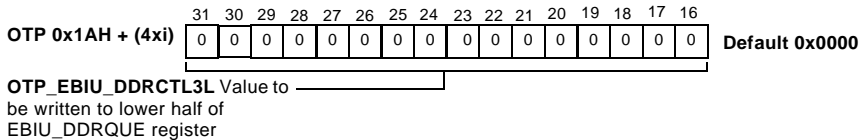
Upper PBS02 Half Page (PBS02H, Upper 47-32)

One-Time Programmable



Upper PBS02 Half Page (PBS02H, Lower 31-16)

One-Time Programmable



Upper PBS02 Half Page (PBS02H, Lower 15-0)

One-Time Programmable

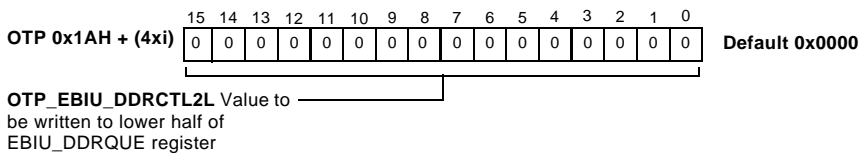



Figure 17-53. Upper PBS02 Half Page (PBS02H, Bits 63–0)

Reserved Half Pages

The half pages PBS01L, PBS03L and PBS03H are reserved and not used in the current silicon.

 Do not use these pages as they may be populated in future silicon revisions.

Data Structures

The boot kernel uses specific data structures for internal processing. Advanced users can customize the booting process by changing the content of the structure within the initcode routines. This section uses C language definitions for documentation purposes. VisualDSP++ users can use these structures directly in assembly programs by using the `.IMPORT` directive. The structures are supplied by the `bfrom.h` header file in your VisualDSP++ installation directory.

ADI_BOOT_HEADER

The structure `ADI_BOOT_HEADER` is used by the boot kernel to load and process a block header.

```
typedef struct {
    s32  dBlockCode;
    void* pTargetAddress;
    s32  dByteCount;
    s32  dArgument;
} ADI_BOOT_HEADER;
```

Every block header is loaded to L1 data memory location `0xFF80 7FF0–0xFF80 7FFF` first or where `pHeader` points to. There it is analyzed by the boot kernel.

ADI_BOOT_BUFFER

The structure `ADI_BOOT_BUFFER` is used for any kind of buffer. For the user, this structure is important when implementing advanced callback mechanisms.

```
typedef struct {
    void* pSource;
    s32   dByteCount;
} ADI_BOOT_BUFFER;
```

ADI_BOOT_DATA

The structure `ADI_BOOT_DATA` is the main data structure. A pointer to a `ADI_BOOT_DATA` structure is passed to most complex subroutines, including load functions, `initcode`, and callback routines. The structure has two parts. While the first is closely related to internal memory load routines, the second provides access to global boot settings.

```
typedef struct {
    void* pSource;
    void* pDestination;
    s16* pControlRegister;
    s16* pDmaControlRegister;
    s32   dControlValue;
    s32   dByteCount;
    s32   dFlags;
    s16   uwDataWidth;
    s16   uwSrcModifyMult;
    s16   uwDstModifyMult;
    s16   uwHwait;
    s16   uwSsel;
    s16   uwUserShort;
    s32   dUserLong;
    s32   dReserved2;
```

Data Structures

```
ADI_BOOT_ERROR_FUNC* pErrorFunction;
ADI_BOOT_LOAD_FUNC* pLoadFunction;
ADI_BOOT_CALLBACK_FUNC* pCallBackFunction;
ADI_BOOT_HEADER* pHeader;
void* pTempBuffer;
void* pTempCurrent;
s32 dTempByteCount;
s32 dBlockCount;
s32 dClock;
void* pLogBuffer;
void* pLogCurrent;
s32 dLogByteCount;
} ADI_BOOT_DATA;
```

[Table 17-17 on page 17-128](#) describes the data structures.

Table 17-17. Structure Variables, ADI_BOOT_DATA

Variable	Description
pSource	In the context of the boot kernel, the pSource pointer points either to the start address of the entire boot stream or to the header of the next boot block. In the context of memory load routines pSource points to the source address of the DMA work unit.
pDestination	The pDestination pointer is only used in memory load routines. It points to the destination address of the DMA work unit. It points to either 0xFF80 7FF0 when a header is loaded, or the target address when the payload data is loaded.
pControlRegister	This pointer holds the MMR address of the peripheral's main control register (for example UARTx_LCR or SPIx_CTL)
pDmaControlRegister	This pointer holds the MMR address of the DMAx_CONFIG register for the DMA channel in use.
dControlValue	The lower 16 bits of this value are written to the pControlRegister location each time a DMA work unit is started.
dByteCount	Number of bytes to be transferred.

Table 17-17. Structure Variables, ADI_BOOT_DATA (Cont'd)

Variable	Description
dFlags	The lower 16 bits of this variable hold the lower 16 bits of the current block code. The upper 16 bits hold global flags. See “ dFlags Word ” on page 17-131 .
uwDataWidth	This instructs the memory load routine to use: 0 = 8-bit DMA 1 = 16-bit DMA 2 = 32-bit DMA
uwSrcModifyMult	This is the multiplication factor used by the DMA source. A value of 1 sets the source modifier to 1 for 8-bit DMA, 2 for 16-bit DMA, or 4 for 32-bit DMA.
uwDstModifyMult	This is the multiplication factor used by the DMA destination. A value of 1 sets the destination modifier to 1 for 8-bit DMA, 2 for 16-bit DMA, or 4 for 32-bit DMA.
uwHwait	This 16-bit value holds the GPIO used for HWAIT signaling. The value can change on the fly. The upper eight bits designate the port number (for example 01 for Port A, 02 for Port B). The lower four bits designate the GPIO in the port. For example, GPIO PH11 has a value of 0x020B, PB7 has a value of 0x0807, PG0 has a value of 0x0700.
uwSsel	This 16-bit value holds the GPIO used for SPI slave select. The value can change on the fly. The upper eight bits designate the port number (for example 01 for Port A, 02 for Port B). The lower four bits designate the GPIO in the port.
uwUserShort	The programmer can use this 16-bit value for passing parameters between modules of a customized booting scheme.
dUserLong	The programmer can use this 32-bit value for passing parameters between modules of a customized booting scheme.
dReserved	This 32-bit value is reserved for future development.
pErrorFunction	This is the pointer to the error handler. See “ Error Handler ” on page 17-48 .
pLoadFunction	This is the pointer to the function responsible for loading data. See “ Load Functions ” on page 17-49
pCallbackFunction;	This is the pointer to the callback function. See “ Callback Routines ” on page 17-46
pHeader	The pHeader pointer holds the address for intermediate storage of the block header. By default this value is set to 0xFF80 7FF0.

Data Structures

Table 17-17. Structure Variables, ADI_BOOT_DATA (Cont'd)

Variable	Description
pTempBuffer	This pointer tells the boot kernel what memory to use for intermediate storage when the BFLAG_INDIRECT flag is set for a given block. The pointer defaults to 0xFF90 7E00. The value can be modified by the initcode routine, but there would be some impact to the VisualDSP++ tools.
pTempCurrent	Defaults to the pTempBuffer value. A load function can modify this value to manipulate subsequent callback and memory DMA routines.
dTempByteCount	This is the size of the intermediate storage buffer used when the BFLAG_INDIRECT flag is set for a given block. This value defaults to 256 and can be modified by an initcode routine. When increasing this value, the pTempBuffer must also be changed since by default the block is at the end of a physical data memory block.
dBlockCount	This 32-bit variable counts the boot blocks that are processed by the boot kernel. If the user sets this value to a negative value, the boot kernel exits when the variable increments to zero.
dClock	The dClock variable holds information about the clock divider used by individual (serial) boot modes.
pLogBuffer	Pointer to the circular log buffer. By default the log buffer resides in L1 scratch pad memory at address 0xFFB0 0400.
pLogCurrent	Pointer to the next free entry of the circular log buffer.
dLogByteCount	Size of the circular log buffer, default is 0x400 bytes.

dFlags Word

Figure 17-54 and Figure 17-55 on page 17-132 describe the dFlags word. dFlags [15-0] is a copy of Block Code[15-0] of the block currently being processed.

dFlags Word, 15-0

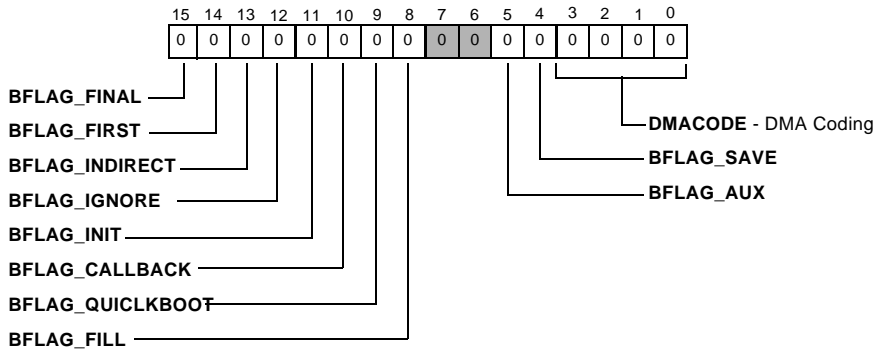


Figure 17-54. dFlags Word (Bits 15–0)

Data Structures

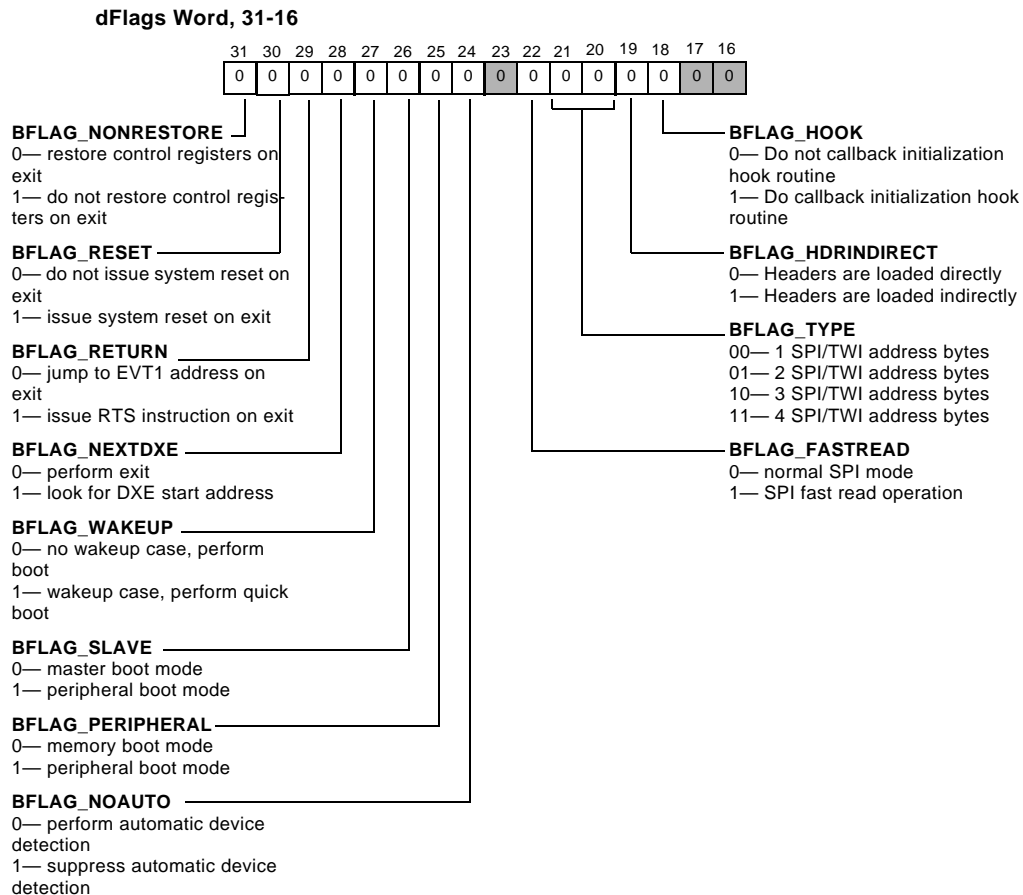


Figure 17-55. dFlags Word (Bits 31–16)

ADI_BOOT_NAND

The boot kernel makes use of a number of data structures for internal processing. Advanced users may manipulate some of contents of the structures from within initcode routines to customize the boot process further.

ADI_BOOT_NAND is the central structure used solely by the NAND boot kernel. The pointer to ADI_BOOT_NAND is stored in the dUserLong parameter of ADI_BOOT_DATA when NAND flash boot mode is enabled. This pointer provides access to the ADI_BOOT_NAND structure through initialization routines in order to further customize the booting process.

```
typedef struct{
    ADI_BOOT_NAND_DEVICE DeviceInfo;
    ADI_BOOT_NAND_BUFFER MainBuffer;
    ADI_BOOT_NAND_BUFFER PrefetchBuffer;
    ADI_BOOT_NAND_ACCESS AddressRequested;
    ADI_BOOT_NAND_ADDRESS AddressCycles;
    ADI_BOOT_NAND_ECC EccParity;
    ADI_BOOT_DATA *pBootData;
    void *pReserved;
} ADI_BOOT_NAND;
```

Table 17-18. Structure Variables, ADI_BOOT_NAND

Variable	Description
DeviceInfo	Properties relating to the NAND flash device
MainBuffer	Information relating to the current contents of the Main-Buffer.
PrefetchBuffer	Information relating to the current contents of Prefetch-Buffer.

Data Structures

Table 17-18. Structure Variables, ADI_BOOT_NAND

Variable	Description
AddressRequested	Contains details of the requested address when the address is converted to an address suitable for accessing the NAND flash.
AddressCycles	Contains information required to correctly read from the NAND flash device.
EccParity	Provides storage for the error correction parity data for a NAND flash page and controls the operation mode of the NAND boot kernel.
pBootData	Pointer to the global ADI_BOOT_DATA structure.
pReserved	Reserved for future enhancements. Do not use.

ADI_BOOT_NAND_DEVICE

This structure provides a number of details about the NAND flash device connected to the NFC. For booting from supported small page NAND flash devices not all parameters are used and thus initialized. For supported large page NAND flash memories, the structure is initialized after reading the electronic signature of the device. The 4th byte of the 4 byte electronic signature contains information that allows for the initialization of the entire structure.

```
typedef struct{
    u32  udIdCode;
    u32  udIdType;
    u16  uwBusWidth;
    u16  uwColumnMaskCount;
    u32  udColumnMask;
    u16  uwPageMaskCount;
    u32  udPageMask;
    u16  uwSpareMaskCount;
    u16  uwSpareAreaBit;
    u32  udBlockSize;
    u16  uwPageSize;
```



```

    u16  uwPagesPerBlock;
    u16  uwSpareAreaSize;
    u16  uwSpareAreaModifier;
    u16  uwNFCPages;
} ADI_BOOT_NAND_DEVICE;

```

Table 17-19. Structure Variables, ADI_BOOT_NAND_DEVICE

Variable	Description
udIdCode	The electronic signature of the device as received after issuing the 'Read Electronic Signature' Command. This is not populated if a small page device is detected as we only support a single small page type. Only used for large page NAND flash devices.
udType	Indicates a small page device '0' or a large page device '1'.
uwBusWidth	Bus width of the device '0' for 8-bit '1' for 16-bit.
uwColumnMaskCount	Number of bits required to address all columns within a NAND flash page (excluding the spare area). This is used to translate the address pSource provided in ADI_BOOT_DATA to a format required to address the NAND flash device.
udColumnMask	Used to extract the column within a page being addressed from the requested source address.
uwPageMaskCount	Number of bits required to address all pages within a single NAND flash block.
udPageMask	Used to extract the page number within a block being addressed from the source address.
uwSpareMaskCount	Number of bits required to address all columns within the spare area section at the end of a NAND flash page.
uwSpareAreaBit	This contains the bit position that needs to be set in order to address the spare area section of the NAND flash page.
udBlockSize	The block size of the device in bytes (excluding spare area).
uwPageSize	The page size of the device in bytes (excluding the spare area).
uwPagesPerBlock	The number of pages within a block.

Data Structures

Table 17-19. Structure Variables, ADI_BOOT_NAND_DEVICE

Variable	Description
uwSpareAreaSize	The number of bytes contained within the spare area section of a page.
uwSpareAreaModifier	The number of bytes in the spare area section dedicated for each 256 byte NAND flash controller page.
uwNFCPages	The number of 256 byte NAND Flash controller pages within a full NAND flash page.

ADI_BOOT_NAND_BUFFER

The `ADI_BOOT_NAND_BUFFER` structure provides details of the current contents of a 256 byte buffer. There are two of these buffers required for NAND boot. The buffer provides details on the location of the buffer as well as its current contents. As 256 byte blocks of data are read from the NAND flash memory at a time, the kernel can determine if a new data fetch is required from the NAND flash or whether the data resides in one of the two buffers located in internal memory.

```
typedef struct{
    void * pBegin;
    u16  uwLoadedNFCPage;
    u16  uwLoadedNANDPage;
    u16  uwLoadedNANDBlock;
} ADI_BOOT_NAND_BUFFER;
```

Table 17-20. Structure Variables, ADI_BOOT_NAND_BUFFER

Variable	Description
pBegin	Pointer to the first address of a 256 byte buffer.
uwnLoadedNFCPage	The currently loaded 256 byte NAND flash controller sub-page.

Table 17-20. Structure Variables, ADI_BOOT_NAND_BUFFER

Variable	Description
uwLoadedNANDPage	The currently loaded NAND flash page.
uwLoadedNANDBlock	The currently loaded NAND flash block.

ADI_BOOT_NAND_ACCESS

The source address provided by the main kernel is analyzed and based on the contents of `ADI_BOOT_NAND_DEVICE` the 256 byte block within a page, the actual page and block in which the data resides can be calculated. This structure contains these access details and in conjunction with the `ADI_BOOT_NAND_BUFFER` structures is used to determine if the data needs to be fetched from the NAND flash memory or whether it already resides in internal memory.

```
typedef struct{
    u16  uwAccessNFCPage;
    u16  uwAccessNANDPage;
    u16  uwAccessNANDBlock;
} ADI_BOOT_NAND_ACCESS;
```

Table 17-21. Structure Variables, `ADI_BOOT_NAND_BUFFER`

Variable	Description
<code>uwAccessNFCPage</code>	The requested 256 byte NAND flash controller sub-page to be accessed
<code>uwAccessNANDPage</code>	The requested NAND flash page to be accessed.
<code>uwAccessNANDBlock</code>	The requested NAND flash block to be accessed.

ADI_BOOT_NAND_ADDRESS

ADI_BOOT_NAND_ADDRESS is modified when the NAND boot kernel decodes the source address provided by the main kernel. Any offsets are applied that enable addressing of alternative blocks in the event that one of the booting features is used that requires the detection of bad blocks or uncorrectable errors when performing the error correction. When the address is decode the structure is filled with the required NAND flash controller commands and address cycles to be issued in order retrieve the required data.

For supported small page NAND flash devices, the number of address cycles is always 4 and the number of command cycles is 1. For large page NAND flash devices, the number of address cycles default value is 5. This is due to the fact that the upper addressing boundaries of the NAND flash device cannot be determined from the electronic signature, in turn the kernel is unable to calculate the exact amount of address cycles required to be issued in order to perform a read from the NAND flash. A majority of large page NAND flash devices simply ignore any additional address cycles on a page read command that are not required. If a NAND flash device is not capable of ignoring the additional address cycles and it requires less than the default 5 address cycles for a page read operation then the device cannot be supported for NAND boot functionality. The number of address cycles required can be re-configured within an initialization file executed before the loading of the main application takes place in order to remove the redundant address cycles.

```
typedef struct{

    void *pSource;
    u32 udMainOffset;
    u32 udPrefetchOffset;
    u16 uwNumAddressCycles;
```

Data Structures

```
    u16 uwNumCommands;  
    u16 uwSerialAccess;  
    ADI_BOOT_NAND *pNandInfo  
    #pragma align 4  
    u8 ubCommand0;  
    u8 ubAddress0;  
    u8 ubAddress1;  
    u8 ubAddress2;  
    u8 ubAddress3;  
    u8 ubAddress4;  
    u8 ubCommand1;  
} ADI_BOOT_NAND_ADDRESS;
```

Table 17-22. Structure Variables, ADI_BOOT_NAND_ADDRESS

Variable	Description
pSource	The source address to be accessed.
udMainOffset	The current block offset applied to data loaded into the main buffer.
udPrefetchOffset	The current block offset applied to data loaded into the prefetch buffer.
uwNumAddressCycles	The number of address cycles required to access the NAND flash device. This is set to 4 for small page device booting and 5 for large page devices.
uwNumCommands	The number of command cycles required to perform a read access from the NAND flash device. This parameter is set to 1 for small page devices and 2 for large page devices.
uwSerialAccess	Indicates that the next read access is from the next sequential 256 byte page to the previous access. This allows for the removal of the issuing of a read transaction thus optimizing throughput without waiting on unnecessary ready/#busy assertions.
pNandInfo	Pointer to ADI_BOOT_NAND structure
ubCommand0	The first command to be issued to perform a page read from the NAND flash device.

Table 17-22. Structure Variables, ADI_BOOT_NAND_ADDRESS

Variable	Description
ubAddress0	The first address cycle issued when performing a page read command.
ubAddress1	The second address cycle issued when performing a page read command.
ubAddress2	The third address cycle issued when performing a page read command.
ubAddress3	The fourth address cycle issued when performing a page read command.
ubAddress4	The fifth address cycle issued when performing a page read command.
ubCommand1	The second command to be issued to perform a page read from the NAND flash device. Only used for large page devices.

ADI_BOOT_NAND_ECC

This structure provides stack storage for the error correction parity data read from the spare area of a page when an access to a new NAND flash page is detected. The spare area contains parity data for each 256 byte block in a page. This allows for error correction and detection to be performed on every 256 byte load from the NAND flash. Enough storage space is provided to support devices up to an including a page size of 8Kbytes. In addition to the parity data, ADI_BOOT_NAND_ECC also contains the fields that need to be modified in order to enable the more advance

Data Structures

NAND boot options that allow for the skipping of bad blocks and booting from mirror images of the original boot stream that may be located in other memory blocks.

```
typedef struct{  
    #pragma align 4  
    u16 uwIndex;  
    u32 udNFCEParity[32];  
    u16 uwError;  
    u16 uwBlockSkipFeature;  
    u16 uwBlockModifier;  
    u16 uwMaxCopies;  
    u16 uwCurrentCopy;  
ADI_BOOT_NAND_ECC;
```

Table 17-23. Structure Variables, ADI_BOOT_NAND_ECC

Variable	Description
nIndex	Index used to access the udNFCEParityArray
udNFCEParity	A 32 deep long word array providing storage for up to 32 256-byte NAND Flash Controller error correction parity data. The array provides support for page sizes up to and including 8 Kbytes.
uwError	Error that was generated within the error correction routine. 0 = No Error, 1 = Error found in parity data, 2 = Uncorrectable error

Table 17-23. Structure Variables, ADI_BOOT_NAND_ECC

Variable	Description
uwBlockSkipFeature	Specifies the NAND Boot technique to be implemented. Defaults to 0 unless otherwise altered through an initialization sequence. 0 = Sequential booting from a single boot stream. No bad block checking performed. 1 = Block Skip Method, allowing for a single boot stream loaded to the NAND flash to skip bad blocks. 2 = Mirror Image Mode, allowing for booting from multiple copies of the application in the event that an uncorrectable error or error in the ECC parity data is detected.
uwError	Indicates the error returned from the error correction routine if one occurred. 0 = No error or correctable error. 1 = Error in ECC parity data. 2 = Uncorrectable error.
uwBlockModifier	The number of blocks to skip if a bad block is detected. If uwBlockSkipFeature is 0 this value is ignored. For an uwBlockSkipFeature value of 1 this parameter must be 1. For an uwBlockSkipFeature of 2 this parameter may be any value indicating the number of blocks between multiple copies of the application.
uwMaxCopies	The number of copies of the application stored in the NAND flash device. Only applicable if nBlockSkipFeature is 2.
uwCurrentCopy	Indicates the current copy of the application that is being accessed. Only applicable if nBlockSkipFeature is 2.

Callable ROM Functions for Booting

The following functions support boot management.

BFROM_FINALINIT

Entry address: 0xEF00 0002

Arguments: no arguments

C prototype: `void bfrom_FinalInit (void);`

The final init function never returns. It only executes a JUMP to the address stored in EVT1.

BFROM_PDMA

Entry address: 0xEF00 0004

Arguments: pointer to ADI_BOOT_DATA in R0

C prototype: `void bfrom_PDma (ADI_BOOT_DATA *p);`

This is the load function for peripherals such as SPI and UART that support DMA in their boot modes.

BFROM_MDMA

Entry address: 0xEF00 0006

Arguments: pointer to ADI_BOOT_DATA in R0

C prototype: `void bfrom_MDma (ADI_BOOT_DATA *p);`

This is the load function used for memory boot modes including the FIFO mode. This routine is also reused when the `BFLAG_FILL` or the `BFLAG_INDIRECT` flags are specified.

BFROM_MEMBOOT

Entry address: `0xEF00 0008`

Arguments:

pointer to boot stream in `R0`

`dFlags` in `R1`

`dBlockCount` in `R2`

`pCallHook` passed over the stack in `[FP+0x14]`

updated block count returned in `R0`

C prototype:

```
s32 bfrom_MemBoot (void* pBootStream, s32 dFlags, s32 dBlock-  
Count, ADI_BOOT_HOOK_FUNC* pCallHook);
```

This routine processes any boot stream that maps to the Blackfin memory starting from address `pBootStream`.

To boot a new application that may overwrite the calling application, the `dFlags` word is usually zero. When done, the routine does not return, but jumps to the `EVT1` vector address. If the `BFLAG_RETURN` flag is set, an RTS is executed instead and the routine returns to the parent function. In this way, fractions of an application can be loaded.

If the `dBlockCount` parameter is zero or a positive value, all boot blocks are processed until the `BFLAG_FINAL` flag is detected. If `dBlockCount` is a negative value, the negative number represents the number of blocks to be booted. For example, `-1` causes the kernel to return immediately, `-2` processes only one block.

Callable ROM Functions for Booting

The routine returns the updated source address `pSource` of the boot stream (for example, the first unused address after the processed boot stream).

The `BFLAG_NEXTDXE` flag suppresses boot loading. The boot kernel steps through the boot stream by analyzing the next-DXE pointers (in the `ARGUMENT` field of a `BFLAG_FIRST` block) and jumping to the next DXE.

Assuming that the boot image is a chained list of boot streams, the boot kernel returns the absolute start address of the requested boot stream. In this example, the start address of the third boot stream (DXE) in a flash device is returned.

```
bfrom_MemBoot((void*)0x20000000,BFLAG_RETURN|BFLAG_NEXTDXE,-3,
NULL);
```

In the above example, the routine would return `0x2000 0000` when `dBlockCount` was set to `-1`. If the parameter `dBlockCount` is zero or positive when used with along with the `BFLAG_NEXTDXE` command, the kernel returns when the `BFLAG_FIRST` flag on a header in the next-DXE chain is not set.

If the `BFLAG_HOOK` switch is set, the `memboot` routine call (`pCallHook` routine) after the `ADI_BOOT _DATA` structure is filled with default values. It then can overrule the default settings of the structure.

BFROM_TWIBOOT

Entry address: `0xEF00 000C`

Arguments:

TWI address in R0

dFlags in R1

dBlockCount in R2

pCallHook passed over the stack in [FP+0x14]

updated block count returned in R0

C prototype:

```
s32 bfrom_TwiBoot (s32 dTwiAddress, s32 dFlags,  
s32 dBlockCount, ADI_BOOT_HOOK_FUNC* pCallHook);
```

This routine processes boot streams residing in TWI memories, using the TWI0 controller. It differs from the BFROM_MEM00T routine in that some functionality is TWI specific.

Additional bits in the dFlags word are relevant. The user should always set the BFLAG_PERIPHERAL flag but never the BFLAG_SAVE bit. The BFLAG_TYPE tells the boot kernel when addressing mode is required for the TWI memory. The boot kernel derives the values for the TWI0_CONTROL and TWI0_CLKDIV registers from the lower four bits of the dFlags word. See [Chapter 29, “Two Wire Interface Controllers”](#).

BFROM_SPIBOOT

Entry address: 0xEF00 000A

Arguments:

SPI address in R0

dFlags in R1

dBlockCount in R2

pCallHook passed over the stack in [FP+0x14]

updated block count returned in R0

C prototype:

```
s32 bfrom_SpiBoot (s32 DSpiAddress, S32 dFlags, s32 dBlockCount,  
ADI_BOOT_HOOK_FUNC* pCallHook);
```

Callable ROM Functions for Booting

This SPI master boot routine processes boot streams residing in SPI memories, using the SPI0 controller. It differs from the `BFROM_TWIBOOT` routine in that some functionality is SPI specific. The fourth argument `pCallHook` is passed over the stack. It provides a hook to call a callback routine after the `ADI_BOOT_DATA` structure is filled with default values. For example, the `pCallHook` routine may overwrite the default value of the `uwSsel` value in the `ADI_BOOT_DATA` structure. The coding follows the rules of `uwHWAIT` (see [“Boot Host Wait \(HWAIT\) Feedback Strobe” on page 17-33](#)). A value of `0x0504` represents GPIO PE4 (SPI0 SEL1), `0x0505` for PE5 (SPI0 SEL2) and so on.

Additional bits in the `dFlags` word are relevant. The user should always set the `BFLAG_PERIPHERAL` flag but never the `BFLAG_SAVE` bit. The `BFLAG_NOAUTO` flag instructs the system to skip the SPI device detection routine. The `BFLAG_TYPE` then tells the boot kernel what addressing mode is required for the SPI memory. (see [“SPI Device Detection Routine” on page 17-72](#)). The `BFLAG_FASTREAD` flag controls whether standard SPI read (`0x3` command) or fast read (`0xB`) is performed. The boot kernel writes the lower bits of the `dFlags` word to the `SPI0_BAUD` registers.

BFROM_OTPBOOT

Entry address: 0xEF00 000E

Arguments:

OTP byte address in R0

dFlags in R1

dBlockCount in R2

pCallHook passed over the stack in [FP+0x14]

Updated block count returned in R0

C prototype: s32 bfrom_otpBoot (s32 d0tpAddress, S32 dFlags, s32 dBlockCount, ADI_BOOT_HOOK_FUNC* pCallHook);

This OTP boot routine processes boot streams residing in the on-chip, serial OTP memory. Unlike the `bfrom_otpRead()` function which uses the half-page addressing method, this one requires byte addressing. For example, set the `d0tpAddress` argument to `0x400` to process a boot stream starting from OTP page `0x40`. Remember that one OTP page spans 16 bytes.

BFROM_NANDBOOT

Entry address: 0xEF00 0010

Arguments:

NAND Flash address in R0

dFlags in R1

dBlockCount in R2

pCallHook passed over the stack in [FP+0x14]

updated block count returned in R0

C prototype: s32 bfrom_NandBoot(s32 dNandAddress, s32 dFlags, s32 dBlockCount, ADI_BOOT_HOOK_FUNC *pCallHook)

This NAND flash boot routine processes boot streams residing in NAND flash memories, using the NAND Flash Controller. Some functionality is NAND flash specific.

Additional bits in the dFlags word are relevant. When the BFLAG_NOAUTO flag is set the BFLAG_TYPE field is used to indicate whether the connected NAND flash is a small page or large page device.

BFLAG_TYPE 0 = Small Page NAND Flash

BLAG_TYPE 1 = Large Page NAND Flash

BFLAG_TYPE values of 2 and 3 are reserved.

Detection of a reserved value results in a call to the error handler.

In the event the `NFC_CTL` register is set to the default reset value of `0x0200` prior to the call to `bfrom_NandBoot()`, the read and write delay strobes of the `NFC_CTL` register will each be set to 3 providing t_{RP} and t_{WP} timings of 4 SCLK cycles.

BFROM_BOOTKERNEL

Entry address: `0xEF00 0020`

Arguments:

pointer to `ADI_BOOT_DATA` in `R0`

returns updated source address `pSource` in `R0`

C prototype: `s32 bfrom_BootKernel (ADI_BOOT_DATA *p);`

This ROM entry provides access to the raw boot kernel routine. It is the user's responsibility to initialize the items passed in the `ADI_BOOT_DATA` structure. Pay particular attention that the function pointers (`pLoadFunction`, and `pErrorFunction`) point to functional routines.

BFROM_CRC32

Entry address: `0xEF00 0030`

Arguments:

pointer to look-up table in `R0`

pointer to data in `R1`

`dByteCount` in `R2`

initial CRC value in `R0`

CRC value returned in `R0`

Callable ROM Functions for Booting

C prototype:

```
s32 bfrom_Crc32 (s32 *pLut, void *pData, s32 dByteCount,  
s32 dInitial);
```

This routine calculates the CRC32 checksum for a given array of bytes. The look-up table is typically generated by the `BFROM_CRC32POLY` routine. During the boot process this routine is called by the `BFROM_CRC32CALLBACK` routine. The `dInitial` value is normally set to zero unless the CRC32 routine is called in multiple slices. Then, the `dInitial` parameter expects the result of the former run.

BFROM_CRC32POLY

Entry address: 0xEF00 0032

Arguments:

- pointer to look-up table in R0
- polynomial in R1
- updated block count returned in R0

C prototype:

```
s32 bfrom_Crc32Poly (unsigned s32 *pLut, s32 dPolynomial);
```

This function generates a 1024-byte look-up table from a given CRC polynomial. During the boot process this routine is hidden by the `BFROM_CRC32INITCODE` routine.

BFROM_CRC32CALLBACK

Entry address: 0xEF00 0034

Arguments:

pointer to ADI_BOOT_BUFFER in R0

pointer to ADI_BOOT_BUFFER in R1

C prototype: `s32 bfrom_Crc32Callback (ADI_BOOT_DATA *pBS,
ADI_BOOT_BUFFER *pCS);`

This is a wrapper function that ensures the BFROM_CRC32 subroutine fits into the boot process.

BFROM_CRC32INITCODE

Entry address: 0xEF00 0036

Arguments: pointer to ADI_BOOT_DATA in R0.

C prototype: `void bfrom_Crc32Initcode (ADI_BOOT_DATA *p);`

This is an initcode residing in ROM with two jobs. Register BFROM_CRC32CALLBACK as a callback routine to the pCallback pointer in ADI_BOOT_DATA. Call BFROM_CRC32POLY to generate the look-up table.

This function is unlikely to be called by user code directly. This function is called as an initcode during the boot process when the CRC calculation is desired. See [“CRC Checksum Calculation” on page 17-49](#) for details.

Programming Examples

System Reset

To perform a system and core reset, use the code shown in [Listing 17-2](#) or [Listing 17-3](#).

Listing 17-1. System Reset in assembly language

```
#include <blackfin.h>

P0.L = LO(BFROM_SYSCONTROL);
P0.H = HI(BFROM_SYSCONTROL);
R0.L = LO(SYSCTRL_SYSRESET);
R0.H = HI(SYSCTRL_SYSRESET);

R1 = 0;
R2 = 0;

CALL (P0);
```

Listing 17-2. System Reset in C language

```
bfrom_SysControl(SYSCTRL_SYSRESET, 0, NULL);
```

Exiting Reset to User Mode

To exit reset while remaining in user mode, use the code shown in [Listing 17-3](#).

Listing 17-3. Exiting Reset to User Mode

```

_reset:
    P1.L = L0(_usercode) ; /* Point to start of user code */
    P1.H = HI(_usercode) ;
    RETI = P1 ;           /* Load address of _start into RETI */
    RTI ;                 /* Exit reset priority */
_reset.end:
_usercode:                /* Place user code here */
...

```

The reset handler most likely performs additional tasks not shown in the examples above. Stack pointers and EVT_x registers are initialized here.

Exiting Reset to Supervisor Mode

To exit reset while remaining in supervisor mode, use the code shown in [Listing 17-4](#).

Listing 17-4. Exiting Reset by Staying in Supervisor Mode

```

_reset:
    P0.L = L0(EVT15) ; /* Point to IVG15 in Event Vector Table */
    P0.H = HI(EVT15) ;
    P1.L = L0(_isr_IVG15) ; /* Point to start of IVG15 code */
    P1.H = HI(_isr_IVG15) ;
    [P0] = P1 ;        /* Initialize interrupt vector EVT15 */

```

Programming Examples

```
P0.L = LO(IMASK) ; /* read-modify-write IMASK register */
R0 = [P0] ; /* to enable IVG15 interrupts */
R1 = EVT_IVG15 (Z);
R0 = R0 | R1 ; /* set IVG15 bit */
[P0] = R0 ; /* write back to IMASK */

RAISE 15 ; /* generate IVG15 interrupt request */
/* IVG 15 is not served until reset
   handler returns */

P0.L = LO(_usercode) ;
P0.H = HI(_usercode) ;
RETI = P0 ; /* RETI loaded with return address */
RTI ; /* Return from Reset Event */
_reset.end:

_usercode: /* Wait in user mode till IVG15 */

    JUMP _usercode; /* interrupt is serviced */
_isr_IVG15: /* IVG15 vectors here due to EVT15 */
...

```

Initcode (SDRAM Controller Setup)

[Listing 17-1](#) shows an example of initcode to setup the SDRAM controller. The SDRAM controller must be initialized *before* data can be booted into it. Therefore, the SDRAM controller is typically initialized by an initcode or by the preboot functionality. The following initcode example assumes that the preboot did not do the job.

Listing 17-5. Example Initcode (SDRAM Controller Setup)

```
#include <defBF548.h>
.section initcode;
/*****SDRAM Setup*****/
Setup_SDRAM:
/* save to stack following C conventions */
void initcode(ADI_BOOT_DATA* pBS)
{
    *pEBIU_RSTCTL |= DDRSRESET;
    *pEBIU_DDRCTL0 =
        SET_tRC(8)|
        SET_tRAS(6)|
        SET_tRP(2)|
        SET_tRFC(10)|
        SET_tREF(1041);

    *pEBIU_DDRCTL1 =
        SET_tWTR(2)|
        DDR_DEVSZ_512|
        DDR_DEVWIDTH_16|
        CS0|
        DDR_DATAWIDTH|
        SET_tWR(2)|
        SET_tMRD(2)|
```

Programming Examples

```
        SET_tRCD(2);

    *pEBIU_DDRCTL2 =
        nREGE|
        nDLLRESET|
        CASLATENCY2|
        BURSTLENGTH1|
        0;

}
```

Since this initcode need execute only once, it can be volatile and can be overwritten by other boot blocks.

Initcode (Power Management Control)

The following example shows how to change PLL and the voltage regulator within an initcode. The example assumes that the preboot did not do the job already.

Listing 17-6. Changing PLL and Voltage Regulator

```
#include <blackfin.h>

void initcode (ADI_BOOT_DATA* pBS)
{
    ADI_SYSCTRL_VALUES mystruct;
    mystruct.uwVrCtl = 0x ... ;
    mystruct.uwPl1Ctl = 0x ... ;
    mystruct.uwPl1Div = 0x ... ;

    bfrom_SysControl(SYSCTRL_VRCTL | SYSCTRL_INTVOLTAGE |
                    SYSCTRL_PLLCTL | SYSCTRL_PLLDIV |
                    SYSCTRL_WRITE,
                    &mystruct, NULL ) ;
}
```

Care must be taken that the reprogramming of the PLL does not break the communication with the booting host. For example, in the case of UART boot, the `UARTx_DLL` and `UARTx_DLH` registers must be updated to keep the old bit rate.

Initcode (NAND Boot Mode Configuration)

[Listing 17-7](#) shows an example of initcode to enable some of the more advanced options available with the NAND boot mode. The initcode is loaded in while the NAND boot kernel is configured for the default boot mode. After the initcode sequence is executed the NAND boot kernel in this example is in `Multiple Image Mode`. This example also alters the number of address cycles issued on all further accesses to optimize the boot kernel further for the attached NAND flash device.

Listing 17-7. Initcode options with NAND Boot Mode

```
#include <bfrom.h>
void initcode(ADI_BOOT_DATA* pBS)
{
    /* Create a pointer to the ADI_BOOT_NAND structure */
    ADI_BOOT_NAND *pNS;

    /* Set the pointer to ADI_BOOT_NAND */
    pNS = pBS->dUserLong;

    /* NAND Boot Kernel Configuration
       Mode:                               Multiple Image Mode
       Number of blocks between each image: 10
       Number of images:                   4
       Number of address cycles:           4
    */
    pNS->EccParity.uwBlockSkipFeature = 2;
    pNS->EccParity.uwBlockModifier = 10;
    pNS->EccParity.uwMaxCopies = 3;
    pNS->AddressCycles.uwNumAddressCycles = 4;
}
```

Quickboot With Restore From SDRAM

This example could be part of an advanced power saving concept. Assume the Blackfin is waking up from hibernate and processing any master boot mode. If the SDRAM has not been shut down, but was put in self-refresh mode, the content of the SDRAM will still be valid after wake up. The boot process would only have to initialize on-chip memories. Several boot blocks might be tagged by the `BFLAG_QUICKBOOT` flag.

Some applications might use a power-down handler that saves the contents of L1 memory to SDRAM before entering the hibernate state. [Listing 17-8](#) assumes a suitable power-down handler was present that generated a partial boot stream in SDRAM at address `0x0001 0000` containing all the instructions required to restore the L1 memory contents.

Listing 17-8. Quickboot With Restore From SDRAM

```
void L1_recovery_initcode (ADI_BOOT_DATA *pBS)
{
    if (pBS->dFlags & BFLAG_WAKEUP) {
        bfrom_MemBoot((void*)0x00010000, BFLAG_RETURN, NULL);
    }
}
```

The boot stream generated at `0x0001 0000` will only be processed upon a wake-up condition. The `BFLAG_RETURN` ensures that the new instance of the boot kernel returns to the `initcode` rather than jumps to the `EVT1` vector.

XOR Checksum

[Listing 17-9](#) illustrates how an initcode can be used to register a callback routine. The routine is called after each boot block that has the `BFLAG_CALLBACK` flag set. The calculated XOR checksum is compared against the block header argument field. When the checksum fails, this example goes into idle mode. Otherwise control is returned to the boot kernel.

Since this callback example accesses the data after it is loaded, it would fail if the target address were in L1 instruction space. Therefore the `BFLAG_INDIRECT` flag should also be set. The `xor_callback` routine could then perform the checksum calculation at an intermediate storage place. The boot kernel transfers the data from the temporary buffer to the final destination after the callback routine returns.

In general, the block size is bigger than the size of the temporary buffer. Therefore, the boot kernel may need to divide the processing of a single block into multiple steps. The callback routine may also need to be invoked multiple times—every time the temporary buffer is filled up and once for the remaining bytes. The boot kernel passes the `dFlags` parameter, so that the callback routines knows whether it is called the first time, the last time or neither. The `dUserLong` variable in the `ADI_BOOT_DATA` structure is used to store the intermediate results between function calls.

Listing 17-9. XOR Checksum

```
bool xor_callback(ADI_BOOT_DATA* pBS, ADI_BOOT_BUFFER* pCS, s32
dFlags)
{
    s32 i;
    if ((pCS != NULL) && (pBS->pHeader != NULL)) {
        if (dFlags & CBFLAG_FIRST) {
            pBS->dUserLong = 0;
        }
    }
}
```

Programming Examples

```
    for (i=0; i<pCS->dByteCount/sizeof(s32); i++) {
        pBS->dUserLong^= ((s32 *)pCS->pSource)[i];
    }
    if (dFlags & CBFLAG_FINAL) {
        if (pBS->dUserLong != pBS->pHeader->dArgument) {
            idle ();
        }
    }
}
return 0;
}

void xor_initcode (ADI_BOOT_DATA *pBS)
{
    pBS->pCallbackFunction = xor_callback;
}
```

Note that the callback routine is not volatile. It should not be overwritten by subsequent boot blocks. It can, however, be overwritten after processing the last block with `BFLAG_CALLBACK` flag set.

The checksum algorithm must be booted first and cannot protect itself. Problems can be avoided by letting initcode and callback execute directly from off-chip flash memory. The ADSP-BF54x processors provide a CRC32 checksum algorithm in the on-chip L1 instruction ROM, that can be used for booting under this scenario. [For more information, see “CRC Checksum Calculation” on page 17-49.](#)

Direct Code Execution

This code example illustrates how to instruct the VisualDSP++ tools to generate a flash image that causes the boot kernel to start code execution at flash address 0x2000 0020 rather than performing a regular boot. See [“Direct Code Execution” on page 17-37.](#)

First, a 32-byte data block is defined in an assembly file that contains the **initial block**.

```
.section bootblock;
.global _firstblock;
.var _firstblock[4] = 0xAD7BD006, 0x20000020, 0x00000010,
0x00000010;
```

Then, the linker is instructed to map the initial block to address **0x2000 0000** in the LDF file.

```
MEMORY
{
    MEM_ASYNC0
    {
        START(0x20000000)
        END(0x23FFFFFF)
        TYPE(ROM)
        WIDTH(8)
    }
}

PROCESSOR p0
{
    RESOLVE(_firstblock,0x20000000)
    RESOLVE(start,0x20000020)
    KEEP(start,_firstblock)
    SECTIONS
    {
        flash
        {
            INPUT_SECTION_ALIGN(4)
            INPUT_SECTIONS( $OBJECTS(program) $LIBRARIES(program))
            INPUT_SECTIONS( $OBJECTS(bootblock))
        } >MEM_ASYNC0
```

Programming Examples

```
    }  
}
```

To invoke the elfloader utility, activate the meminit feature and use the command-line switches `-romsplitter` and `-maskaddr`. Refer to the application note *Running Programs from Flash on ADSP-BF533 Blackfin Processors (EE-239)* for further details.

Managing PBS Pages in OTP Memory

The following code snips illustrate how to read and write OTP memory, as it is required for the Preboot Settings (PBS). For detailed description of OTP API functions `bfrom_OtpCommand()`, `bfrom_OtpRead()` and `bfrom_OtpWrite()` used here, see [Chapter 16, “One-Time Programmable Memory”](#).

The first example reads PBS settings from OTP and stores them into an instance of the `ADI_PBS_BLOCK` structure. This is an union composite of the `ADI_PBS_HALFPAGES` or the `ADI_PBS_BITFIELDS` types. These structure types are defined in the `bfrom.h` header file. The `dPbsSet` variable describes the set of PBS pages which is of interest. A `0x00` value reads from OTP pages `0x18` to `0x1B`. A `0x01` value reads from OTP pages `0x1C` to `0x1F` and so on.

Listing 17-10. Reading a set of PBS Pages from OTP Memory

```
#include <blackfin.h>
#include <bfrom.h>
ADI_PBS_BLOCK PBS;
u32 dPbsSet = 0;
bfrom_OtpRead(PBS00+dPbsSet*4,OTP_LOWER_HALF,&(PBS.HalfPages.uqPbs00L));
bfrom_OtpRead(PBS00+dPbsSet*4, OTP_UPPER_HALF,&(PBS.HalfPages.uqPbs00H));
bfrom_OtpRead(PBS01+dPbsSet*4,OTP_LOWER_HALF,&(PBS.HalfPages.uqPbs01L));
bfrom_OtpRead(PBS01+dPbsSet*4,OTP_UPPER_HALF,&(PBS.HalfPages.uqPbs01H));
bfrom_OtpRead(PBS02+dPbsSet*4,OTP_LOWER_HALF,&(PBS.HalfPages.uqPbs02L));
```

The next example shows how PBS pages can be written:

Listing 17-11. Programming a set of PBS Pages from OTP Memory

```
#include <blackfin.h>
#include <bfrom.h>
ADI_PBS_BLOCK PBS;
u32 dPbsSet = 0;
/* fill PBS with meaningful data */
bfrom_OtpWrite(PBS00+dPbsSet*4, OTP_LOWER_HALF |
OTP_CHECK_FOR_PREV_WRITE,&(PBS.HalfPages.uqPbs00L));
bfrom_OtpWrite(PBS00+dPbsSet*4, OTP_UPPER_HALF |
OTP_CHECK_FOR_PREV_WRITE,&(PBS.HalfPages.uqPbs00H));
bfrom_OtpWrite(PBS01+dPbsSet*4, OTP_LOWER_HALF |
OTP_CHECK_FOR_PREV_WRITE,&(PBS.HalfPages.uqPbs01L));
bfrom_OtpWrite(PBS01+dPbsSet*4, OTP_UPPER_HALF |
OTP_CHECK_FOR_PREV_WRITE,&(PBS.HalfPages.uqPbs01H));
```

Programming Examples

```
bfrom_OtpWrite(PBS02+dPbsSet*4, OTP_LOWER_HALF |  
OTP_CHECK_FOR_PREV_WRITE,&(PBS.HalfPages.uqPbs02L));  
bfrom_OtpWrite(PBS02+dPbsSet*4, OTP_UPPER_HALF |  
OTP_CHECK_FOR_PREV_WRITE,&(PBS.HalfPages.uqPbs02H));  
bfrom_OtpWrite(PBS03+dPbsSet*4, OTP_LOWER_HALF |  
OTP_CHECK_FOR_PREV_WRITE,&(PBS.HalfPages.uqPbs03L));  
bfrom_OtpWrite(PBS03+dPbsSet*4, OTP_UPPER_HALF |  
OTP_CHECK_FOR_PREV_WRITE,&(PBS.HalfPages.uqPbs03H));
```

If a set of PBS pages has been written earlier, but need to be replaced by a new set, the old PBS pages have to be invalidated. Do not use the `OTP_CHECK_FOR_PREV_WRITE` option in this case.

Listing 17-12. Invalidating a set of PBS Pages

```
#include <blackfin.h>  
#include <bfrom_h>  
u32 dPbsSet = 0;  
u64 d1Invalidate = (u64)0xC000000000000000;  
bfrom_OtpWrite(PBS00+dPbsSet*4,  
    OTP_LOWER_HALF | OTP_NO_ECC,  
    &d1Invalidate);  
dPbsSet++;  
/* write next set as in Listing x-2 */
```

For production one may want to lock the PBS to protect them from being any overwritten in the field. This can be performed by the following instructions:

Listing 17-13. Write protecting a set of PBS Pages

```
#include <blackfin.h>
#include <bfrom.h>
u32 dPbsSet = 0;
bfrom_OtpCommand( OTP_INIT, OTP_INIT_VALUE);
bfrom_OtpWrite(PBS00+dPbsSet*4, OTP_LOCK, NULL);
bfrom_OtpWrite(PBS01+dPbsSet*4, OTP_LOCK, NULL);
bfrom_OtpWrite(PBS02+dPbsSet*4, OTP_LOCK, NULL);
bfrom_OtpWrite(PBS03+dPbsSet*4, OTP_LOCK, NULL);
bfrom_OtpCommand( OTP_CLOSE, 0);
```

When locking PBS pages remember the recommendation to duplicate the active set of PBS pages to approach best possible reliability. If in above examples the dPbsSet 4 contains the final configuration, program also the set 5 with the same data. For completeness, note that the above code example does not lock the ECC fields corresponding to the PBS pages. See [Chapter 16, “One-Time Programmable Memory”](#) for details.

Programming Examples

18 DYNAMIC POWER MANAGEMENT

This chapter describes the dynamic power management functionality of the Blackfin processor and includes the following sections:


- [“Phase-Locked Loop and Clock Control”](#) on page 18-1
- [“Dynamic Power Management Controller”](#) on page 18-7
- [“PLL and VR Registers”](#) on page 18-26
- [“System Control ROM Function”](#) on page 18-31
- [“Programming Examples”](#) on page 18-37

Phase-Locked Loop and Clock Control

The input clock into the processor, `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 `CLKIN`. The resulting, multiplied signal is the voltage controlled oscillator (`VCO`) clock. A user-programmable value then divides the `VCO` clock signal to generate the core clock (`CCLK`).

Phase-Locked Loop and Clock Control

Another user-programmable value divides the VCO signal to generate the system clock (SCLK). The SCLK signal clocks the peripheral access bus (PAB), DMA access bus (DAB), external access 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 (PLL_DIV), select a divider value that allows these buses to run at or below the maximum SCLK rate specified in the processor data sheet.

To optimize performance and power dissipation, the processor allows the core and system clock frequencies to change dynamically in a “coarse adjustment.” For a “fine adjustment,” the PLL clock frequency can also be varied.

PLL Overview

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 18-7](#).

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 18-1 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.

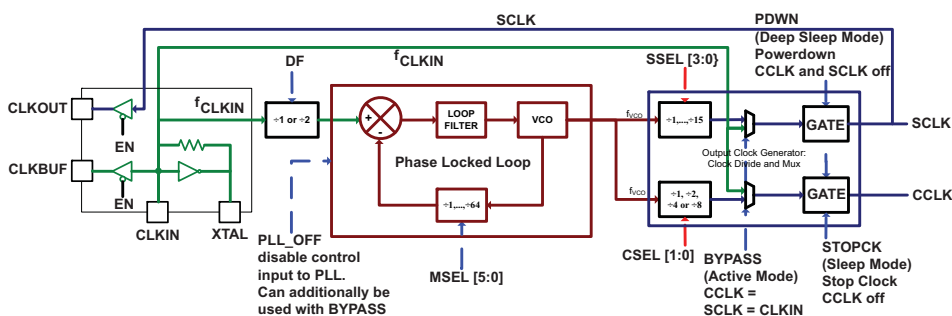


Figure 18-1. PLL Block Diagram

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 18-28](#).

Phase-Locked Loop and Clock Control

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 0x8. This value can be reprogrammed at startup in the boot code.

Table 18-1 illustrates the VCO multiplication factors for the various MSEL and DF settings.

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 the processor data sheet for maximum and minimum frequencies for CLKIN, CCLK, and VCO.

Table 18-1. MSEL Encodings

Signal Name MSEL[5:0]	VCO Frequency	
	DF = 0	DF = 1
0	64x	32x
1	1x	0.5x
2	2x	1x
N = 3–62	Nx	0.5Nx
63	63x	31.5x

The PLL control register (PLL_CTL) controls operation of the PLL (see [Figure 18-5 on page 18-28](#)). Note that changes to the PLL_CTL register do not take effect immediately. In general, the PLL_CTL register is first pro-

grammed with a new value, and then a specific PLL programming sequence must be executed to implement the changes. This is handled by the System Control ROM Function (SysControl), shown on page 18-31.

Core Clock/System Clock Ratio Control

Table 18-2 describes the programmable relationship between the VCO frequency and the core clock. Table 18-3 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 the 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 18-27.

The reset value of CSEL[1:0] is 0x0, and the reset value of SSEL[3:0] is 0x4. These values can be reprogrammed at startup by the boot code.

By updating PLL_DIV with an appropriate value, 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.

Unlike writing the PLL_CTL register, the PLL_DIV register can be updated at any time to change the CCLK and SCLK divide values without the PLL programming sequence.

Table 18-2. Core Clock Ratio

Signal Name CSEL[1:0]	Divider Ratio VCO/CCLK	Example Frequency Ratios (MHz)	
		VCO	CCLK
00	1	300	300
01	2	600	300

Phase-Locked Loop and Clock Control

Table 18-2. Core Clock Ratio (Cont'd)

Signal Name CSEL[1:0]	Divider Ratio VCO/CCLK	Example Frequency Ratios (MHz)	
		VCO	CCLK
10	4	600	150
11	8	400	50

As long as the MSEL and DF control bits in the PLL control register (PLL_CTL) remain constant, the PLL is locked.

Table 18-3. System Clock Ratio

Signal Name SSEL[3:0]	Divider Ratio VCO/SCLK	Example Frequency Ratios (MHz)	
		VCO	SCLK
0000	Reserved	N/A	N/A
0001	1:1	100	100
0010	2:1	200	100
0011	3:1	400	133
0100	4:1	500	125
0101	5:1	600	120
0110	6:1	600	100
N = 7–15	N:1	600	600/N



If changing the clock ratio through writing a new SSEL value into PLL_DIV, take care that the enabled peripherals do not suffer data loss due to SCLK frequency changes.

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 the 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 18-8](#).

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 18-8](#).
- 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 V_{DD} (V_{DDEXT}) supply.

Depending on the needs of the system, the voltage level can be reduced to save power. See [“Controlling the Voltage Regulator” on page 18-18](#).

Operating Modes

The processor works in four operating modes, each with unique performance and power saving benefits. [Table 18-4](#) summarizes the operational characteristics of each mode.

Table 18-4. Operational Characteristics

Operating Mode	Power Savings	PLL		CCLK	SCLK	Allowed DMA Access
		Status	Bypassed			
Full On	None	Enabled	No	Enabled	Enabled	L1
Active	Medium	Enabled ¹	Yes	Enabled	Enabled	L1
Sleep	High	Enabled	No	Disabled	Enabled	–
Deep Sleep	Maximum	Disabled	–	Disabled	Disabled	–

¹ PLL can also be disabled in this mode.

Dynamic Power Management Controller States

Power management states are synonymous with the PLL control state. The active and full on states of the DPMC/PLL can be determined by reading the PLL status register (see [“PLL Status \(PLL_STAT\) Register” on page 18-29](#)). In these modes, the core can either execute instructions or be in the idle core state. If the core is in the Idle state, it can be awakened by several sources.

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. The system clock (SCLK) frequency is determined by the SSEL-specified ratio to VCO. DMA access is available to L1 and external memories. From full on mode, the processor can transition directly to active, sleep, or deep sleep modes, as shown in [Figure 18-2 on page 18-12](#).

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 and external 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.



In this mode or in the transition phase to other modes, changes to MSEL are not latched by the PLL.

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. Since 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

When sleep mode is exited, the processor resumes execution from the program counter value present immediately prior to entering sleep mode.

Dynamic Power Management Controller

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 a hardware reset event or an RTC interrupt. A hardware reset begins the hardware reset sequence. An RTC interrupt causes the processor to transition to active mode, and execution resumes from where the program counter was when deep sleep mode was entered. If an interrupt is also enabled in SIC_IMASK0, the vector is taken immediately after exiting deep sleep and the ISR is executed.

Note an RTC interrupt in deep sleep mode automatically resets some fields of the PLL control register (PLL_CTL). See [Table 18-5](#).



When in deep sleep mode, clocking to the DDR is turned off. Before entering deep sleep mode, software should ensure that important information in DDR memory is saved to a non-volatile memory and/or the DDR is placed into self-refresh mode.

Table 18-5. PLL_CTL Values After RTC Wake-up Interrupt

Field	Value
PLL_OFF	0
STOPCK	0
PDWN	0
BYPASS	1

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 in the diagram of

[Figure 18-2](#). Since 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 18-22.

Operating Mode Transitions

[Figure 18-2](#) graphically illustrates the operating modes and transitions. In the diagram, ellipses represent operating modes and rectangles represent processor states. Arrows show the allowed transitions into and out of each mode or state.

For mode transitions, 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 transitions to processor states, the text next to each transition arrow shows either a processor event (for example, RTC wake-up or hardware reset) or the fields in the voltage regulator control register (VR_CTL) that must be changed for the transition to occur.

For information about how to effect mode transitions, see [“Programming Operating Mode Transitions”](#) on page 18-14.

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:

- **PLL disabled:** In addition to being bypassed in the active mode, the PLL can be disabled.

When the PLL is disabled, additional power savings are achieved although they are relatively small. To disable 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 disabled, it can be re-enabled later when additional performance is required.

The PLL must be re-enabled before transitioning to full on or sleep operating modes. To re-enable the PLL, clear the `PLL_OFF` bit in the `PLL_CTL` register, and then execute the PLL programming sequence.

Dynamic Power Management Controller

- New multiplier ratio: The clock-in to VCO clock (CLKIN to VCO) multiplier ratio can also be changed while in full on mode.

The PLL state automatically transitions to active mode while the PLL is locking. After locking, the PLL returns to full on mode. 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.

Table 18-6 summarizes the allowed operating mode transitions.



Attempting to cause mode transitions other than those shown in Table 18-6 causes unpredictable behavior.

Table 18-6. Allowed Operating Mode Transitions

New Mode	Current Mode			
	Full On	Active	Sleep	Deep Sleep
Full On	–	Allowed	Allowed	–
Active	Allowed	–	Allowed	Allowed
Sleep	Allowed	Allowed	–	–
Deep Sleep	Allowed	Allowed	–	–

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. This sequence is managed by an user-callable routine in the on-chip ROM called `bfrom_SysControl()`. When calling this function, no further precautions have to be taken.

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 relock. To relock, 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 (`PLL_LOCKCNT`), the PLL sets the `PLL_LOCKED` bit in the PLL status register (`PLL_STAT`), and the PLL asserts the PLL wake-up interrupt.

When the `bfrom_SysControl()` routine reprograms the `PLL_CTL` register with a new value, it executes a subsequent `IDLE` instruction. It prevents all other system interrupt sources other than the DPMC from waking the core up from the idle state. If the lock counter expires, the PLL issues an interrupt and the code execution continues with the instruction after the `IDLE` instruction. Therefore, the system is in the new state by the time the `bfrom_SysControl()` routine returns.



If the new value written to the `PLL_CTL` or `VR_CTL` register is the same as the previous value, the PLL wake-up occurs immediately (PLL is already locked), but the core and system clock are bypassed for the `PLL_LOCKCNT` duration. For this interval, code executes at the `CLKIN` rate instead of at the expected `CCLK` rate. Software guards against this condition by comparing the current value to the new value before writing the new value.

When the wake-up signal is asserted, the processor continues, causing a transition to:

- Active mode if the `BYPASS` bit 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 the sleep operating mode, the processor immediately transitions to the sleep mode and waits for a wake-up signal before continuing.

Dynamic Power Management Controller

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 operation mode and to return from the `bfrom_SysControl()` routine.
- A hardware reset causes the processor to execute the reset sequence. For more information about hardware reset, see [Chapter 17, “System Reset and Booting”](#)

If no operating mode transition is programmed, the PLL generates a wake-up signal, and `bfrom_SysControl()` routine returns.

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 multiple power domains. Each power domain has a separate V_{DD} supply. Note that the internal logic of the processor and much of the processor I/O can be run over a range of voltages. See the product data sheet for details on the allowed voltage ranges for each power domain and power dissipation data.

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 18-3](#). 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 switches 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.

Dynamic Power Management Controller

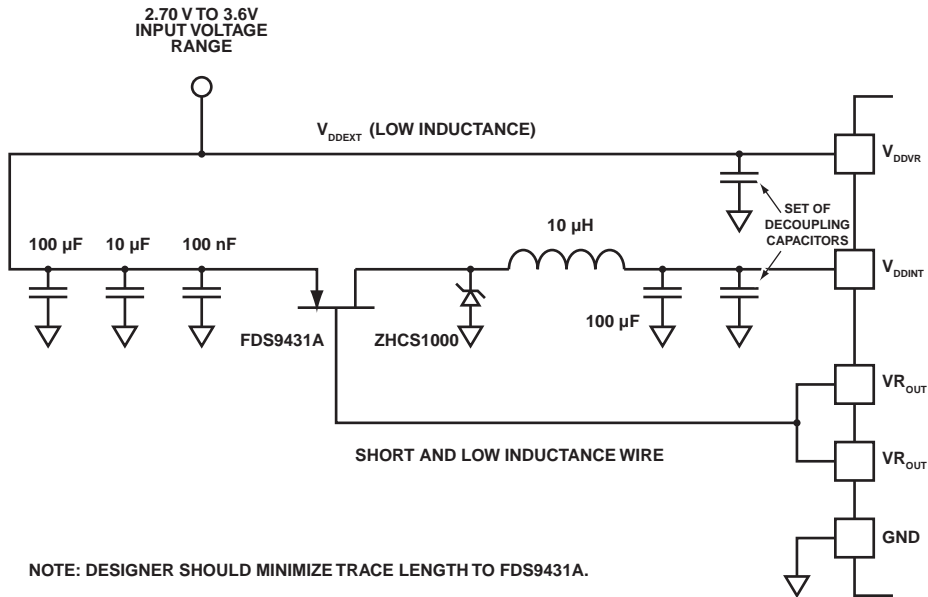


Figure 18-3. Processor Voltage Regulator

Controlling the Voltage Regulator

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 18-8 on page 18-30](#)). The state of the VR_CTL register is maintained during power down modes and hibernate state. It is only set to its reset value by a powerup reset sequence. The VR_CTL register should not be written directly. Rather, the `bfrom_SysControl()` routine, which resides in the on-chip ROM, should be used to access it.

The on-chip switching regulator can be modified in terms of its transient behavior in the `GAIN` and `FREQ` fields of the VR_CTL register.

The two-bit `GAIN` field controls the internal loop gain of the switching regulator loop; this field 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.

[Table 18-7](#) lists the gain levels configured by `GAIN[1:0]`.

Table 18-7. GAIN Encodings

GAIN	Value
b#00	5
b#01	10
b#10	20
b#11	50


The two-bit `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).

Dynamic Power Management Controller

Table 18-8 lists the switching frequency values configured by `FREQ[1:0]`.


Table 18-8. FREQ Encodings

FREQ	Value
b#00	Powerdown/bypass onboard regulation
b#01	333 kHz
b#10	667 kHz
b#11	1 MHz

 To bypass onboard regulation, program a value of `b#00` in the `FREQ` field and leave the `VROUT` pins floating. Nevertheless, the `VLEV` field in the applied `VR_CTL` value should still reflect the applied voltage value.

Changing Voltage

Minor changes in operating voltage can be accommodated without requiring special consideration or action by the application program. See the processor data sheet for more information about supported voltage levels, regulator 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 requires 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. The four-bit voltage level (`VLEV`) field identifies the nominal internal voltage level. Please refer to the processor data sheet for the applicable `VLEV` voltage range and associated voltage tolerances.

Table 18-9 lists the voltage level values for VLEV[3:0].

Table 18-9. VLEV Encodings

VLEV	Voltage
b#0000–b#0101	Reserved
b#0110	Reserved
b#0111	Reserved
b#1000	0.95 volts
b#1001	1.00 volts
b#1010	1.05 volts
b#1011	1.10 volts
b#1100	1.15 volts
b#1101	1.20 volts
b#1110	1.25 volts
b#1111	1.30 volts

After changing the voltage level in the VR_CTL register, the PLL automatically enters the active mode when the processor enters the idle state. At that point, the voltage level changes and the PLL relocks with the new voltage. After the PLL_LOCKCNT has expired, the part returns to the full on state. When changing voltages, a larger PLL_LOCKCNT value may be necessary than when changing just the PLL frequency. See the processor data sheet for details.

After the voltage is 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 the processor data sheet for the new operating voltage level.

Dynamic Power Management Controller

Even if the internal voltage regulator is bypassed and the VDDINT voltage is applied by an external regulator, the `bfrom_SysControl()` routine must be called at startup or whenever the voltage changes at run time. Afterwards, the `SYCTRL_EXTVOLTAGE` bit should be set along with a proper `VLEV` value in the `VR_CTL` register.

Powering Down the Core (Hibernate State)

The internal supply regulator for the processor can be shut off by writing `b#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 $\overline{\text{RESET}}$ pin always exits hibernate state and requires no modification to `VR_CTL`.
- 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).
- General-purpose event (all processors *except* ADSP-BF549). Set the general-purpose wake-up enable (`GPWE`) control bit to enable wake-up upon detection of an active low signal on the $\overline{\text{GPW}}$ pin.
- MXVR event (ADSP-BF549 processor *only*). Set the MXVR wake-up enable (`MXVRWE`) control bit to enable wake-up upon detection of an active low signal on the $\overline{\text{MRXON}}$ pin. For more details, see the “MXVR Module” chapter in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 2 of 2)*.

- Activity on either CAN_{RX} pin. Set the CAN RX wake-up enable ($CANWE$) control bit to enable wake-up upon detection of CAN bus activity on either of the CAN_{RX} pins. For more details, see the “CAN Module” chapter in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 2 of 2)*.
- Activity on the rotary counter pins. Set the rotary counter wake-up enable ($ROTWE$) control bit to enable wake-up upon activity on the rotary counter pins. If any edge is detected on either the CUD or CDG pins, or if an active low state is detected on the CZM pin, this wake-up event is generated. For more details, see the “Rotary Counter Module” chapter in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 2 of 2)*.
- Activity on the keypad pins. Set the keypad wake-up enable ($KPADWE$) control bit to enable wake-up upon activity on the keypad pins. If an active low state is detected on any of the KEY_ROW_x pins, this wake-up event is generated. For more details, see the “Keypad Module” chapter in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 2 of 2)*.
- USB activity. Set the USB wake-up enable ($USBWE$) control bit to enable wake-up upon USB activity. If any edge is detected on the USB_DP , USB_DM , or USB_VBUS pins, this wake-up event is generated. For more details, see the “USB Module” chapter in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 2 of 2)*.
- The hibernate functions will only work if V_{DDRTC} is supplied. This is the supply that is needed to maintain the VR_CTL register.




For the peripheral hibernate wake-up sources described above, a general-purpose wake-up can be implemented if the peripheral isn't used. For example, if $MXVR$ is not used, an external host can be connected to the \overline{MRXON} pin that holds the pin high until the

Dynamic Power Management Controller

wake-up is required. If `MXVRWE` is set, a transition to low on `MRXON` will exit hibernate state, and the host could be set up to provide this signal.

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 0 V, and so 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 `SCKE-low-during-reset` (`SCKELOW`) control bit in `VR_CTL` 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 `DDR` out of self-refresh mode, resulting in data decay in the `DDR` 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.

To power down the internal supply:

1. Write 0 to the appropriate bits in the `SIC_IWRx` registers to prevent enabled peripheral resources from interrupting the hibernate process.
2. Call the `bfrom_SysControl()` routine, ensuring that the `FREQ` bits in the `VR_CTL` variable are set to `b#00` and the appropriate wake-up bit(s) are set to 1 (`USBWE`, `ROTWE`, `GPWE/MXVRWE`, `KPADWE`, `CANWE`, `PHYWE`, `WAKE`). Optionally, set the `SCKELOW` bit if `DDR` data should be maintained.

3. The `bfrom_SysControl()` routine will execute until V_{DDINT} transitions to 0V. It never returns.
4. When the processor is woken up, the PLL relocks and the boot sequence defined by the `BMODE[3:0]` pin settings takes effect.

The `WURESET` in the `SYSCTRL` register is set and stays set until the next hardware reset. The `WURESET` bit may control conditional boot process.

Recovery From Hibernate State

When utilizing the hibernate state to maximize power savings, additional features of the ADSP-BF54x Blackfin processors can be used to coordinate system response and subsequent system activity when the processor resumes execution upon a hibernate wake-up event.

For the system outside of the Blackfin processor, the `EXT_WAKE` output pin is asserted (driven high) when the Blackfin processor is about to enter the Hibernate state. This pin can be used in the system to signal an external component that it is now safe to remove the power supply. The state of the `EXT_WAKE` pin is not affected by the reset sequence, and no clock is required for it to be driven. The `EXT_WAKE` pin is then driven low when the on-chip regulator is again stable after resuming operation from the hibernate state.

For the Blackfin processor itself, the `PLL_STAT` register contains a set of wake-up status bits, which can be interrogated upon warm-boot to determine which source caused the wake-up event. This information can be useful to coordinate with external system components regarding lost traffic due to the previous activity causing a wake-up event rather than a processed message. For example, if a CAN message took the processor out of hibernate state, that message would not have been received by the processor because the processor would have had to perform a self-reset and boot and run the application before being able to actually handle a CAN message.

PLL and VR Registers

The `SCKELOW` bit in the `VR_CTL` register is maintained during the hibernate state. Typical use of this bit is to protect data in DDR memory during the hibernate state and subsequent reset event. Because of this, `SCKELOW` can be checked by software to determine whether the processor is being booted for the first time or if it is restarting after a hibernate event. If the application had set the bit prior to hibernate to protect the contents of DDR memory, the bit will read as 1 after the reset event takes place. This feature is useful if, for example, the desire is to shorten boot times as much as possible. For larger applications, anything resolved to external memory and preserved for the duration of the hibernate state does not need to boot again after the wake-up event takes place. Adding code to an initialization block that simply checks the `SCKELOW` bit provides the application with the ability to determine whether a full boot or some abridged boot is necessary to have the full application resolved to the internal and external memory spaces.

PLL and VR Registers

The user interface to the PLL and VR is through five memory-mapped registers (MMRs) shown in [Table 18-10](#) and illustrated in [Figure 18-4](#) through [Figure 18-8](#).

Table 18-10. PLL/VR Register Mapping

Register Name	Function	See More Information ...	Notes
PLL_CTL	PLL control register	on page 18-28	Requires reprogramming sequence when written
PLL_DIV	PLL divisor register	on page 18-27	Can be written freely
PLL_STAT	PLL status register	on page 18-29	Monitors active modes of operation and wake-up events

Table 18-10. PLL/VR Register Mapping (Cont'd)

Register Name	Function	See More Information ...	Notes
PLL_LOCKCNT	PLL lock count register	on page 18-29	Number of SCLKs allowed for PLL to relock
VR_CTL	Voltage regulator control register	on page 18-30	Requires PLL reprogramming sequence when written

All four 16-bit MMRs must be accessed with aligned 16-bit reads/writes.

PLL Divide (PLL_DIV) Register

PLL Divide Register (PLL_DIV)

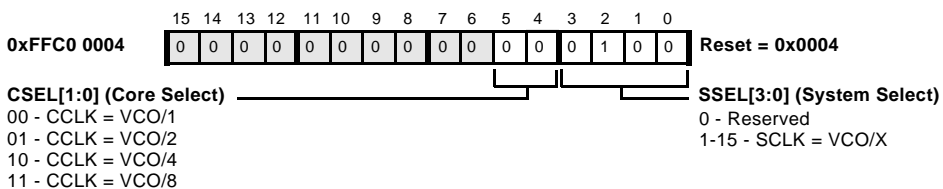


Figure 18-4. PLL Divide (PLL_DIV) Register

PLL Control (PLL_CTL) Register

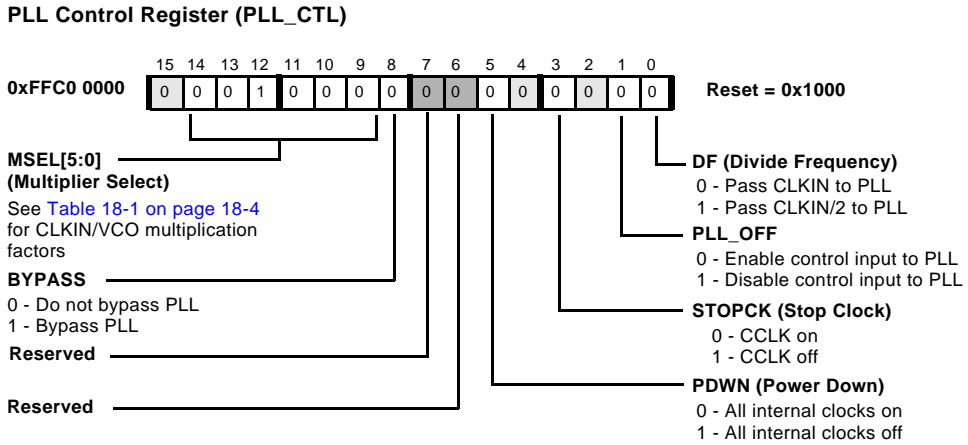


Figure 18-5. PLL Control (PLL_CTL) Register

PLL Status (PLL_STAT) Register

PLL Status Register (PLL_STAT)

Read only. Unless otherwise noted, 1 - Processor operating in this mode. For more information, see “Operating Modes” on page 18-8.

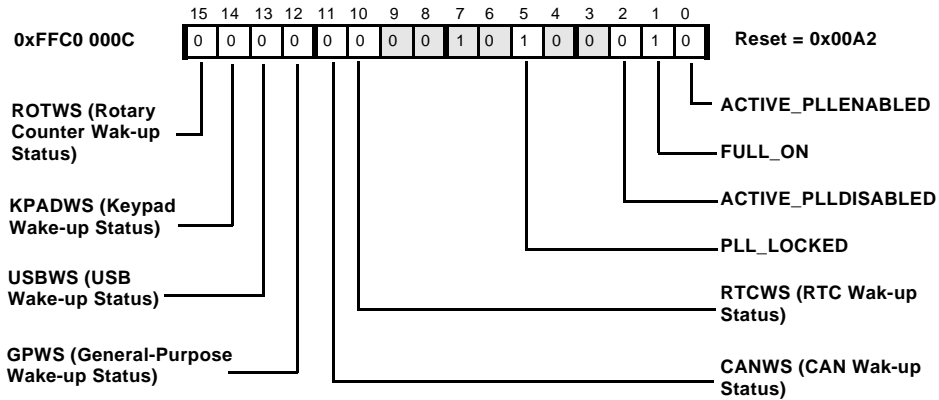


Figure 18-6. PLL Status (PLL_STAT) Register

PLL Lock Count (PLL_LOCKCNT) Register

PLL Lock Count Register (PLL_LOCKCNT)

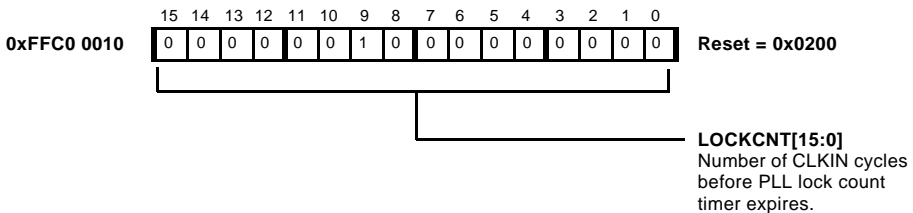


Figure 18-7. PLL Lock Count (PLL_LOCKCNT) Register

Voltage Regulator Control (VR_CTL) Register

Voltage Regulator Control Register (VR_CTL)

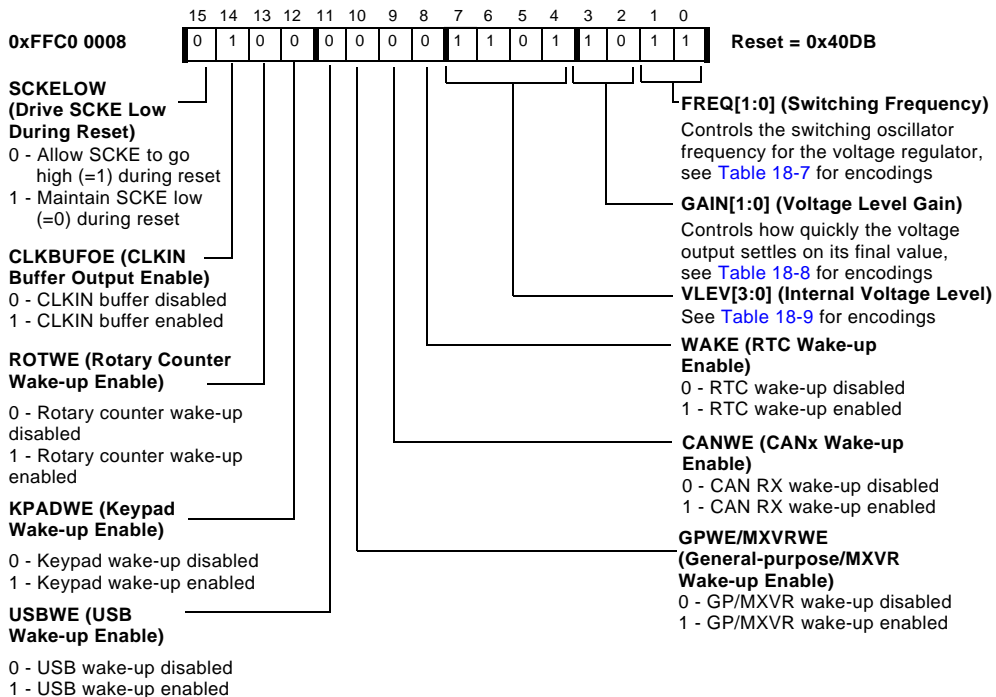


Figure 18-8. Voltage Regulator Control (VR_CTL) Register

i Bit 10 enables the general-purpose wake-up (GPWE) for all processors *except* the ADSP-BF549. On ADSP-BF549 processors *only*, bit 10 enables the MXVR wake-up (MXVRWE).

The CLKIN buffer output enable (CLKBUFOE) control bit allows the Blackfin processor and another device to run from a single crystal oscillator. Clearing this bit prevents the CLKBUF pin from driving a buffered version of the input clock CLKIN.

System Control ROM Function

The PLL and voltage regulator registers should never be accessed directly. Rather, always use to the `bfrom_SysControl()` function to alter or read the register values. This function resides in the on-chip ROM and can be called by the user following C language style calling conventions.

Entry address: 0xEF00 0038

Arguments:

- `dActionFlags` word in R0
- `pSysCtrlSettings` pointer in R1
- zero value in R2

A potential error message from the internally called `bfrom_OtpRead()` function is forwarded and returned in R0.



The System Control ROM Function does not verifying the correctness of the forwarded arguments. Therefore, it is up to the programmer to choose the correct values.

C prototype: `u32 bfrom_SysControl(u32 dActionFlags, ADI_SYSCTRL_VALUES *pSysCtrlSettings, void *reserved);`

The first argument (`u32 dActionFlags`) holds the instruction flags. The following flags are supported:

```
#define SYSCTRL_READ      0x00000000
#define SYSCTRL_WRITE    0x00000001
#define SYSCTRL_SYSRESET 0x00000002
#define SYSCTRL_SOFTRESET 0x00000004
#define SYSCTRL_VRCTL    0x00000010
#define SYSCTRL_EXTVOLTAGE 0x00000020
#define SYSCTRL_INTVOLTAGE 0x00000000
#define SYSCTRL_OTPVOLTAGE 0x00000040
```

System Control ROM Function

```
#define SYSCTRL_PLLCTL      0x00000100
#define SYSCTRL_PLLDIV     0x00000200
#define SYSCTRL_LOCKCNT    0x00000400
#define SYSCTRL_PLLSTAT    0x00000800
```


With `SYSCTRL_READ` and `SYSCTRL_WRITE`, a read or a write operation is initialized. `SYSCTRL_SYSRESET` performs a system reset, and `SYSCTRL_SOFTRESET` combines a core and a system reset. The `SYSCTRL_EXTVOLTAGE` and `SYSCTRL_INTVOLTAGE` indicate whether if `VDDINT` is supplied externally or generated by the on-chip regulator; `SYSCTRL_OTPVOLTAGE` is for factory purposes only. The last five flags (`_VRCTL`, `_PLLCTL`, `_PLLDIV`, `_LOCKCNT`, `_PLLSTAT`) tell the system control ROM function, which register to write or read. Remember, `SYSCTRL_PLLSTAT` is read only.

The second argument (`ADI_SYSCTRL_VALUES *pSysCtrlSettings`) passes a pointer to a special structure, which has entries for all PLL and voltage regulator registers. It is predefined in the `bfrom.h` header file as

```
typedef struct {
    u16 uwVrCtl;
    u16 uwPl1Ctl;
    u16 uwPl1Div;
    u16 uwPl1LockCnt;
    u16 uwPl1Stat;
} ADI_SYSCTRL_VALUES;
```

The third argument is reserved and should always be kept zero (NULL pointer).

For the return value, see the description of the `bfrom_OtpRead()` ROM routine, whereby single-bit warnings are suppressed.

 The System Control ROM Function automatically performs the correct programming sequence for the Dynamic Power Management System of the Blackfin processor.

Programming Model

The programming model for the Access System Control ROM Function in C/C++ and Assembly is described in the following sections.

Access System Control ROM Function in C/C++

To read the `PLL_DIV` and `PLL_CTL` register values, specify the `SYSCTRL_READ` instruction flag along with `SYSCTRL_PLLCTL` and `SYSCTRL_PLLDIV` register flags. The `bfrom_OtpRead()` function then only updates the `uwPllCtl` and `uwPllDiv` variables.

```
ADI_SYSCTRL_VALUES read;  
bfrom_SysControl ( SYSCTRL_PLLCTL | SYSCTRL_PLLDIV |  
SYSCTRL_READ, &read, NULL );
```

The values of the `PLL_CTL` and `PLL_DIV` registers can be accessed in the `read.uwPllCtl` and `read.uwPllDiv`, respectively.

To update register values, specify the `SYSCTRL_WRITE` instruction flag along with the register flags of those register that should be modified and have valid data in the respective `ADI_SYSCTRL_VALUES` variables.

```
ADI_SYSCTRL_VALUES write;  
write.uwPllCtl = 0x1400;  
write.uwPllDiv = 0x0005;  
bfrom_SysControl ( SYSCTRL_PLLCTL | SYSCTRL_PLLDIV |  
SYSCTRL_WRITE, &write, NULL );
```

System Control ROM Function

Access System Control ROM Function in Assembly

The assembler supports C structs. Here it is required to import the file `bfrom.h`.

```
#include <bfrom.h>
.IMPORT "bfrom.h";
.STRUCT ADI_SYSCtrl_VALUES dpm;
```

You are free to pre-load the struct:

```
.STRUCT ADI_SYSCtrl_VALUES dpm = { 0x40DB, 0x1400, 0x0005,
0x0200, 0x00A2 };
```

You can also load the values dynamically inside the code:

```
P5.H = hi(dpm);
```

```
P5.L = lo(dpm->uwVrCtl);
R7 = 0x40DB (z);
w[P5] = R7;
```

```
P5.L = lo(dpm->uwP11Ctl);
R7 = 0x1400 (z);
w[P5] = R7;
```

```
P5.L = lo(dpm->uwP11Div);
R7 = 0x0005 (z);
w[P5] = R7;
```

```
P5.L = lo(dpm->uwP11LockCnt);
R7 = 0x0200 (z);
w[P5] = R0;
```

The function `u32 bfrom_SysControl(u32 dActionFlags, ADI_SYSCTRL_VALUES *pSysCtrlSettings, void *reserved)`; can be accessed by `bfrom_syscontrol`. Following the C/C++ run-time environment conventions, the parameters passed are held by the data registers R0, R1 and R2.

```
/* 10 = sizeof(ADI_SYSCTRL_VALUES). uimm18m4: 18-bit unsigned
field that must be a multiple of 4, with a range of 8 through
262,152 bytes (0x00000 through 0x3FFFC) */
link sizeof(ADI_SYSCTRL_VALUES)+2;
```

```
[--SP] = (R7:0,P5:0);
```

```
/* Always allocate at least 12 bytes on the stack for outgoing
arguments, even if the function being called requires less than
this. */
```

```
SP += -12;
```

```
R0 = ( SYSCTRL_VRCTL      |
        SYSCTRL_INTVOLTAGE |
        SYSCTRL_PLLCTL   |
        SYSCTRL_PLLDIV   |
        SYSCTRL_WRITE    );
```

```
R1.H = hi(dpm);
```

```
R1.L = lo(dpm);
```

```
R2 = 0 (z);
```

```
P5.H = hi(BFROM_SYSCONTROL);
```

```
P5.L = lo(BFROM_SYSCONTROL);
```

```
call(P5);
```

```
SP += 12;
```

```
(R7:0,P5:0) = [SP++];
```

```
unlink;
```

```
rts;
```

System Control ROM Function

As an alternative for taking a C-struct, the processor's internal scratchpad memory can be used too. Therefore, the stack/frame pointer must be loaded and passed.

```
/* 10 = sizeof(ADI_SYSCtrl_VALUES). uimm18m4: 18-bit unsigned
field that must be a multiple of 4, with a range of 8 through
262,152 bytes (0x00000 through 0x3FFFC) */
link sizeof(ADI_SYSCtrl_VALUES)+2;
```

```
[--SP] = (R7:0,P5:0);
```

```
/* Always allocate at least 12 bytes on the stack for outgoing
arguments, even if the function being called requires less than
this. */
SP += -12;
```

```
R7 = 0;
R7.L = 0x40DB;
w[FP+sizeof(ADI_SYSCtrl_VALUES)+
offsetof(ADI_SYSCtrl_VALUES,uwVrCtl)] = R7;
R7.L = 0x1400;
w[FP+sizeof(ADI_SYSCtrl_VALUES)+
offsetof(ADI_SYSCtrl_VALUES,uwPllCtl)] = R7;
R7.L = 0x0005;
w[FP+sizeof(ADI_SYSCtrl_VALUES)+
offsetof(ADI_SYSCtrl_VALUES,uwPllDiv)] = R7;
R7.L = 0x0200;
w[FP+sizeof(ADI_SYSCtrl_VALUES)+
offsetof(ADI_SYSCtrl_VALUES,uwPllLockCnt)] = R7;
```

```
R0 = ( SYSCTRL_VRCTL      |
        SYSCTRL_INTVOLTAGE |
        SYSCTRL_PLLCTL   |
        SYSCTRL_PLLDIV   |
        SYSCTRL_WRITE     );
```



```
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);
R2 = 0;

P5.H = hi(BFROM_SYSCONTROL);
P5.L = lo(BFROM_SYSCONTROL);
call(P5);

SP += 12;
(R7:0,P5:0) = [SP++];
unlink;
rts;
```

Programming Examples

The following code examples illustrate how to use the System Control ROM Function in running various operating mode transitions. Some setup code has been removed for clarity, and the following assumptions are made:

- PLL Control (PLL_CTL) Register setting: 0x1400
- PLL Divider (PLL_DIV) Register setting: 0x0005
- PLL Lock Count (PLL_LOCKCNT) Register setting: 0x0200
- Clock in (CLKIN) frequency: 25MHz

VCO frequency is 250MHz, core clock frequency is 250MHz and system clock frequency is 50MHz.

- Voltage Regulator Control (VR_CTL) Register setting: 0x40DB
- Logical voltage level (VDDINT) is at 1.20V

Programming Examples

For operating mode transition and voltage regulator examples:

- C

```
#include <blackfin.h>
```

```
#include <bfrom.h>
```

- Assembly

```
#include <blackfin.h>
```

```
#include <bfrom.h>
```

```
.IMPORT "bfrom.h";
```

```
#define IMM32(reg,val) reg##.H=hi(val); reg##.L=lo(val)
```

Full On Mode to Active Mode and Back

[Listing 18-1](#) and [Listing 18-2](#) provide code for transitioning from full on operating mode to active mode, in C and Blackfin assembly code, respectively.

Listing 18-1. Transitioning from Full On Mode to Active Mode (C)

```
void active(void)
{
    ADI_SYSCTRL_VALUES active;
    bfrom_SysControl( SYSCTRL_INTVOLTAGE | SYSCTRL_PLLCTL |
    SYSCTRL_READ, &active, NULL );
    active.uwPllCtl |= (BYPASS | PLL_OFF); /* PLL_OFF bit optional */
    bfrom_SysControl( SYSCTRL_INTVOLTAGE | SYSCTRL_PLLCTL |
    SYSCTRL_WRITE, &active, NULL );
    return;
}
```

Listing 18-2. Transitioning from Full On Mode to Active Mode (ASM)

```

__active:

link sizeof(ADI_SYSCTRL_VALUES)+2;
[--SP] = (R7:0,P5:0);
SP += -12;

R0 = ( SYSCTRL_INTVOLTAGE | SYSCTRL_PLLCTL | SYSCTRL_READ );
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);
R2 = 0 (z);
IMM32(P4,BFROM_SYSCONTROL);
call(P4);

R0 = w[FP+-sizeof(ADI_SYSCTRL_VALUES)+
offsetof(ADI_SYSCTRL_VALUES,uwP11Ct1)];
bitset(R0,bitpos(BYPASS));
bitset(R0,bitpos(PLL_OFF));
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+
offsetof(ADI_SYSCTRL_VALUES,uwP11Ct1)] = R0;

R0 = ( SYSCTRL_INTVOLTAGE | SYSCTRL_PLLCTL | SYSCTRL_PLLDIV |
SYSCTRL_WRITE );
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);
R2 = 0 (z);
IMM32(P4,BFROM_SYSCONTROL);
call(P4);

SP += 12;
(R7:0,P5:0) = [SP++];
unlink;
rts;

```

Programming Examples

```
__active.end:
```

To return from active mode (go back to full on mode), the `BYPASS` bit and the `PLL_OFF` bit, respectively, must be cleared again.

Transition to Sleep Mode or Deep Sleep Mode

[Listing 18-3](#) and [Listing 18-4](#) provide code for transitioning from full on operating mode to sleep or deep sleep mode, in C and Blackfin assembly code, respectively.

Listing 18-3. Transitioning to Sleep Mode or Deep Sleep Mode, respectively (C)

```
void sleep(void)
{
    ADI_SYSCTRL_VALUES sleep;
    bfrom_SysControl( SYSCTRL_INTVOLTAGE | SYSCTRL_PLLCTL |
        SYSCTRL_READ, &sleep, NULL );
    active.uwPllCtl |= STOPCK; /* either: Sleep Mode */
    active.uwPllCtl |= PDWN; /* or: Deep Sleep Mode */
    bfrom_SysControl( SYSCTRL_INTVOLTAGE | SYSCTRL_PLLCTL |
        SYSCTRL_WRITE, &sleep, NULL );
    return;
}
```

Listing 18-4. Transitioning to Sleep Mode or Deep Sleep Mode, respectively (ASM)

```
__sleep:
```

```
link sizeof(ADI_SYSCTRL_VALUES)+2;
```

```
[--SP] = (R7:0,P5:0);
SP += -12;

R0 = ( SYSCTRL_INTVOLTAGE | SYSCTRL_PLLCTL | SYSCTRL_READ );
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);
R2 = 0 (z);
IMM32(P4,BFROM_SYSCONTROL);
call(P4);

R0 = w[FP+sizeof(ADI_SYSCTRL_VALUES)+
offsetof(ADI_SYSCTRL_VALUES,uwP11Ct1)];
bitset(R0,bitpos(STOPCK)); /* either: Sleep Mode */
bitset(R0,bitpos(PDWN)); /* or: Deep Sleep Mode */
w[FP+sizeof(ADI_SYSCTRL_VALUES)+
offsetof(ADI_SYSCTRL_VALUES,uwP11Ct1)] = R0;

R0 = ( SYSCTRL_INTVOLTAGE | SYSCTRL_PLLCTL | SYSCTRL_WRITE );
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);
R2 = 0 (z);
IMM32(P4,BFROM_SYSCONTROL);
call(P4);

SP += 12;
(R7:0,P5:0) = [SP++];
unlink;
rts;

__sleep.end:
```

Setting Wakeups and Entering Hibernate State

[Listing 18-5](#) and [Listing 18-6](#) provide code for configuring the regulator wakeups (RTC wakeup) and placing the regulator in the hibernate state, in C and Blackfin assembly code, respectively.

Listing 18-5. Configuring Regulator Wakeups and Entering Hibernate State (C)

```
void hibernate(void)
{
    ADI_SYSCCTRL_VALUES hibernate;
    /* SCLKELOW = 1: Enable Drive CKE Low During Reset */
    /* Protect DDR contents during reset after wakeup */
    hibernate.uwVrCtl = SCKELOW |
        WAKE      | /* RTC/Reset Wake-Up Enable */
        nCANWE    | /* CAN Wake-Up Disable */
        nGPWE     | /*General-Purpose Wake-Up Disable*/
        nUSBWE    | /* USB Wake-Up Disable */
        nKPADWE   | /* Keypad Wake-Up Disable */
        nROTWE    | /* Rotary Wake-Up Disable */
        nCLKBUFOE | /* CLKIN Buffer Output Disable */
        HIBERNATE; /* Powerdown/Bypass On-Board Regulation */
    bfrom_SysControl( SYSCCTRL_VRCTL | SYSCCTRL_INTVOLTAGE |
        SYSCCTRL_WRITE, &hibernate, NULL );
    /* Hibernate State: no code executes until wakeup triggers reset
    */
}
```

Listing 18-6. Configuring Regulator Wakeups and Entering Hibernate State (ASM)

```

__hibernate:

link sizeof(ADI_SYSCCTRL_VALUES)+2;
[--SP] = (R7:0,P5:0);
SP += -12;

cli R6; /* disable interrupts, copy IMASK to R6 */

/* SCLKELOW = 1: Enable Drive CKE Low During Reset */
/* Protect DDR contents during reset after wakeup */
R0.L = SCKELOW |
        WAKE      | /* RTC/Reset Wake-Up Enable */
        nCANWE   | /* CAN Wake-Up Disable */
        nGPWE    | /* General-Purpose Wake-Up Disable */
        nUSBWE   | /* USB Wake-Up Disable */
        nKPADWE  | /* Keypad Wake-Up Disable */
        nROTWE   | /* Rotary Wake-Up Disable */
        nCLKBUFOE | /* CLKIN Buffer Output Disable */
        HIBERNATE ; /* Powerdown/Bypass On-Board Regulation */
w[FP+sizeof(ADI_SYSCCTRL_VALUES)+
offsetof(ADI_SYSCCTRL_VALUES,uwVrCtl)] = R0;

R0 = ( SYSCTRL_VRCTL | SYSCTRL_INTVOLTAGE | SYSCTRL_WRITE );
R1 = FP;
R1 += -sizeof(ADI_SYSCCTRL_VALUES);
R2 = 0 (z);
IMM32(P4,BFROM_SYSCCONTROL);
call(P4);

```

Programming Examples

```
/* Hibernate State: no code executes until wakeup triggers reset
*/
```

```
__hibernate.end:
```

Perform a System Reset or Soft-Reset

[Listing 18-7](#) and [Listing 18-8](#) provide code for executing a system reset or a soft-reset (= system reset + core reset), in Blackfin assembly and C code, respectively.

Listing 18-7. Execute a System Reset or a Soft-Reset

```
void reset(void)
{
    bfrom_SysControl( SYSCTRL_SYSRESET, NULL, NULL ); /* either */
    bfrom_SysControl( SYSCTRL_SOFTRESET, NULL, NULL ); /* or */
    return;
}
```

Listing 18-8. Listing 8. Execute a System Reset or a Soft-Reset

```
__reset:

    link sizeof(ADI_SYSCTRL_VALUES)+2;
    [--SP] = (R7:0,P5:0);
    SP += -12;

    R0 = ( SYSCTRL_SYSRESET ); /* either */
    R0 = ( SYSCTRL_SOFTRESET ); /* or */
    R1 = 0 (z);
    R2 = 0 (z);
    IMM32(P4,BFROM_SYSCONTROL);
    call(P4);
```



```

SP += 12;
(R7:0,P5:0) = [SP++];
unlink;
rts;

__reset.end:

```

Change VCO Frequency, Core Clock Frequency and System Clock Frequency

[Listing 18-9](#) and [Listing 18-10](#) provide code for changing the CLKIN to VCO multiplier (from 10x to 21x), keeping CSEL divider at (1) and changing the SSEL divider (from 5 to 4) in full on operating mode, in C and Blackfin assembly code, respectively.

Listing 18-9. Transition of Frequencies (C)

```

void frequency(void)
{
    ADI_SYSCTRL_VALUES frequency;

    /* Set MSEL = 0-63 --> VCO = CLKIN*MSEL */
    frequency.uwP11Ctl = SET_MSEL(21) ;

    /* Set SSEL = 1-15 --> SCLK = VCO/SSEL */
    /* CCLK = VCO / 1 */
    frequency.uwP11Div = SET_SSEL(4) |
    CSEL_DIV1 ;

    frequency.uwP11LockCnt = 0x0200;

```

Programming Examples

```
bfrom_SysControl( SYSCTRL_INTVOLTAGE | SYSCTRL_PLLCTL |
SYSCTRL_PLLDIV | SYSCTRL_LOCKCNT | SYSCTRL_WRITE, &frequency,
NULL );
return;
}
```

Listing 18-10. Transition of Frequencies (ASM)

```
__frequency:

link sizeof(ADI_SYSCTRL_VALUES)+2;
[--SP] = (R7:0,P5:0);
SP += -12;

/* write the struct */
R0 = 0;

R0.L = SET_MSEL(21) ; /* Set MSEL = 0-63 --> VCO = CLKIN*MSEL */
w[FP+sizeof(ADI_SYSCTRL_VALUES)+
offsetof(ADI_SYSCTRL_VALUES,uwPl1Ctl)] = R0;

R0.L = SET_SSEL(4) | /* Set SSEL = 1-15 --> SCLK = VCO/SSEL */
      CSEL_DIV1 ; /* CCLK = VCO / 1 */
w[FP+sizeof(ADI_SYSCTRL_VALUES)+
offsetof(ADI_SYSCTRL_VALUES,uwPl1Div)] = R0;

R0.L = 0x0200;
w[FP+sizeof(ADI_SYSCTRL_VALUES)+
offsetof(ADI_SYSCTRL_VALUES,uwPl1LockCnt)] = R0;

/* argument 1 in R0 */
R0 = ( SYSCTRL_INTVOLTAGE | SYSCTRL_PLLCTL | SYSCTRL_PLLDIV |
SYSCTRL_WRITE );
```

```
/* argument 2 in R1: structure lays on local stack */
R1 = FP;
R1 += -sizeof(ADI_SYSCCTRL_VALUES);

/* argument 3 must always be NULL */
R2 = 0;

/* call of SysControl function */
IMM32(P4,BFROM_SYSCONTROL);
call (P4); /* R0 contains the result from SysControl */

SP += 12;
(R7:0,P5:0) = [SP++];
unlink;
rts;

__frequency.end;
```

Changing Voltage Levels

[Listing 18-11](#) and [Listing 18-12](#) provide code for changing the voltage level dynamically, in C and Blackfin assembly code, respectively. The voltage level will be increased to 1.25V. Additional code may be required to alter the core clock frequency when voltage level is being decreased. Please refer to the processor data sheet for the applicable VLEV voltage range and associated supported core clock speeds.

Listing 18-11. Changing Core Voltage via the On-Chip Regulator (C)

```
void voltage(void)
{
    ADI_SYSCCTRL_VALUES voltage;
    voltage.uwVrCtl = VLEV_125 | /* VLEV = 1.25 V */
                    CLKBUF0E | /* CLKIN Buffer Output Enable */
}
```

Programming Examples

```
        GAIN_20    | /* GAIN = 20 */
        FREQ_1000 ; /* Switching Frequency Is 1 MHz */
bfrom_SysControl( SYSCTRL_VRCTL | SYSCTRL_INTVOLTAGE |
SYSCTRL_WRITE, &voltage, NULL );
return;
}
```

Listing 18-12. Changing Core Voltage through the On-Chip Regulator (ASM)

```
__voltage:

link sizeof(ADI_SYSCTRL_VALUES)+2;
[--SP] = (R7:0,P5:0);
SP += -12;

R0.L = VLEV_125    | /* VLEV = 1.25 V */
        CLKBUFOE  | /* CLKIN Buffer Output Enable */
        GAIN_20    | /* GAIN = 20 */
        FREQ_1000 ; /* Switching Frequency Is 1 MHz */
w[FP+-sizeof(ADI_SYSCTRL_VALUES)+
offsetof (ADI_SYSCTRL_VALUES,uwVrCt1)] = R0;

R0 = ( SYSCTRL_VRCTL | SYSCTRL_INTVOLTAGE | SYSCTRL_WRITE );
R1 = FP;
R1 += -sizeof(ADI_SYSCTRL_VALUES);
R2 = 0 (z);
IMM32(P4,BFROM_SYSCONTROL);
call(P4);

SP += 12;
(R7:0,P5:0) = [SP++];
unlink;
rts;
```

`__voltage.end:`

The previous sequence must also be executed when the V_{DDINT} voltage is applied externally to ensure internal timings can appropriately be adjusted for the constant or changing V_{DDINT} voltage. In this case, replace the `SYSCTRL_INTVOLTAGE` flag with the `SYSCTRL_EXTVOLTAGE` flag.

Programming Examples

19 SYSTEM DESIGN

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, the design issues cited here are discussed in detail in other sections of this manual. In such cases, a reference is made to the corresponding section of the text, instead of repeating the discussion in this chapter.

- [“Pin Descriptions” on page 19-1](#)
- [“Managing Clocks” on page 19-2](#)
- [“Configuring and Servicing Interrupts” on page 19-2](#)
- [“Semaphores” on page 19-3](#)
- [“Data Delays, Latencies, and Throughput” on page 19-4](#)
- [“Bus Priorities” on page 19-5](#)
- [“System-Level Hardware Design” on page 19-5](#)
- [“Recommended Reading” on page 19-19](#)

Pin Descriptions

Refer to the processor data sheet for pin information, including pin numbers for the 400-ball MBGA.

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 through the `CSEL` bit settings in the `PLL_DIV` register. The system clock is based on a divider ratio that is programmed through the `SSEL` bit settings in the `PLL_DIV` register. For detailed information about how to set and change `CCLK` and `SCLK` frequencies, see [Chapter 18, “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 through the system interrupt assignment registers (`SIC_IARx`). For more information, see [Chapter 4, “System Interrupts”](#).

The processor core supports nested and non-nested interrupts, as well as self-nested interrupts.

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 through 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.

Data Delays, Latencies, and Throughput

Example Code for Query Semaphore

[Listing 19-1](#) provides an example of a query semaphore that checks the availability of a shared resource.

Listing 19-1. Query Semaphore

```
/* Query semaphore. Denotes "Busy" if its value is nonzero. Wait
until free (or reschedule thread-- see note below). P0 holds
address of semaphore. */
QUERY:
TESTSET ( P0 ) ;
IF !CC JUMP QUERY ;
/* At this point, semaphore is granted to current thread, and all
other contending threads are postponed because semaphore value at
[P0] is nonzero. Current thread could write thread_id to sema-
phore location to indicate current owner of resource. */
R0.L = THREAD_ID ;
B[P0] = R0 ;
/* When done using shared resource, write a zero byte to [P0] */
R0 = 0 ;
B[P0] = R0 ;
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 2, "Chip Bus Hierarchy"](#).

Bus Priorities

For an explanation of prioritization between the various internal buses, refer to [Chapter 2, “Chip Bus Hierarchy”](#).

System-Level Hardware Design

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.

External Memory Design Issues

This section describes design issues related to external memory.

DDR Memory

The DDR controller has a dedicated set of pins that require special attention in board design and layout:

- Follow the recommendations of the DDR memory manufacturer. A good example of DDR layout recommendations is: *TN-46-14: Hardware Tips for Point-to-Point System Design* from Micron Technology.
- Proper board design and trace length matching is a critical part of reducing the DQS to DQ and DQM skew of any DDR design. Proper clock and V_{REF} layout are also critical.

System-Level Hardware Design

- When matching trace lengths in board layout, be careful of the shape of the serpentine pattern. Capacitive coupling between segments of the trace reduces the effectiveness of length matching. About 20 mils is a good distance between segments of the same trace.
- Use serial termination on all data, address and control signals (except the CLKs) for small memory systems of one to four chips. Four or more devices may require parallel termination to a separately-generated V_{REF} instead of the serial termination.
- The maximum trace length of DQS, DQM and DQ signals should be less than 3.5 inches. This is a maximum total length for each signal measured by adding the length before, after and through any serial termination.
- Signal `DDR_VSSR` on the Blackfin processor is a shield signal to reduce noise on the `DDR_VREF` signal. It should be connected to ground right at the ball.
- `DDR_VREF` is a standard DDR signal with a voltage value of $VDD_DDR/2$. Place a `DDR_VREF` filtration capacitor to ground within 0.1 inch of the ball on the Blackfin processor. The signal should be derived in the standard way using 1% resistors and 0.1 μ F capacitor or capacitors even if using mobile DDR devices that do not also need the `DDR_REF` signal.

Memory Bus Pin Muxing and Flow Control

DDR memory has a complete set of dedicated functional pins and has no flow control.

All other parallel peripherals and memory types (such as SRAM, FLASH, BURST NOR FLASH, NAND FLASH) have dedicated pins and some pins that can be used for other functions such as GPIO.

Address bits 4 to 25 are muxed with GPIO functions on port I and port H and should be selected as address pins before using the asynchronous memory bus. Only the address pins used in the application need to be allocated as address pins. If high-order address bits are not needed, those pins may be used as GPIO. Note however that if booting from a parallel memory source, all 25 address pins and the BG and BGH pins are driven as outputs during boot time.

When using BURST NOR FLASH, select PI15/A25/NR_CLK for the NOR_CLK muxed function.

When using NAND, select PJ2/ND_RB and PJ1/ND_CE for the ND_RB and ND_CE muxed functions.

The bus request flow control pin is also muxed with GPIO functions. To use the asynchronous memory port, PJ11/AMC_BR should be selected for the AMC_BR function. This pin must then be held high with logic or a pulldown resistor to allow bus transactions to be initiated by the processor.

Example Asynchronous Memory Interfaces

This section shows glueless connections to 16-bit 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.

System-Level Hardware Design

Figure 19-1 shows the interface to 8-bit SRAM or FLASH. Figure 19-2 shows the interface to 16-bit SRAM or FLASH

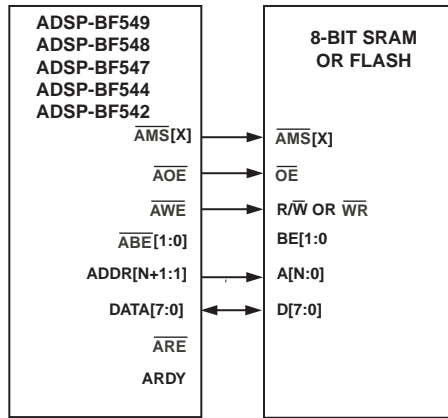


Figure 19-1. Interface to 8-Bit SRAM or FLASH

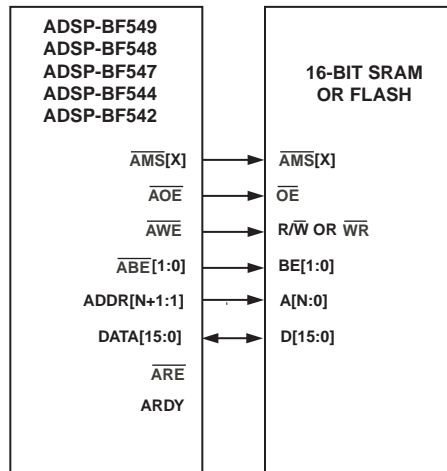


Figure 19-2. Interface to 16-Bit SRAM or FLASH

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. Bus contention can also occur during reset or hibernation. This can be avoided by external resistors to inactivate chip selects.

There are two cases where contention can occur caused by bus timing. 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

System-Level Hardware Design

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.

BURST FLASH

The use of BURST FLASH is similar to that of asynchronous memory but requires special attention.

The burst flash connection requires only three special connections in addition to standard asynchronous memory signals (See [Figure 19-3](#)). The burst clock signal to the flash is provided on `PI15` and is the same pin as `A25`. The second signal requiring special attention is `ADV` provided by the Blackfin processor `A0E` pin. The `WAIT` output of the burst flash that should be connected directly to `ARDY`.

NAND FLASH

NAND FLASH shares the Asynchronous data bus. The NAND FLASH connection has many unique connections (See [Figure 19-4](#)). The chip enable is provided by `PJ1/ND_CE` while the ready busy signal is `PJ2/ND_RB`. `PJ1/ND_CE` requires a pull-up resistor because the general-purpose pins are inputs at power-up. Other unique signal functions include `ND_CLE` provided by `ABE0` and `ND_ALE` provided by `ABE1`. I/O 0 to 7 or I/O 0 to 15 are supplied EBIU data pins `D0` to `D15`.

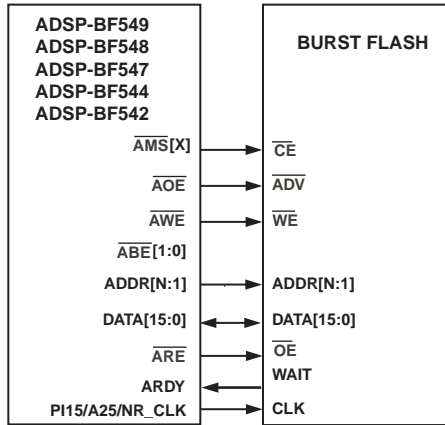


Figure 19-3. Interface to BURST FLASH

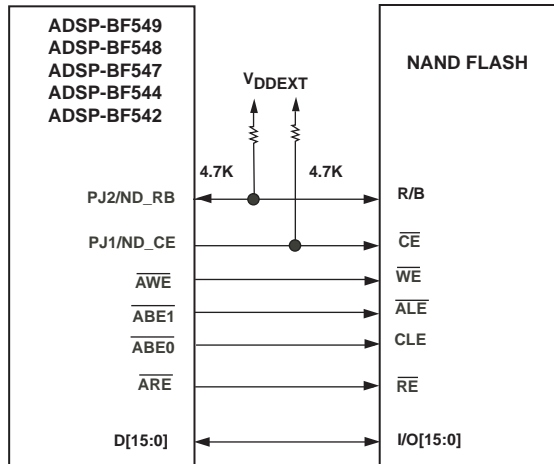


Figure 19-4. Interface to NAND FLASH

USB Controller

The UTMI (universal transceiver macro interface) of the USB controller is unique. It is what some companies call the PHY section of the USB controller. It has many features that allow connection directly to a USB cable connector. The important system hardware requirements are:

- The UTMI section of the USB does not use the system clock. An external clock is needed. The frequency must have an exact multiple to 480 MHz. The default value would be a 24 MHz clock to the `USB_XI` pin or a 24 MHz crystal circuit used with `USB_XI` and `USB_X0`. If using a crystal, use the same circuit as shown in the datasheet for `CLKIN` and `XTAL`.
- The UTMI section of the USB has our standard level of ESD protection. External protection diodes should be added at the connector to DP, DM, ID, and VBUS for proper ESD protection. There are several sources of ESD protection designed specifically for USB2.
- When operating in USB OTG host mode, the user must supply an external 5 V supply at 8 ma or more. The 5 volts can be provided with a “charge pump” or a normal voltage regulator depending on the available input voltages available for the application. In either case it must be enabled and disabled in software using a GPIO with a resistor to set the initial value to disable the external 5 V source. GPIO pin `PE7` is used to enable this regulator in our software examples.
- DP and DM are intended for direct connection to the D+ and D- of a USB cable connector. They do not require any pull-up or pull-down resistors as these are applied internally by the UTMI in accordance with the programmed application mode. Note also that like any USB design, DP and DM should be routed as a differential pair with 90 to 100 Ohms mutual impedance.

- If using the USB in device mode only, you may put a pull-up resistor on `USB_ID` pin or leave the pin disconnected. Either a pull-up resistor or leave the pin disconnected will work.
- The `USB_RST` pin should be connected to an unpopulated resistor as there may be a future advantage to this configuration.
- The `USB_VREF` pin should be connected to a 0.1 μF capacitor to ground.
- As stated in the data sheet, the 5 V tolerance of the `UTMI` pins is only true if V_{DDUSB} has some level of power. Some applications may anticipate V_{BUS} power from a device (perhaps located at the other end of the cable) when the local power is off. If this condition is expected to last for long periods measured in years, precautions should be taken to prevent long term damage to the product. One method for correcting this situation is to use the V_{BUS} power from the external device to power the V_{DDUSB} pins of the processor.

ATAPI Bus

Special care is needed for ATAPI connections that require 5 V logic. Active voltage level translation buffers are required for any peripheral that uses 5 V logic levels.

Voltage Regulator

An internal voltage regulator can be used with the recommended external circuit to provide a flexible system of power management. Many applications require a fixed internal voltage value and can use a simple external voltage regulator to generate the V_{DDINT} supply voltage. The `EXT_WAKE` signal is provided to turn off the external voltage regulator when using the

System-Level Hardware Design

hibernate operating mode. Because it is a high true power-up signal, it may be connected directly to the low true shutdown input of many common regulators.

Signal Integrity

In addition to reducing signal length and capacitive loading, critical signals should be treated like transmission lines.

Use simple signal integrity methods to prevent transmission line reflections that may cause extraneous extra clock and sync signals. Additionally, avoid overshoot and undershoot that can cause long term damage to input pins.

Some signals are especially critical for short trace length and usually require series termination. The `CLKIN` pin should have impedance-matching-series resistance at its driver. SPORT interface signals `TCLK`, `RCLK`, `RFS`, and `TFS` should use some termination. 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 often require resistive termination located at the source. (Note also that `TFS` and `RFS` should not be shorted in multi-channel mode.) On the PPI interface, the `PPI_CLK` and `SYNC` signals also benefit from these standard signal integrity techniques. If these pins have multiple sources, it is difficult to keep the traces short. Consider termination of SDRAM clocks, control, address, and data to improve signal quality and reduce unwanted EMI.

Adding termination to fix a problem on an existing board requires delays for new artwork and new boards. A transmission line simulator is recommended for critical signals. IBIS models are available from Analog Devices Inc. that will assist signal simulation software. Some signals can be corrected with a small zero or 22 Ohm resistor located near the driver. The resistor value can be adjusted after measuring the signal at all endpoints.

For details, see the reference sources in [“Recommended Reading” on page 19-19](#) for suggestions on transmission line termination.

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.
- 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.
- 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 V_{DDEXT} and V_{DDINT} pins of the package as shown in [Figure 19-5](#). 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.

Connect the power plane to the power supply pins directly with minimum trace length. A ground plane should be located near the component side of the board to reduce the distance that ground current must travel through vias. The ground planes must not be densely perforated with vias or traces as their effectiveness is reduced.

V_{DDINT} is the highest frequency and requires special attention. Two things help power filtering above 100 MHz. First, capacitors should be physically small to reduce the inductance. Surface-mount capacitors of size 0402 give better results than larger sizes. Secondly, lower values of

System-Level Hardware Design

capacitance raises the resonant frequency of the LC circuit. While a cluster of $0.1\mu\text{F}$ is acceptable below 50 MHz, a mix of 0.1, 0.01, $0.001\mu\text{F}$ and even 100 pF is preferred in the 500 MHz range.

Note that the instantaneous voltage on both internal and external power pins must at all times be within the recommended operating conditions as specified in the product data sheet. Local “bulk capacitance” (many microfarads) is also necessary. Although all capacitors should be kept close to the power consuming device, small capacitance values should be the closest. Larger values may be placed further from the chip.

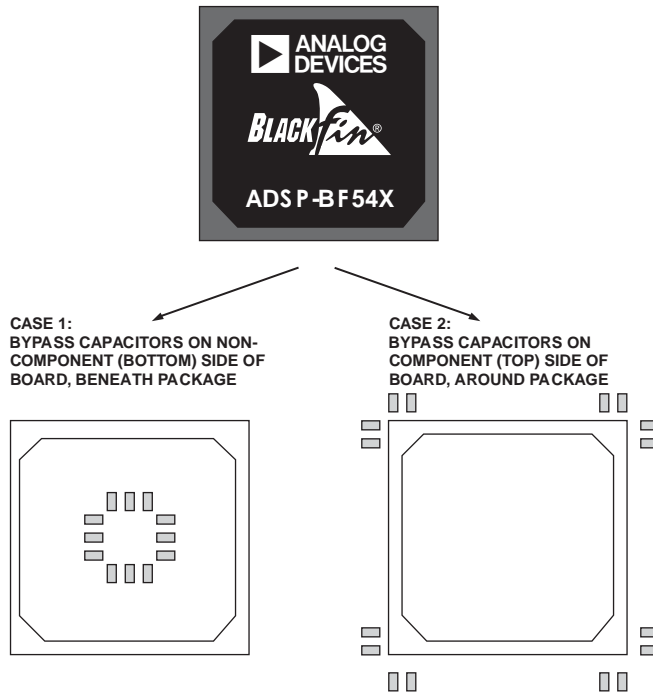


Figure 19-5. Bypass Capacitor Placement

5 Volt Tolerance

Outputs that connect to inputs on 5 V devices can float or be pulled up to 5 V. Only the few pins listed as 5 V tolerant in the data sheet should be subjected to 5 volts. Current limiting resistors are not sufficient to maintain long-term reliability. Level shifters are required on all other Blackfin pins to keep the pin voltage at or below absolute maximum ratings.


Resetting the Processor

The reset pin requires a monotonic rise and fall. Therefore the pin should not be connected directly to an R/C time delay because such a circuit could be noise-sensitive. In addition to the hardware reset mode provided through the `RESET` pin, the processor supports several software reset modes.

Recommendations for Unused Pins

Most often, there is no need to terminate unused pins, but the handful that do require termination are listed at the end of the pin list description section of the product data sheet.

If the real-time clock is not used, `RTXI` should be pulled low. Also note that unused peripherals may have separate power connections. These should be driven to the specified value.

 Peripheral specific power pins require power and ground even when the peripheral is not used.

Programmable Outputs and Pin Multiplexing

During power-up, each GPIO pin is set to an input and any pins used in the system as an output should be connected to a pullup or pulldown resistor to maintain the desired state.

This would be particularly important in motor drive applications. It is also important for UART TX and `RTS`, CAN TX, SPI and serial TWI, ATAPI and other communications interfaces.

Boot Modes that use `HWAIT` require a pullup or pulldown resistor on `PB11` on the ADSP-BF54x processors. `HWAIT` is driven both high and low during all boot cycles and may cause contention or unwanted values if also used as a GPIO.

After the boot cycle, GPIO pins may already be set to input or output depending on ADSP-BF54x family number and the boot cycle chosen. The I/O/GPIO muxing of all pins may need to be reprogrammed to support the users application. Care should be taken for compatibility of function and state, before boot, during boot, and during application pin usage.

Test Point Access

The debug process is aided by test points on signals such as CLKOUT or SCLK, bank selects, PPICLK, and $\overline{\text{RESET}}$. If selection pins such as boot mode are connected directly to power or ground, they are inaccessible under a BGA chip. Use pull-up and pull-down resistors instead.

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

For more information, refer to *High-Speed Digital Design: A Handbook of Black Magic*, Johnson & Graham, Prentice Hall, Inc., ISBN 0-13-395724-1.

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
- Ribbon cables
- Clock distribution
- Clock oscillators

Consult your CAD software tools vendor. Some companies offer demonstration versions of signal integrity software. Simply by using their free software, you can learn:

- Transmission lines are real
- Unterminated printed circuit board traces ring and have overshoot and undershoot
- Simple termination controls signal integrity problems

G GLOSSARY

ALU.

See *Arithmetic/Logic Unit*

AMC (Asynchronous Memory Controller).

A configurable memory controller supporting multiple banks of asynchronous memory including SRAM, ROM, and flash, where each bank can be independently programmed with different timing parameters.

Arithmetic/Logic Unit (ALU).

A processor component that performs arithmetic, comparative, and logical functions.

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, it 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.

base address.

The starting address of a circular buffer.

base register.

A Data Address Generator (DAG) register that contains the starting address for a circular buffer.

bit-reversed addressing.

The addressing mode in which the Data Address Generator (DAG) provides a bit-reversed address during a data move without reversing the stored address.

Boot memory space.

Internal memory space designated for a program that is executed immediately after powerup or after a software reset.

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 followed by a NOP (no operation) command, respectively (Number of NOPs = $\text{burst length} - 1$). Burst lengths of full page, 8, 4, 2, and 1 (no burst) are available. The burst length is selected by writing the BL bits in the SDRAM's mode register during the SDRAM powerup sequence.

Burst Stop command.

The burst stop command is one of several ways to terminate a burst read or write operation.

burst type.

The burst type determines the address order in which the SDRAM delivers burst data. The burst type is selected by writing the BT bits in the SDRAM's mode register during the SDRAM powerup sequence.

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 document, *cache line* is used for *cache block*.

cache miss.

A memory access that does not match any valid entry in the cache.

cache tag.

Upper address bits, stored along with the cached data line, to identify the specific address source in memory that the cached line represents.

Cacheability Protection Lookaside Buffer (CPLB).

Storage area that describes the access characteristics of the core memory map.

CAM (Content Addressable Memory).

Also called associative memory. A memory device that includes comparison logic with each bit of storage. A data value is broadcast to all words in memory; it is compared with the stored values; and values that match are flagged.

CAS (Column Address Strobe).

A signal sent from the SDC to a DRAM device to indicate that the column address lines are valid.

CAS latency (also t_{AA} , t_{CAC} , CL).

The CAS latency or read latency specifies the time between latching a read address and driving the data off chip. This spec is normalized to the system clock and varies from 2 to 3 cycles based on the speed. The CAS latency is selected by writing the CL bits in the SDRAM's mode register during the SDRAM powerup sequence.

CBR (CAS Before RAS) memory refresh.

DRAM devices have a built-in counter for the refresh row address. By activating Column Address Strobe (CAS) before activating Row Address Strobe (RAS), this counter is selected to supply the row address instead of the address inputs.

CEC.

See *Core Event Controller*

circular addressing.

The process by which the Data Address Generator (DAG) “wraps around” or repeatedly steps through a range of registers.

companding.

(Compressing/expanding). The process of logarithmically encoding and decoding data to minimize the number of bits that must be sent.

conditional branches.

Jump or call/return instructions whose execution is based on defined conditions.

core.

The core consists of these functional blocks: CPU, L1 memory, event controller, core timer, and performance monitoring registers.

Core Event Controller (CEC).

The CEC works with the System Interrupt Controller (SIC) to prioritize and control all system interrupts. The CEC handles general-purpose interrupts and interrupts routed from the SIC.

CPLB.

See *Cacheability Protection Lookaside Buffer*

DAB.

See *DMA Access Bus*

DAG.

See *Data Address Generator*

Data Address Generator (DAG).

Processing component that provides memory addresses when data is transferred between memory and registers.

Data Register File.

A set of data registers that is used to transfer data between computation units and memory while providing local storage for operands.

data registers (Dreg).

Registers located in the data arithmetic unit that hold operands and results for multiplier, ALU, or shifter operations.

DCB.

See *DMA Core Bus*

DEB.

See *DMA External Bus*

descriptor block, DMA.

A set of parameters used by the direct memory access (DMA) controller to describe a set of DMA sequences.

descriptor loading, DMA.

The process in which the direct memory access (DMA) controller downloads a DMA descriptor from data memory and autoinitializes the DMA parameter registers.

DFT (Design For Testability).

A set of techniques that helps designers of digital systems ensure that those systems are testable.

Digital Signal Processor (DSP).

An integrated circuit designated for high-speed manipulation of analog information that is converted into digital form.

direct branches.

Jump or call/return instructions that use absolute addresses that do not change at runtime (such as a program label), or they use a PC-relative address.

direct-mapped.

Cache architecture where each line has only one place that it can appear in the cache. Also described as 1-way associative.

Direct Memory Access (DMA).

A way of moving data between system devices and memory in which the data is transferred through a DMA port without involving the processor.

dirty, modified.

A state bit, stored along with the tag, indicating whether the data in the data cache line is changed since it was copied from the source memory and, therefore, needs to be updated in that source memory.

DMA.

See *Direct Memory Access*

DMA Access Bus (DAB).

A bus that provides a means for DMA channels to be accessed by the peripherals.

DMA chaining.

The linking or chaining of multiple direct memory access (DMA) sequences. In chained DMA, the I/O processor loads the next DMA descriptor into the DMA parameter registers when the current DMA finishes and autoinitializes the next DMA sequence.

DMA Core Bus (DCB).

A bus that provides a means for DMA channels to gain access to on-chip memory.

DMA descriptor registers.

Registers that hold the initialization information for a direct memory access (DMA) process.

DMA External Bus (DEB).

A bus that provides a means for DMA channels to gain access to off-chip memory.

DPMC (Dynamic Power Management Controller).

A processor's control block that allows the user to dynamically control the processor's performance characteristics and power dissipation.

DQM Data I/O Mask Function.

The `SDQM[1:0]` pins provide a byte-masking capability on 8-bit writes to SDRAM.

DRAM (Dynamic Random Access Memory).

A type of semiconductor memory in which the data is stored as electrical charges in an array of cells, each consisting of a capacitor and a transistor. The cells are arranged on a chip in a grid of rows and columns. Since the capacitors discharge gradually—and the cells lose their information—the array of cells has to be refreshed periodically.

DSP.

See *Digital Signal Processor*

EAB.

See *External Access Bus*

EBC.

See *External Bus Controller*

EBIU.

See *External Bus Interface Unit*

edge-sensitive interrupt.

A signal or interrupt the processor detects if the input signal is high (inactive) on one cycle and low (active) on the next cycle when sampled on the rising edge of `CLKIN`.

Endian format.

The ordering of bytes in a multibyte number.

EPB.

See *External Port Bus*

EPROM (Erasable Programmable Read-Only Memory).

A type of semiconductor memory in which the data is stored as electrical charges in isolated (“floating”) transistor gates that retain their charges almost indefinitely without an external power supply. An EPROM is programmed by “injecting” charge into the floating gates—a process that requires relatively high voltage (usually 12V – 25V). Ultraviolet light, applied to the chip’s surface through a quartz window in the package, discharges the floating gates, allowing the chip to be reprogrammed.

EVT (Event Vector Table).

A table stored in memory that contains sixteen 32-bit entries; each entry contains a vector address for an interrupt service routine (ISR). When an event occurs, instruction fetch starts at the address location in the corresponding EVT entry. See *ISR*.

exclusive, clean.

The state of a data cache line indicating the line is valid and 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.

External Access Bus (EAB).

A bus mastered by the core memory management unit to access external memory.

External Bus Controller (EBC).

A component that provides arbitration between the External Access Bus (EAB) and the DMA External Bus (DEB), granting at most one requester per cycle.

External Bus Interface Unit (EBIU).

A component that provides glueless interfaces to external memories. It services requests for external memory from the core or from a DMA channel.

external port.

A channel or port that extends the processor's internal address and data buses off-chip, providing the processor's interface to off-chip memory and peripherals.

External Port Bus (EPB).

A bus that connects the output of the EBIU to external devices.

FFT (Fast Fourier Transform).

An algorithm for computing the Fourier transform of a set of discrete data values. The FFT expresses a finite set of data points, for example a periodic sampling of a real-world signal, in terms of its component frequencies. Or conversely, the FFT reconstructs a signal from the frequency data. The FFT can also be used to multiply two polynomials.

FIFO (First In, First Out).

A hardware buffer or data structure from which items are taken out in the same order they were put in.

flash memory.

A type of single transistor cell, erasable memory in which erasing can only be done in blocks or for the entire chip.

fully associative.

Cache architecture where each line can be placed anywhere in the cache.

glueless.

No external hardware is required.

Harvard architecture.

A processor memory architecture that uses separate buses for program and data storage. The two buses let the processor fetch a data word and an instruction word simultaneously.

HLL (High Level Language).

A programming language that provides some level of abstraction above assembly language, often using English-like statements, where each command or statement corresponds to several machine instructions.

I²C.

A bus standard specified in the *Philips I²C Bus Specification version 2.1* dated January 2000.

IDLE.

An instruction that causes the processor to cease operations, holding its current state until an interrupt occurs. Then, the processor services the interrupt and continues normal execution.

index.

Address portion that is used to select an array element (for example, line index).

Index registers.

A Data Address Generator (DAG) register that holds an address and acts as a pointer to memory.

indirect branches.

Jump or call/return instructions that use a dynamic address from the data address generator, evaluated at runtime.

input clock.

Device that generates a steady stream of timing signals to provide the frequency, duty cycle, and stability to allow accurate internal clock multiplication through the phase locked loop (PLL) module.

internal memory bank.

There are up to 4 internal memory banks on a given SDRAM. Each of these banks can be accessed with the bank select lines $BA[1:0]$. The bank address can be thought of as part of the row address.

interrupt.

An event that suspends normal processing and temporarily diverts the flow of control through an interrupt service routine (ISR). See *ISR*.

invalid.

Describes the state of a cache line. When a cache line is invalid, a cache line match cannot occur.

IrDA (Infrared Data Association).

A nonprofit trade association that established standards for ensuring the quality and interoperability of devices using the infrared spectrum.

isochronous.

Processes where data must be delivered within certain time constraints.

ISR (Interrupt Service Routine).

Software that is executed when a specific interrupt occurs. A table stored in low memory contains pointers, also called vectors, that direct the processor to the corresponding ISR. See *EVT*.

JTAG (Joint Test Action Group).

An IEEE Standards working group that defines the IEEE 1149.1 standard for a test access port for testing electronic devices.

JTAG port.

A channel or port that supports the IEEE standard 1149.1 JTAG standard for system test. This standard defines a method for serially scanning the I/O status of each component in a system.

jump.

A permanent transfer of the program flow to another part of program memory.

latency.

The overhead time used to find the correct place for memory access and preparing to access it.

Least Recently Used algorithm.

Replacement algorithm used by cache that first replaces lines that have been unused for the longest time.

Least Significant Bit (LSB).

The last or rightmost bit in the normal representation of a binary number—the bit of a binary number giving the number of ones.

Length registers.

A Data Address Generator (DAG) register that specifies the range of addresses in a circular buffer.

Level 1 (L1) memory.

Memory that is directly accessed by the core with no intervening memory subsystems between it and the core.

Level 2 (L2) memory.

Memory that is at least one level removed from the core. L2 memory has a larger capacity than L1 memory, but it requires additional latency to access.

level-sensitive interrupts.

A signal or interrupt that the processor detects if the input signal is low (active) when sampled on the rising edge of `CLKIN`.

LIFO (Last In, First Out).

A data structure from which the next item taken out is the most recent item put in.

little endian.

The native data store format of the 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 at the highest byte address of the data storage location.

loop.

A sequence of instructions that executes several times.

LRU.

See *Least Recently Used algorithm*

LSB.

See *Least Significant Bit*

MAC (Media Access Control).

The Ethernet MAC provides a 10/100Mbit/s Ethernet interface, compliant to IEEE Std. 802.3-2002, between an MII (Media Independent Interface) and the Blackfin peripheral subsystem.

MAC (Multiply/Accumulate).

A mathematical operation that multiplies two numbers and then adds a third to get the result (see *Multiply Accumulator*).

Memory Management Unit (MMU).

A component of the processor that supports protection and selective caching of memory by using cacheability protection lookaside buffers (CPLBs).

Mode register.

Internal configuration registers within SDRAM devices which allow specification of the SDRAM device's functionality.

modified addressing.

The process whereby the Data Address Generator (DAG) produces an address that is incremented by a value or the contents of a register.

Modify register.

A Data Address Generator (DAG) register that provides the increment or step size by which an index register is pre- or post-modified during a register move.

MMR (Memory-Mapped Register).

A specific location in main memory used by the processor as if it were a register.

MMU.

See *Memory Management Unit*

MSB (Most Significant Bit).

The first or leftmost bit in the normal representation of a binary number—the bit of a binary number with the greatest weight ($2^{(n-1)}$).

multifunction computations.

The parallel execution of multiple computational instructions. These instructions complete in a single cycle, and they combine parallel operation of the computational units and memory accesses. The multiple operations perform the same as if they were in corresponding single function computations.

multiplier.

A computational unit that performs fixed-point multiplication and executes fixed-point multiply/add and multiply/subtract operations.

NMI (Nonmaskable Interrupt).

A high priority interrupt that cannot be disabled by another interrupt.

NRZ (Non-return-to-Zero).

A binary encoding scheme in which a 1 is represented by a change in the signal and a 0 by no change—there is no return to a reference (0) voltage between encoded bits. This method eliminates the need for a clock signal.

NRZI (Non-return-to-Zero Inverted).

A binary encoding scheme in which a 0 is represented by a change in the signal and a 1 is represented by no change—there is no return to a reference (0) voltage between encoded bits. This method eliminates the need for a clock signal.

orthogonal.

The characteristic of being independent. An orthogonal instruction set allows any register to be used in an instruction that references a register.

PAB.

See *Peripheral Access Bus*

page size.

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.

Parallel Peripheral Interface (PPI).

The PPI is a half-duplex, bidirectional port accommodating up to 16 bits of data. It has a dedicated clock pin and three multiplexed frame sync pins.

PC (Program Counter).

A register that contains the address of the next instruction to be executed.

peripheral.

Functional blocks not included as part of the core, and typically used to support system level operations.

Peripheral Access Bus (PAB).

A bus used to provide access to EBIU memory-mapped registers.

PF (Programmable Flag).

General-purpose I/O pins. Each PF pin can be individually configured as either an input or an output pin, and each PF pin can be further configured to generate an interrupt.

Phase-Locked Loop (PLL).

An on-chip frequency synthesizer that produces a full speed master clock from a lower frequency input clock signal.

PLL.

See *Phase-Locked Loop*

PPI.

See *Parallel Peripheral Interface*

precision.

The number of bits after the binary point in the storage format for the number.

post-modify addressing.

The process in which the Data Address Generator (DAG) provides an address during a data move and auto-increments after the instruction is executed.

precharge command.

The precharge command closes a specific active page in an internal bank and the precharge all command closes all 4 active pages in all 4 banks.

pre-modify addressing.

The process in which the Data Address Generator (DAG) provides an address during a data move and auto-increments before the instruction is executed.

PWM (Pulse Width Modulation).

Also called Pulse Duration Modulation (PDM), PWM is a pulse modulation technique in which the duration of the pulses is varied by the modulating voltage.

RAS (Row Address Strobe).

A signal sent from the SDC to a DRAM device to indicate validity of row address lines.

Real-Time Clock (RTC).

A component that generates timing pulses for the digital watch features of the processor, including time of day, alarm, and stopwatch countdown features.

ROM (Read-Only Memory).

A data storage device manufactured with fixed contents. This term is most often used to refer to non-volatile semiconductor memory.

RTC.

See *Real-Time Clock*

RZ (Return-to-Zero modulation).

A binary encoding scheme in which two signal pulses are used for every bit. A 0 is represented by a change from the low voltage level to the high voltage level; a 1 is represented by a change from the high voltage level to the low voltage level. A return to a reference (0) voltage is made between encoded bits.

RZI (Return-to-Zero-Inverted modulation).

A binary encoding scheme in which two signal pulses are used for every bit. A 1 is represented by a change from the low voltage level to the high voltage level; a 0 is represented by a change from the high voltage level to the low voltage level. A return to a reference (0) voltage is made between encoded bits.

saturation (ALU saturation mode).

A state in which all positive fixed-point overflows return the maximum positive fixed-point number, and all negative overflows return the maximum negative number.

SDC (SDRAM Controller).

A configurable memory controller supporting a bank of synchronous memory consisting of SDRAM.

SDRAM (Synchronous Dynamic Random Access Memory).

A form of DRAM that includes a clock signal with its other control signals. This clock signal allows SDRAM devices to support “burst” access modes that clock out a series of successive bits.

SDRAM bank.

Region of external memory that can be configured to be 16M bytes, 32M bytes, 64M bytes, or 128M bytes and is selected by the \overline{SMS} pin.

Serial Peripheral Interface (SPI).

A synchronous serial protocol used to connect integrated circuits.

serial ports (SPORTs).

A high speed synchronous input/output device on the processor. The processor uses two synchronous serial ports that provide inexpensive interfaces to a wide variety of digital and mixed-signal peripheral devices.

set.

A group of N -line storage locations in the ways of an N -way cache, selected by the index field of the address.

set associative.

Cache architecture that limits line placement to a number of sets (or ways).

shifter.

A computational unit that completes logical and arithmetic shifts.

SIC (System Interrupt Controller).

Part of the processor's two-level event control mechanism. The SIC works with the Core Event Controller (CEC) to prioritize and control all system interrupts. The SIC provides mapping between the peripheral interrupt sources and the prioritized general-purpose interrupt inputs of the core.

SIMD (Single Instruction, Multiple Data).

A parallel computer architecture in which multiple data operands are processed simultaneously using one instruction.

SP (Stack Pointer).

A register that points to the top of the stack.

SPI.

See *Serial Peripheral Interface*

SRAM.

See *Static Random Access Memory*

stack.

A data structure for storing items that are to be accessed in Last In, First Out (LIFO) order. When a data item is added to the stack, it is “pushed”; when a data item is removed from the stack, it is “popped.”

Static Random Access Memory (SRAM).

Very fast read/write memory that does not require periodic refreshing.

system.

The system includes the peripheral set (timers, real-time clock, programmable flags, UART, SPORTs, PPI, and SPIs), the external memory controller (EBIU), the memory DMA controller, as well as the interfaces between these peripherals, and the optional, external (off-chip) resources.

System clock (SCLK).

A component that delivers clock pulses at a frequency determined by a programmable divider ratio within the PLL.

System Interrupt Controller (SIC).

Component that maps and routes events from peripheral interrupt sources to the prioritized, general-purpose interrupt inputs of the Core Event Controller (CEC).

TAP (Test Access Port).

See *JTAG port*

TDM.

See *Time Division Multiplexing*

Time Division Multiplexing (TDM).

A method used for transmitting separate signals over a single channel. Transmission time is broken into segments, each of which carries one element. Each word belongs to the next consecutive channel so that, for example, a 24-word block of data contains one word for each of the 24 channels.

TWI.

See *Two Wire Interface*

Two Wire Interface (TWI).

The TWI controller allows a device to interface to an Inter IC bus as specified by the *Philips I²C Bus Specification version 2.1* dated January 2000. The interface is essentially a shift register that serially transmits and receives data bits, one bit at a time at the SCL rate, to and from other TWI devices.

UART.

See *Universal Asynchronous Receiver Transmitter*

Universal Asynchronous Receiver Transmitter (UART).

A module that contains both the receiving and transmitting circuits required for asynchronous serial communication.

Valid.

A state bit (stored along with the tag) that indicates 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.

Von Neumann architecture.

The architecture used by most non-DSP microprocessors. This architecture uses a single address and data bus for memory access.

Way.

An array of line storage elements in an N -way cache.

W1C.

See *Write-1-to-Clear*

W1S.

See *Write-1-to-Set*

Write-1-to-Clear (W1C) bit.

A control or status bit that can be cleared (= 0) by being written to with 1.

Write-1-to-Set (W1S) bit.

A control or status bit that is set by writing 1 to it. It cannot be cleared by writing 0 to it.

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.

write through.

A cache write policy (also known as store through). The write data is written to both the cache line and to source memory. The modified cache line is *not* written to the source memory when it is replaced.

A SYSTEM MMR ASSIGNMENTS

This appendix lists MMR addresses and register names for the system memory-mapped registers (MMRs), the core timer registers, and the processor-specific memory registers mentioned in this manual. To find more information about an MMR, refer to the page shown in the “See Page” column. When viewing the PDF version of this document, click a reference in the “See Page” column to jump to additional information about the MMR.

This chapter includes the following sections:

- [“Dynamic Power Management Registers” on page A-3](#)
- [“System Reset and Interrupt Control Registers” on page A-4](#)
- [“Watchdog Timer Registers” on page A-6](#)
- [“Real-Time Clock Registers” on page A-6](#)
- [“UART0 Controller Registers” on page A-7](#)
- [“SPI0 Controller Registers” on page A-8](#)
- [“TWI Controller Registers” on page A-8](#)
- [“SPORT0 Controller Registers” on page A-8](#)
- [“MXVR Registers” on page A-9](#)
- [“Keypad Registers” on page A-9](#)
- [“SDH Registers” on page A-10](#)

- “ATAPI Registers” on page A-10
- “USB_OTG Registers” on page A-10
- “External Bus Interface Unit Registers” on page A-10
- “DMA/Memory DMA Control Registers” on page A-12
- “EPPI0 Registers” on page A-14
- “Host DMA Registers” on page A-15
- “PIXC Registers” on page A-15
- “Ports Registers” on page A-17
- “Timer Registers” on page A-26
- “CAN Registers” on page A-28
- “Handshake MDMA Control Registers” on page A-29
- “NAND Flash Controller Registers” on page A-30
- “Core Timer Registers” on page A-31
- “Rotary Counter Registers” on page A-31
- “Security Registers” on page A-32
- “Processor-Specific Memory Registers” on page A-33

These notes provide general information about the system memory-mapped registers (MMRs):

- The system MMR address range is 0xFFC0 0000 – 0xFFDF 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.

Dynamic Power Management Registers

Dynamic power management registers (0xFFC0 0000 – 0xFFC0 00FF) are listed in [Table A-1](#).

Table A-1. Dynamic Power Management Registers

Memory-Mapped Address	Register Name	See Page
0xFFC0 0000	PLL_CTL	“PLL Control (PLL_CTL) Register” on page 18-28
0xFFC0 0004	PLL_DIV	“PLL Divide (PLL_DIV) Register” on page 18-27
0xFFC0 0008	VR_CTL	“Voltage Regulator Control (VR_CTL) Register” on page 18-30
0xFFC0 000C	PLL_STAT	“PLL Status (PLL_STAT) Register” on page 18-29
0xFFC0 0010	PLL_LOCKCNT	“PLL Lock Count (PLL_LOCKCNT) Register” on page 18-29

System Reset and Interrupt Control Registers

System reset and interrupt control registers (0xFFC0 0100 – 0xFFC0 01FF) are listed in [Table A-2](#).

Table A-2. System Reset and Interrupt Control Registers

Memory-Mapped Address	Register Name	See Page
0xFFC0 0100	SYRST	“Software Reset Register” on page 17-108
0xFFC0 0104	SYSCR	“System Reset Configuration Register” on page 17-110
0xFFC0 010C	SIC_IMASK0	“System Interrupt Mask Register 0 (SIC_IMASK0)” on page 4-35
0xFFC0 0110	SIC_IMASK1	“System Interrupt Mask Register 1 (SIC_IMASK1)” on page 4-36
0xFFC0 01114	SIC_IMASK2	“System Interrupt Mask Register 2 (SIC_IMASK2)” on page 4-37
0xFFC0 0130	SIC_IAR0	“System Interrupt Assignment Register 0 (SIC_IAR0)” on page 4-28
0xFFC0 0134	SIC_IAR1	“System Interrupt Assignment Register 1 (SIC_IAR1)” on page 4-28
0xFFC0 0138	SIC_IAR2	“System Interrupt Assignment Register 2 (SIC_IAR2)” on page 4-29
0xFFC0 013C	SIC_IAR3	“System Interrupt Assignment Register 3 (SIC_IAR3)” on page 4-29
0xFFC0 0140	SIC_IAR4	“System Interrupt Assignment Register 4 (SIC_IAR4)” on page 4-30
0xFFC0 0144	SIC_IAR5	“System Interrupt Assignment Register 5 (SIC_IAR5)” on page 4-30
0xFFC0 0148	SIC_IAR6	“System Interrupt Assignment Register 6 (SIC_IAR6)” on page 4-31

Table A-2. System Reset and Interrupt Control Registers (Cont'd)

Memory-Mapped Address	Register Name	See Page
0xFFC0 014C	SIC_IAR7	“System Interrupt Assignment Register 7 (SIC_IAR7)” on page 4-31
0xFFC0 0150	SIC_IAR8	“System Interrupt Assignment Register 8 (SIC_IAR8)” on page 4-32
0xFFC0 0154	SIC_IAR9	“System Interrupt Assignment Register 9 (SIC_IAR9)” on page 4-32
0xFFC0 0158	SIC_IAR10	“System Interrupt Assignment Register 10 (SIC_IAR10)” on page 4-33
0xFFC0 015C	SIC_IAR11	“System Interrupt Assignment Register 11 (SIC_IAR11)” on page 4-33
0xFFC0 0118	SIC_ISR0	“System Interrupt Status Register 0 (SIC_ISR0)” on page 4-38
0xFFC0 011C	SIC_ISR1	“System Interrupt Status Register 1 (SIC_ISR1)” on page 4-39
0xFFC0 0120	SIC_ISR2	“System Interrupt Status Register 2 (SIC_ISR2)” on page 4-40
0xFFC0 0124	SIC_IWR0	“System Interrupt Wakeup Register 0 (SIC_IWR0)” on page 4-41
0xFFC0 0128	SIC_IWR1	“System Interrupt Wakeup Register 1 (SIC_IWR1)” on page 4-42
0xFFC0 012C	SIC_IWR2	“System Interrupt Wakeup Register 2 (SIC_IWR2)” on page 4-43

Watchdog Timer Registers

Watchdog timer registers (0xFFC0 0200 – 0xFFC0 02FF) are listed in [Table A-3](#).

Table A-3. Watchdog Timer Registers

Memory-Mapped Address	Register Name	See Page
0xFFC0 0200	WDOG_CTL	“Watchdog Control (WDOG_CTL) Register” on page 12-9
0xFFC0 0204	WDOG_CNT	“Watchdog Count (WDOG_CNT) Register” on page 12-7
0xFFC0 0208	WDOG_STAT	“Watchdog Status (WDOG_STAT) Register” on page 12-8

Real-Time Clock Registers

Real-time clock registers (0xFFC0 0300 – 0xFFC0 03FF) are listed in [Table A-4](#).

Table A-4. Real-Time Clock Registers

Memory-Mapped Address	Register Name	See Page
0xFFC0 0300	RTC_STAT	“RTC Status (RTC_STAT) Register” on page 14-21
0xFFC0 0304	RTC_ICTL	“RTC Interrupt Control (RTC_ICTL) Register” on page 14-21
0xFFC0 0308	RTC_ISTAT	“RTC Interrupt Status (RTC_ISTAT) Register” on page 14-22
0xFFC0 030C	RTC_SWCNT	“RTC Stopwatch Count (RTC_SWCNT) Register” on page 14-22

Table A-4. Real-Time Clock Registers (Cont'd)

Memory-Mapped Address	Register Name	See Page
0xFFC0 0310	RTC_ALARM	“RTC Alarm (RTC_ALARM) Register” on page 14-23
0xFFC0 0314	RTC_PREN	“Prescaler Enable (RTC_PREN) Register” on page 14-23

UART0 Controller Registers

UART0 controller registers (0xFFC0 0400 – 0xFFC0 04FF) are listed in the “System MMR Assignments” appendix in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 2 of 2)*.

UART1 Controller Registers

UART1 controller registers (0xFFC0 2000 – 0xFFC0 20FF) are listed in the “System MMR Assignments” appendix in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 2 of 2)*.

UART2 Controller Registers

UART2 controller registers are listed in the “System MMR Assignments” appendix in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 2 of 2)*.

UART3 Controller Registers

UART1 controller registers are listed in the “System MMR Assignments” appendix in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 2 of 2)*.

SPI0 Controller Registers

SPI0 controller registers (0xFFC0 0500 – 0xFFC0 05FF) are listed in the “System MMR Assignments” appendix in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 2 of 2)*.

SPI1 Controller Registers

SPI1 controller registers are listed in the “System MMR Assignments” appendix in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 2 of 2)*.

TWI Controller Registers

TWIX controller registers are listed in the “System MMR Assignments” appendix in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 2 of 2)*.

SPORT0 Controller Registers

SPORT0 controller registers (0xFFC0 0800 – 0xFFC0 08FF) are listed in the “System MMR Assignments” appendix in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 2 of 2)*.

SPORT1 Controller Registers

SPORT1 controller registers (0xFFC0 0900 – 0xFFC0 09FF) are listed in the “System MMR Assignments” appendix in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 2 of 2)*.

SPORT2 Controller Registers

SPORT2 controller registers are listed in the “System MMR Assignments” appendix in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 2 of 2)*.

SPORT3 Controller Registers

SPORT3 controller registers are listed in the “System MMR Assignments” appendix in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 2 of 2)*.

MXVR Registers

MXVR registers are listed in the “System MMR Assignments” appendix in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 2 of 2)*.

Keypad Registers

Keypad registers are listed in the “System MMR Assignments” appendix in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 2 of 2)*.

SDH Registers

Secure Digital Host (SDH) registers are listed in the “System MMR Assignments” appendix in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 2 of 2)*.

ATAPI Registers

ATAPI registers are listed in the “System MMR Assignments” appendix in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 2 of 2)*.

USB_OTG Registers

USB_OTG registers are listed in the “System MMR Assignments” appendix in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 2 of 2)*.

External Bus Interface Unit Registers

External bus interface unit (EBIU) registers (0xFFC0 0A00 – 0xFFC0 0AFF) are listed in [Table A-5](#).

Table A-5. EBIU Memory-Mapped Registers

Address	Name	Description
0xFFC0 0A00	EBIU_AMGCTL	Asynchronous memory global control register (on page 6-57)
0xFFC0 0A04	EBIU_AMBCTL0	Asynchronous memory bank control 0 register (on page 6-58)
0xFFC0 0A08	EBIU_AMBCTL1	Asynchronous memory bank control 1 register (on page 6-58)

Table A-5. EBIU Memory-Mapped Registers

Address	Name	Description
0xFFC0 0A0C	EBIU_MBSCTL	Memory bank select control register (on page 6-63)
0xFFC0 0A10	EBIU_ARBSTAT	Arbiter status register (on page 6-69)
0xFFC0 0A14	EBIU_MODE	Memory mode control register (on page 6-63)
0xFFC0 0A18	EBIU_FCTL	Flash control register (on page 6-64)
0xFFC0 0A20	EBIU_DDRCTL0	“Memory Control Register 0 (EBIU_DDRCTL0)” on page 6-21
0xFFC0 0A24	EBIU_DDRCTL1	“Memory Control Register 1 (EBIU_DDRCTL1)” on page 6-22
0xFFC0 0A28	EBIU_DDRCTL2	“Memory Control Register 2 (EBIU_DDRCTL2)” on page 6-23
0xFFC0 0A2C	EBIU_DDRCTL3	“Memory Control Register 3 (EBIU_DDRCTL3), Regular DDR Devices” on page 6-24 “Memory Control Register 3 (EBIU_DDRCTL3) Mobile DDR Devices” on page 6-25
0xFFC0 0A30	EBIU_DDRQUE	“Queue Configuration Register (EBIU_DDRQUE)” on page 6-26
0xFFC0 0A34	EBIU_ERRADD	“Error Address Register (EBIU_ERRADD)” on page 6-27
0xFFC0 0A38	EBIU_ERRMST	“Error Master Register (EBIU_ERRMST)” on page 6-28
0xFFC0 0A3C	EBIU_RSTCTL	“Reset Control Register 0 (EBIU_RSTCTL)” on page 6-29
0xFFC0 0A1C	Reserved	Reserved

DMA/Memory DMA Control Registers

DMA/Memory DMA control registers (0xFFC0 0B00 – 0xFFC0 0FFF) are listed in [Table A-6](#).

Table A-6. DMA/Memory DMA Control Registers

Memory-Mapped Address	Register Name	See Page
0xFFC0 0B0C	DMA_TCPER	“DMA Traffic Control Counter Period (DMACx_TCPER) Registers” on page 5-126
0xFFC0 0B10	DMA_TCCNT	“DMA Traffic Control Counter (DMACx_TCCNT) Registers” on page 5-127

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 A-7](#) and [Table A-8](#). [Table A-7](#) identifies the base address of each DMA channel, as well as the register prefix that identifies the channel. [Table A-8](#) 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 A-7. DMA Channel Base Addresses

DMA Channel Identifier	MMR Base Address	Register Prefix
0	0xFFC0 0C00	DMA0_
1	0xFFC0 0C40	DMA1_
2	0xFFC0 0C80	DMA2_
3	0xFFC0 0CC0	DMA3_

Table A-7. DMA Channel Base Addresses (Cont'd)

DMA Channel Identifier	MMR Base Address	Register Prefix
4	0xFFC0 0D00	DMA4_
5	0xFFC0 0D40	DMA5_
6	0xFFC0 0D80	DMA6_
7	0xFFC0 0DC0	DMA7_
8	0xFFC0 0E00	DMA8_
9	0xFFC0 0E40	DMA9_
10	0xFFC0 0E80	DMA10_
11	0xFFC0 0EC0	DMA11_
MemDMA stream 0 destination	0xFFC0 0F00	MDMA_D0_
MemDMA stream 0 source	0xFFC0 0F40	MDMA_S0_
MemDMA stream 1 destination	0xFFC0 0F80	MDMA_D1_
MemDMA stream 1 source	0xFFC0 0FC0	MDMA_S1_

Table A-8. DMA Register Suffix and Offset

Register Suffix	Offset From Base	See Page
NEXT_DESC_PTR	0x00	“Next Descriptor Pointer Registers” on page 5-111
START_ADDR	0x04	“Start Address Registers” on page 5-92
CONFIG	0x08	“DMA Configuration Registers” on page 5-82
X_COUNT	0x10	“Inner Loop Count Registers” on page 5-96
X_MODIFY	0x14	“Inner Loop Address Increment Registers” on page 5-101

EPPIO Registers

Table A-8. DMA Register Suffix and Offset (Cont'd)

Register Suffix	Offset From Base	See Page
Y_COUNT	0x18	“Outer Loop Count Registers” on page 5-103
Y_MODIFY	0x1C	“Outer Loop Address Increment Registers” on page 5-108
CURR_DESC_PTR	0x20	“Current Descriptor Pointer Registers” on page 5-114
CURR_ADDR	0x24	“Current Address Registers” on page 5-94
IRQ_STATUS	0x28	“Interrupt Status Registers” on page 5-88
PERIPHERAL_MAP	0x2C	“Peripheral Map Registers” on page 5-79
CURR_X_COUNT	0x30	“Current Inner Loop Count Registers” on page 5-99
CURR_Y_COUNT	0x38	“Current Outer Loop Count Registers” on page 5-106

EPPIO Registers

PPIO registers (0xFFC0 1000 – 0xFFC0 10FF) are listed in the “System MMR Assignments” appendix in the *ADSP-BF54x Blackfin Peripherals Hardware Reference (Volume 2 of 2)*.

EPPI1 Registers

PPI1 registers are listed in the “System MMR Assignments” appendix in the *ADSP-BF54x Blackfin Peripherals Hardware Reference (Volume 2 of 2)*.

Host DMA Registers

Host DMA registers are listed in [Table A-9](#).

Table A-9. Host DMA Registers

Memory-Mapped Address	Register Name	See Page
0xFFC0 3A00	HOST_CONTROL	“HOSTDP Control (HOST_CONTROL) Register” on page 8-25
0xFFC0 3A04	HOST_STATUS	“HOSTDP Status (HOST_STATUS) Register” on page 8-27
0xFFC0 3A08	HOST_TIMEOUT	“HOSTDP Timeout (HOST_TIMEOUT) Register” on page 8-29

PIXC Registers

The Pixel Compositor has memory-mapped registers (MMRs) that regulate its operation. These registers are listed in [Table A-10](#). Descriptions and bit diagrams for each of these MMRs are provided in the following sections.

Table A-10. PIXC Memory-Mapped Registers

Memory-mapped Address	Register Name	Function
0xFFC04400	PIXC_CTL on page 7-38	Overlay enable, resampling mode selection, input/output data format selection, transparent color enable, watermark level selection, image/overlay FIFO status.
0xFFC04404	PIXC_PPL on page 7-39	Holds the number of pixels per line of the display.
0xFFC04408	PIXC_LPF on page 7-39	Holds the number of lines per frame of the display.

PIXC Registers

Table A-10. PIXC Memory-Mapped Registers (Cont'd)

Memory-mapped Address	Register Name	Function
0xFFC0440C	PIXC_AHSTART on page 7-40	Contains horizontal start pixel information of the overlay data (set A).
0xFFC04410	PIXC_AHEND on page 7-40	Contains horizontal end pixel information of the overlay data (set A).
0xFFC04414	PIXC_AVSTART on page 7-41	Contains vertical start pixel information of the overlay data (set A).
0xFFC04418	PIXC_AVEND on page 7-41	Contains vertical end pixel information of the overlay data (set A).
0xFFC0441C	PIXC_ATRANSP on page 7-42	Contains the transparency ratio (set A).
0xFFC04420	PIXC_BHSTART on page 7-40	Contains horizontal start pixel information of the overlay data (set B).
0xFFC04424	PIXC_BHEND on page 7-40	Contains horizontal end pixel information of the overlay data (set B).
0xFFC04428	PIXC_BVSTART on page 7-41	Contains vertical start pixel information of the overlay data (set B).
0xFFC0442C	PIXC_BVEND on page 7-41	Contains vertical end pixel information of the overlay data (set B).
0xFFC04430	PIXC_BTRANSP on page 7-42	Contains the transparency ratio (set B).
0xFFC0443C	PIXC_INTRSTAT on page 7-42	Overlay interrupt configuration/status.
0xFFC04440	PIXC_RYCON on page 7-43	Color space conversion matrix register. Contains the R/Y conversion coefficients.
0xFFC04444	PIXC_GUCON on page 7-44	Color space conversion matrix register. Contains the G/U conversion coefficients.
0xFFC04448	PIXC_BVCON on page 7-45	Color space conversion matrix register. Contains the B/V conversion coefficients.

Table A-10. PIXC Memory-Mapped Registers (Cont'd)

Memory-mapped Address	Register Name	Function
0xFFC0444C	PIXC_CCBIAS on page 7-46	Bias values for the color space conversion matrix.
0xFFC04450	PIXC_TC on page 7-47	Holds the transparent color value.

Ports Registers

[Table A-11](#) lists the registers for port control and [Table A-12](#) lists the registers for pin interrupt programming.

Table A-11. Port Control Registers (Multiplexing and GPIO)

Memory-mapped Address	Register Name	More Information begins ...
0xFFC014C0	PORTA_FER	“Port Control Registers (Multiplexing and GPIO)” on page 9-30
0xFFC014C4	PORTA	“Port Control Registers (Multiplexing and GPIO)” on page 9-30
0xFFC014C8	PORTA_SET	“Port Control Registers (Multiplexing and GPIO)” on page 9-30
0xFFC014CC	PORTA_CLEAR	“Port Control Registers (Multiplexing and GPIO)” on page 9-30
0xFFC014D0	PORTA_DIR_SET	“Port Control Registers (Multiplexing and GPIO)” on page 9-30
0xFFC014D4	PORTA_DIR_CLEAR	“Port Control Registers (Multiplexing and GPIO)” on page 9-30
0xFFC014D8	PORTA_INEN	“Port Control Registers (Multiplexing and GPIO)” on page 9-30
0xFFC014DC	PORTA_MUX	“Port Control Registers (Multiplexing and GPIO)” on page 9-30

Ports Registers

Table A-11. Port Control Registers (Multiplexing and GPIO) (Cont'd)

Memory-mapped Address	Register Name	More Information begins ...
0xFFC014E0	PORTB_FER	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC014E4	PORTB	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC014E8	PORTB_SET	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC014EC	PORTB_CLEAR	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC014F0	PORTB_DIR_SET	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC014F4	PORTB_DIR_CLEAR	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC014F8	PORTB_INEN	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC014FC	PORTB_MUX	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC01500	PORTC_FER	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC01504	PORTC	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC01508	PORTC_SET	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC0150C	PORTC_CLEAR	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC01510	PORTC_DIR_SET	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC01514	PORTC_DIR_CLEAR	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC01518	PORTC_INEN	"Port Control Registers (Multiplexing and GPIO)" on page 9-30

Table A-11. Port Control Registers (Multiplexing and GPIO) (Cont'd)

Memory-mapped Address	Register Name	More Information begins ...
0xFFC0151C	PORTC_MUX	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC01520	PORTD_FER	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC01524	PORTD	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC01528	PORTD_SET	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC0152C	PORTD_CLEAR	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC01530	PORTD_DIR_SET	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC01534	PORTD_DIR_CLEAR	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC01538	PORTD_INEN	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC0153C	PORTD_MUX	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC01540	PORTE_FER	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC01544	PORTE	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC01548	PORTE_SET	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC0154C	PORTE_CLEAR	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC01550	PORTE_DIR_SET	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC01554	PORTE_DIR_CLEAR	"Port Control Registers (Multiplexing and GPIO)" on page 9-30

Ports Registers

Table A-11. Port Control Registers (Multiplexing and GPIO) (Cont'd)

Memory-mapped Address	Register Name	More Information begins ...
0xFFC01558	PORTE_INEN	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC0155C	PORTE_MUX	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC01560	PORTF_FER	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC01564	PORTF	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC01568	PORTF_SET	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC0156C	PORTF_CLEAR	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC01570	PORTF_DIR_SET	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC01574	PORTF_DIR_CLEAR	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC01578	PORTF_INEN	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC0157C	PORTF_MUX	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC01580	PORTG_FER	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC01584	PORTG_DIR_CLEAR	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC01588	PORTG_SET	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC0158C	PORTG_CLEAR	"Port Control Registers (Multiplexing and GPIO)" on page 9-30
0xFFC01590	PORTG_DIR_SET	"Port Control Registers (Multiplexing and GPIO)" on page 9-30

Table A-11. Port Control Registers (Multiplexing and GPIO) (Cont'd)

Memory-mapped Address	Register Name	More Information begins ...
0xFFC01594	PORTG	“Port Control Registers (Multiplexing and GPIO)” on page 9-30
0xFFC01598	PORTG_INEN	“Port Control Registers (Multiplexing and GPIO)” on page 9-30
0xFFC0159C	PORTG_MUX	“Port Control Registers (Multiplexing and GPIO)” on page 9-30
0xFFC015A0	PORTH_FER	“Port Control Registers (Multiplexing and GPIO)” on page 9-30
0xFFC015A4	PORTH	“Port Control Registers (Multiplexing and GPIO)” on page 9-30
0xFFC015A8	PORTH_SET	“Port Control Registers (Multiplexing and GPIO)” on page 9-30
0xFFC015AC	PORTH_CLEAR	“Port Control Registers (Multiplexing and GPIO)” on page 9-30
0xFFC015B0	PORTH_DIR_SET	“Port Control Registers (Multiplexing and GPIO)” on page 9-30
0xFFC015B4	PORTH_DIR_CLEAR	“Port Control Registers (Multiplexing and GPIO)” on page 9-30
0xFFC015B8	PORTH_INEN	“Port Control Registers (Multiplexing and GPIO)” on page 9-30
0xFFC015BC	PORTH_MUX	“Port Control Registers (Multiplexing and GPIO)” on page 9-30
0xFFC015C0	PORTI_FER	“Port Control Registers (Multiplexing and GPIO)” on page 9-30
0xFFC015C4	PORTI	“Port Control Registers (Multiplexing and GPIO)” on page 9-30
0xFFC015C8	PORTI_SET	“Port Control Registers (Multiplexing and GPIO)” on page 9-30
0xFFC015CC	PORTI_CLEAR	“Port Control Registers (Multiplexing and GPIO)” on page 9-30

Ports Registers

Table A-11. Port Control Registers (Multiplexing and GPIO) (Cont'd)

Memory-mapped Address	Register Name	More Information begins ...
0xFFC015D0	PORTI_DIR_SET	“Port Control Registers (Multiplexing and GPIO)” on page 9-30
0xFFC015D4	PORTI_DIR_CLEAR	“Port Control Registers (Multiplexing and GPIO)” on page 9-30
0xFFC015D8	PORTI_INEN	“Port Control Registers (Multiplexing and GPIO)” on page 9-30
0xFFC015DC	PORTI_MUX	“Port Control Registers (Multiplexing and GPIO)” on page 9-30
0xFFC015E0	PORTJ_FER	“Port Control Registers (Multiplexing and GPIO)” on page 9-30
0xFFC015E4	PORTJ	“Port Control Registers (Multiplexing and GPIO)” on page 9-30
0xFFC015E8	PORTJ_SET	“Port Control Registers (Multiplexing and GPIO)” on page 9-30
0xFFC015EC	PORTJ_CLEAR	“Port Control Registers (Multiplexing and GPIO)” on page 9-30
0xFFC015F0	PORTJ_DIR_SET	“Port Control Registers (Multiplexing and GPIO)” on page 9-30
0xFFC015F4	PORTJ_DIR_CLEAR	“Port Control Registers (Multiplexing and GPIO)” on page 9-30
0xFFC015F8	PORTJ_INEN	“Port Control Registers (Multiplexing and GPIO)” on page 9-30
0xFFC015FC	PORTJ_MUX	“Port Control Registers (Multiplexing and GPIO)” on page 9-30

Table A-12 lists the registers for pin interrupt programming.

Table A-12. Pin Interrupt Registers

Memory-mapped Address	Register Name	More Information begins...
0xFFC01400	PINT0_MASK_SET	"Pin Interrupt Registers" on page 9-33
0xFFC01404	PINT0_MASK_CLEAR	"Pin Interrupt Registers" on page 9-33
0xFFC01408	PINT0_REQUEST	"Pin Interrupt Registers" on page 9-33
0xFFC0140C	PINT0_ASSIGN	"Pin Interrupt Registers" on page 9-33
0xFFC01410	PINT0_EDGE_SET	"Pin Interrupt Registers" on page 9-33
0xFFC01414	PINT0_EDGE_CLEAR	"Pin Interrupt Registers" on page 9-33
0xFFC01418	PINT0_INVERT_SET	"Pin Interrupt Registers" on page 9-33
0xFFC0141C	PINT0_INVERT_CLEAR	"Pin Interrupt Registers" on page 9-33
0xFFC01420	PINT0_PINSTATE	"Pin Interrupt Registers" on page 9-33
0xFFC01424	PINT0_LATCH	"Pin Interrupt Registers" on page 9-33
0xFFC01430	PINT1_MASK_SET	"Pin Interrupt Registers" on page 9-33
0xFFC01434	PINT1_MASK_CLEAR	"Pin Interrupt Registers" on page 9-33
0xFFC01438	PINT1_REQUEST	"Pin Interrupt Registers" on page 9-33
0xFFC0143C	PINT1_ASSIGN	"Pin Interrupt Registers" on page 9-33

Ports Registers

Table A-12. Pin Interrupt Registers (Cont'd)

Memory-mapped Address	Register Name	More Information begins...
0xFFC01440	PINT1_EDGE_SET	“Pin Interrupt Registers” on page 9-33
0xFFC01444	PINT1_EDGE_CLEAR	“Pin Interrupt Registers” on page 9-33
0xFFC01448	PINT1_INVERT_SET	“Pin Interrupt Registers” on page 9-33
0xFFC0144C	PINT1_INVERT_CLEAR	“Pin Interrupt Registers” on page 9-33
0xFFC01450	PINT1_PINSTATE	“Pin Interrupt Registers” on page 9-33
0xFFC01454	PINT1_LATCH	“Pin Interrupt Registers” on page 9-33
0xFFC01460	PINT2_MASK_SET	“Pin Interrupt Registers” on page 9-33
0xFFC01464	PINT2_MASK_CLEAR	“Pin Interrupt Registers” on page 9-33
0xFFC01468	PINT2_REQUEST	“Pin Interrupt Registers” on page 9-33
0xFFC0146C	PINT2_ASSIGN	“Pin Interrupt Registers” on page 9-33
0xFFC01470	PINT2_EDGE_SET	“Pin Interrupt Registers” on page 9-33
0xFFC01474	PINT2_EDGE_CLEAR	“Pin Interrupt Registers” on page 9-33
0xFFC01478	PINT2_INVERT_SET	“Pin Interrupt Registers” on page 9-33
0xFFC0147C	PINT2_INVERT_CLEAR	“Pin Interrupt Registers” on page 9-33
0xFFC01480	PINT2_PINSTATE	“Pin Interrupt Registers” on page 9-33

Table A-12. Pin Interrupt Registers (Cont'd)

Memory-mapped Address	Register Name	More Information begins...
0xFFC01484	PINT2_LATCH	“Pin Interrupt Registers” on page 9-33
0xFFC01490	PINT3_MASK_SET	“Pin Interrupt Registers” on page 9-33
0xFFC01494	PINT3_MASK_CLEAR	“Pin Interrupt Registers” on page 9-33
0xFFC01498	PINT3_REQUEST	“Pin Interrupt Registers” on page 9-33
0xFFC0149C	PINT3_ASSIGN	“Pin Interrupt Registers” on page 9-33
0xFFC014A0	PINT3_EDGE_SET	“Pin Interrupt Registers” on page 9-33
0xFFC014A4	PINT3_EDGE_CLEAR	“Pin Interrupt Registers” on page 9-33
0xFFC014A8	PINT3_INVERT_SET	“Pin Interrupt Registers” on page 9-33
0xFFC014AC	PINT3_INVERT_CLEAR	“Pin Interrupt Registers” on page 9-33
0xFFC014B0	PINT3_PINSTATE	“Pin Interrupt Registers” on page 9-33
0xFFC014B4	PINT3_LATCH	“Pin Interrupt Registers” on page 9-33

Timer Registers

Timer registers (0xFFC0 0600 – 0xFFC0 06FF) are listed in [Table A-13](#).

Table A-13. Timer Registers

Memory-Mapped Address	Register Name	See Page
0xFFC0 1600	TIMER0_CONFIG	“Timer Configuration (TIMERx_CONFIG) Registers” on page 10-48
0xFFC0 1604	TIMER0_COUNTER	“Timer Counter (TIMERx_COUNTER) Registers” on page 10-51
0xFFC0 1608	TIMER0_PERIOD	“Timer Period (TIMERx_PERIOD) Registers” on page 10-54
0xFFC0 160C	TIMER0_WIDTH	“Timer Width (TIMERx_WIDTH) Registers” on page 10-55
0xFFC0 1610	TIMER1_CONFIG	“Timer Configuration (TIMERx_CONFIG) Registers” on page 10-48
0xFFC0 1614	TIMER1_COUNTER	“Timer Counter (TIMERx_COUNTER) Registers” on page 10-51
0xFFC0 1618	TIMER1_PERIOD	“Timer Period (TIMERx_PERIOD) Registers” on page 10-54
0xFFC0 161C	TIMER1_WIDTH	“Timer Width (TIMERx_WIDTH) Registers” on page 10-55
0xFFC0 1620	TIMER2_CONFIG	“Timer Configuration (TIMERx_CONFIG) Registers” on page 10-48
0xFFC0 1624	TIMER2_COUNTER	“Timer Counter (TIMERx_COUNTER) Registers” on page 10-51
0xFFC0 1628	TIMER2_PERIOD	“Timer Period (TIMERx_PERIOD) Registers” on page 10-54
0xFFC0 162C	TIMER2_WIDTH	“Timer Width (TIMERx_WIDTH) Registers” on page 10-55
0xFFC0 1630	TIMER3_CONFIG	“Timer Configuration (TIMERx_CONFIG) Registers” on page 10-48

Table A-13. Timer Registers (Cont'd)

Memory-Mapped Address	Register Name	See Page
0xFFC0 1634	TIMER3_COUNTER	“Timer Counter (TIMERx_COUNTER) Registers” on page 10-51
0xFFC0 1638	TIMER3_PERIOD	“Timer Period (TIMERx_PERIOD) Registers” on page 10-54
0xFFC0 163C	TIMER3_WIDTH	“Timer Width (TIMERx_WIDTH) Registers” on page 10-55
0xFFC0 1640	TIMER4_CONFIG	“Timer Configuration (TIMERx_CONFIG) Registers” on page 10-48
0xFFC0 1644	TIMER4_COUNTER	“Timer Counter (TIMERx_COUNTER) Registers” on page 10-51
0xFFC0 1648	TIMER4_PERIOD	“Timer Period (TIMERx_PERIOD) Registers” on page 10-54
0xFFC0 164C	TIMER4_WIDTH	“Timer Width (TIMERx_WIDTH) Registers” on page 10-55
0xFFC0 1650	TIMER5_CONFIG	“Timer Configuration (TIMERx_CONFIG) Registers” on page 10-48
0xFFC0 1654	TIMER5_COUNTER	“Timer Counter (TIMERx_COUNTER) Registers” on page 10-51
0xFFC0 1658	TIMER5_PERIOD	“Timer Period (TIMERx_PERIOD) Registers” on page 10-54
0xFFC0 165C	TIMER5_WIDTH	“Timer Width (TIMERx_WIDTH) Registers” on page 10-55
0xFFC0 1660	TIMER6_CONFIG	“Timer Configuration (TIMERx_CONFIG) Registers” on page 10-48
0xFFC0 1664	TIMER6_COUNTER	“Timer Counter (TIMERx_COUNTER) Registers” on page 10-51
0xFFC0 1668	TIMER6_PERIOD	“Timer Period (TIMERx_PERIOD) Registers” on page 10-54
0xFFC0 166C	TIMER6_WIDTH	“Timer Width (TIMERx_WIDTH) Registers” on page 10-55

CAN Registers

Table A-13. Timer Registers (Cont'd)

Memory-Mapped Address	Register Name	See Page
0xFFC0 1670	TIMER7_CONFIG	“Timer Configuration (TIMERx_CONFIG) Registers” on page 10-48
0xFFC0 1674	TIMER7_COUNTER	“Timer Counter (TIMERx_COUNTER) Registers” on page 10-51
0xFFC0 1678	TIMER7_PERIOD	“Timer Period (TIMERx_PERIOD) Registers” on page 10-54
0xFFC0 167C	TIMER7_WIDTH	“Timer Width (TIMERx_WIDTH) Registers” on page 10-55
0xFFC0 1680	TIMER_ENABLE	“Timer Enable 0 (TIMER_ENABLE0) Register” on page 10-40
0xFFC0 1684	TIMER_DISABLE	“Timer Disable 0 (TIMER_DISABLE0) Register” on page 10-42
0xFFC0 1688	TIMER_STATUS	“Timer Status 0 (TIMER_STATUS0) Register” on page 10-45

CAN Registers

CAN registers (0xFFC0 2A00 – 0xFFC0 2FFF) are listed in the “System MMR Assignments” appendix in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 2 of 2)*.

Handshake MDMA Control Registers

Handshake MDMA registers (0xFFC0 3300 – 0xFFC0 33FF) are listed in [Table A-14](#).

Table A-14. HMDMA Registers

Memory-Mapped Address	Register Name	See Page
0xFFC0 4500	HMDMA0_CONTROL	“Handshake MDMA Control (HMDMAx_CONTROL) Registers” on page 5-119
0xFFC0 4504	HMDMA0_ECINIT	“Handshake MDMA Initial Edge Count (HMDMAx_ECINIT) Registers” on page 5-123
0xFFC0 4508	HMDMA0_BCINIT	“Handshake MDMA Initial Block Count (HMDMAx_BCINIT) Registers” on page 5-120
0xFFC0 450C	HMDMA0_ECURGENT	“Handshake MDMA Edge Count Urgent (HMDMAx_ECURGENT) Registers” on page 5-124
0xFFC0 4510	HMDMA0_ECOVERFLOW	“Handshake MDMA Edge Count Overflow Interrupt (HMDMAx_ECOVERFLOW) Registers” on page 5-125
0xFFC0 4514	HMDMA0_ECOUNT	“Handshake MDMA Current Edge Count (HMDMAx_ECOUNT) Registers” on page 5-122
0xFFC0 4518	HMDMA0_BCOUNT	“Handshake MDMA Current Block Count (HMDMAx_BCOUNT) Registers” on page 5-121
0xFFC0 4540	HMDMA1_CONTROL	“Handshake MDMA Control (HMDMAx_CONTROL) Registers” on page 5-119
0xFFC0 4544	HMDMA1_ECINIT	“Handshake MDMA Initial Edge Count (HMDMAx_ECINIT) Registers” on page 5-123

NAND Flash Controller Registers

Table A-14. HMDMA Registers (Cont'd)

Memory-Mapped Address	Register Name	See Page
0xFFC0 4548	HMDMA1_BCINIT	“Handshake MDMA Initial Block Count (HMDMAx_BCINIT) Registers” on page 5-120
0xFFC0 454C	HMDMA1_ECURGENT	“Handshake MDMA Edge Count Urgent (HMDMAx_ECURGENT) Registers” on page 5-124
0xFFC0 4550	HMDMA1_ECOVERFLOW	“Handshake MDMA Edge Count Overflow Interrupt (HMDMAx_ECOVERFLOW) Registers” on page 5-125
0xFFC0 4554	HMDMA1_ECOUNT	“Handshake MDMA Current Edge Count (HMDMAx_ECOUNT) Registers” on page 5-122
0xFFC0 4558	HMDMA1_BCOUNT	“Handshake MDMA Current Block Count (HMDMAx_BCOUNT) Registers” on page 5-121

NAND Flash Controller Registers

NRC registers are listed in the “System MMR Assignments” appendix in the *ADSP-BF54x Blackfin Processor Hardware Reference (Volume 2 of 2)*.

Core Timer Registers

Core timer registers (0xFFE0 3000 – 0xFFE0 300C) are listed in [Table A-15](#).

Table A-15. Core Timer Registers

Memory-Mapped Address	Register Name	See Page
0xFFE0 3000	TCNTL	“Core Timer Control (TCNTL) Register” on page 11-5
0xFFE0 3004	TPERIOD	“Core Timer Period (TPERIOD) Register” on page 11-7
0xFFE0 3008	TSCALE	“Core Timer Scale (TSCALE) Register” on page 11-8
0xFFE0 300C	TCOUNT	“Core Timer Count (TCOUNT) Register” on page 11-6

Rotary Counter Registers

The rotary encoder interface has eight memory-mapped registers (MMRs) that regulate its operation. Refer to [Table A-16](#) for an overview of all MMRs associated with the rotary encoder interface.

Table A-16. Counter Module Registers

Memory-mapped Address	Register Name
0xFFC04200	CNT_CONFIG (on page 13-24)
0xFFC04204	CNT_IMASK (on page 13-24)
0xFFC04208	CNT_STATUS (on page 13-24)
0xFFC0420C	CNT_COMMAND (on page 13-24)

Security Registers

Table A-16. Counter Module Registers (Cont'd)

Memory-mapped Address	Register Name
0xFFC04210	CNT_DEBOUNCE (on page 13-24)
0xFFC04214	CNT_COUNTER (on page 13-24)
0xFFC04218	CNT_MAX (on page 13-24)
0xFFC0421C	CNT_MIN (on page 13-33)

Security Registers

There are three registers which provide information that can be used during security mode control and to return status of the Secure State Machine states (See [Table A-17](#)). These registers require privileged access depending on the operating state of the processor.

Table A-17. Security Registers

Memory-Mapped Address	Register	Description
0xFFC04320	SECURE_SYSSWT	Secure System Switches
0xFFC04324	SECURE_CONTROL	Secure Control
0xFFC04328	SECURE_STATUS	Secure Status

Processor-Specific Memory Registers

Processor-specific memory registers (0xFFE0 0004 – 0xFFE0 0300) are listed in [Table A-18](#).

Table A-18. Processor-Specific Memory Registers

Memory-Mapped Address	Register Name	See Page
0xFFE0 0004	DMEM_CONTROL	“L1 Data Memory Control Register” on page 3-29
0xFFE0 0300	DTEST_COMMAND	“Data Test Command Register” on page 3-46

Processor-Specific Memory Registers

I INDEX

Numerics

BCODE, [15-60](#), [15-61](#), [15-64](#), [15-67](#),
[17-110](#)
16-bit flash interface, [19-8](#)
16-bit SRAM, interface, [19-7](#), [19-8](#)
2D DMA, [5-19](#)
BMODE, [15-60](#), [15-61](#), [15-64](#), [15-67](#),
[17-110](#)
8/16 bit mode (DATA_SIZE), [8-25](#)
8-bit flash interface, [19-8](#)
8-bit SRAM interface, [19-8](#)

A

A11 (A11 element/coefficient) bits, [7-43](#),
[17-111](#) to [17-132](#)
A11 element/coefficients (A11) bits, [7-43](#)
A12 (A12 element/coefficient) bits, [7-43](#),
[17-27](#), [17-111](#), [17-112](#), [17-113](#),
[17-114](#), [17-116](#), [17-117](#), [17-118](#),
[17-119](#), [17-120](#), [17-121](#), [17-122](#),
[17-123](#), [17-125](#), [17-131](#), [17-132](#)
A13 (A13 element/coefficient) bits, [7-43](#),
[17-27](#), [17-111](#), [17-112](#), [17-113](#),
[17-114](#), [17-116](#), [17-117](#), [17-118](#),
[17-119](#), [17-120](#), [17-121](#), [17-122](#),
[17-123](#), [17-125](#), [17-131](#), [17-132](#)
A14 (A14 bias vector) bits, [7-46](#)
A14 bias vector (A14) bits, [7-46](#)
A21 (A21 element/coefficient) bits, [7-44](#)
A21 element/coefficients (A21) bits, [7-44](#)
A22 (A22 element/coefficient) bits, [7-44](#)

A22 element/coefficients (A22) bits, [7-44](#)
A23 (A23 element/coefficient) bits, [7-44](#)
A23 element/coefficients (A23) bits, [7-44](#)
A24 (A24 bias vector) bits, [7-46](#)
A24 bias vector (A24) bits, [7-46](#)
A31 (A31 element/coefficient) bits, [7-45](#)
A31 element/coefficients (A31) bits, [7-45](#)
A32 (A32 element/coefficient) bits, [7-45](#)
A32 element/coefficients (A32) bits, [7-45](#)
A33 (A33 element/coefficient) bits, [7-45](#)
A33 element/coefficients (A33) bits, [7-45](#)
A34 (A34 bias vector) bits, [7-46](#)
A34 bias vector (A34) bits, [7-46](#)
aborts, DMA, [5-38](#)

Access

request priority, *See also* Arbitration
DMA stalls to L1 or L2 memory, [2-14](#)
latency and throughput for L2 memory,
[2-14](#)
priority, L2 port, [2-13](#)
active descriptor queue, and DMA
synchronization, [5-71](#)
active mode, [1-32](#), [18-9](#)
ACTIVE_PLLDISABLED bit, [18-29](#)
ACTIVE_PLENABLED bit, [18-29](#)
address-tag compare operation, [3-18](#)
A_HEND (overlay A horizontal end) bits,
[7-40](#)
A_HSTART (overlay A horizontal start)
bits, [7-40](#)
alarm clock, RTC, [14-2](#)
alarm interrupt enable bit, [14-21](#)

Index

- alignment exceptions, [3-81](#)
- AMC, [1-14](#)
- AMS, [6-53](#)
- application data, loading, [17-1](#)
- Arbitration
 - DMA bus, [2-17](#)
 - priority for Sys L2 port access request, [2-13](#)
- Arbitration priority for Sys L2 port access request, [2-13](#)
- asynchronous controller, [1-14](#)
- asynchronous FIFO connection, [5-48](#)
- asynchronous memory, [6-2](#)
- Asynchronous Memory Bank Address
 - Range (table), [6-53](#)
- asynchronous memory bank address range (table), [6-6](#), [6-13](#), [6-15](#)
- ASYNCR memory banks, [6-4](#)
- atomic operations, [3-82](#)
- A_TRANSP (overlay A transparency) bits, [7-42](#)
- autobaud, and general-purpose timers, [10-35](#)
- autobaud detection, [10-35](#)
- autobuffer mode, [5-19](#), [5-37](#), [5-84](#)
- A_VEND (overlay A vertical end) bits, [7-41](#)
- A_VSTART (overlay A vertical start) bits, [7-41](#)

- B**
- bank activate command, [G-1](#)
- BCINIT[15:0] field, [5-120](#)
- BCOUNT[15:0] field, [5-121](#)
- BDI bit, [5-119](#)
- BDIE bit, [5-51](#), [5-119](#)
- BDR (burst DMA requests) bit, [8-25](#), [8-27](#)
- BFLAG_FASTREAD, [17-132](#)
- BFLAG_NEXTDXE, [17-132](#)
- BFLAG_NOAUTO, [17-132](#)
- BFLAG_NONRESTORE, [17-132](#)
- BFLAG_PERIPHERAL, [17-132](#)
- BFLAG_RESET, [17-132](#)
- BFLAG_RETURN, [17-132](#)
- BFLAG_SLAVE, [17-132](#)
- BFLAG_TYPE, [17-132](#)
- BFLAG_WAKEUP, [17-132](#)
- B_HEND (overlay B horizontal end) bits, [7-40](#)
- B_HSTART (overlay B horizontal start) bits, [7-40](#)
- bit 15 overflow interrupt enable (COV15IE) bit, [13-28](#)
- bit 15 overflow interrupt identifier (COV15II) bit, [13-29](#)
- bit 31 overflow interrupt enable (COV31IE) bit, [13-28](#)
- bit 31 overflow interrupt identifier (COV31II) bit, [13-29](#)
- BK_ID, [17-111](#), [17-112](#), [17-113](#), [17-114](#)
- BK_PROJECT, [17-111](#), [17-112](#), [17-113](#), [17-114](#)
- BK_UPDATE, [17-111](#), [17-112](#), [17-113](#), [17-114](#)
- BK_VERSION, [17-111](#), [17-112](#), [17-113](#), [17-114](#)
- Blackfin processor family
 - I/O memory space, [1-9](#)
 - memory architecture, [1-5](#)
- block, DMA, [5-17](#)
- block code field, [17-27](#)
- block count, DMA, [5-47](#)
- Block diagram, core, [2-6](#)

- block diagrams
 - core timer, 11-2
 - DMA controller, 5-6, 5-7
 - EBIU, 6-5
 - general-purpose timers, 10-4
 - interrupt processing, 4-25
 - PLL, 18-3
 - processor, 1-4
 - RTC, 14-4
 - watchdog timer, 12-3
- block done interrupt, DMA, 5-51
- Block Flags, 17-29
- block transfers, DMA, 5-47
- BMODE00_DIS, 17-122
- BMODE01_DIS, 17-122
- BMODE02_DIS, 17-122
- BMODE03_DIS, 17-122
- BMODE04_DIS, 17-122
- BMODE05_DIS, 17-122
- BMODE06_DIS, 17-122
- BMODE07_DIS, 17-122
- BMODE08_DIS, 17-122
- BMODE09_DIS, 17-122
- BMODE10_DIS, 17-122
- BMODE11_DIS, 17-122
- BMODE12_DIS, 17-122
- BMODE13_DIS, 17-122
- BMODE14_DIS, 17-122
- BMODE15_DIS, 17-122
- BMODE[2:0] pins, 17-6
- BMODE pins, 17-2
- BNDMODE (boundary register mode)
 - bits, 13-27
- boot
 - call boot kernel at run time, 17-50
 - load function, 17-49
 - manager, 17-54
 - quick, 17-44
 - ROM functions, 17-55
 - streams
 - multi-DXE, 17-55
- boot host wait
 - HWAIT, 17-33
- booting, 17-1 to ??
 - BFROM_MEMBOOT, 17-55
 - BFROM_SPIBOOT, 17-55
 - BFROM_TWIBOOT, 17-55
 - boot stream, 17-22
 - host boot scenarios, 17-23
 - host DMA boot modes, 17-90
 - indirect, 17-45
 - initialization code execution/boot, 17-42
 - memory locations, 17-24
 - NAND flash boot mode, 17-93
 - SPI slave mode, 17-76
 - TWI master mode, 17-79
 - TWI slave mode, 17-83
- boot kernel, 17-1
- Boot Management, 17-54
- boot mode
 - FIFO boot, 17-69
 - flash boot, 17-64
 - no-boot, 17-63
 - SDRAM boot, 17-69
 - SPI device detection, 17-72
 - UART slave, 17-84
- boot ROM
 - internal, 17-1
- boot stream, 17-1, 17-22
- boot termination, 17-35
- boundary register mode (BNDMODE)
 - bits, 13-27
- BT_EN (bus timeout enable) bit, 8-25
- B_TRANSP (overlay B transparency) bits, 7-42
- buffer registers, timers, 10-52
- buffers
 - Cacheability Protection Lookaside
 - Buffers (CPLBs), 3-54, 3-55
- burst DMA requests (BDR) bit, 8-25, 8-27

Index

- burst length, [G-2](#)
- burst type, [G-2](#)
- Bus connection and arbitration, DMA, [2-17](#)
- bus contention, avoiding, [19-9](#)
- bus error, EBIU, [6-9](#)
- buses
 - bandwidth, [1-3](#)
 - and DMA, [5-52](#)
 - and peripherals, [1-3](#)
 - prioritization and DMA, [5-59](#)
- bus standard, I²C, [1-15](#)
- bus timeout enable (BT_EN) bit, [8-25](#)
- B_VEND (overlay B vertical end) bits, [7-41](#)
- BV_MULT4 (multiply row by 4) bit, [7-45](#)
- B_VSTART (overlay B vertical start) bits, [7-41](#)
- BV_TRANS (transparent color - B/V) bits, [7-47](#)
- BxMAP (byte x mapping) bits, [9-56](#)
- BYPASS bit, [18-28](#)
- bypass capacitor placement, [19-17](#)
- byte x mapping (BxMAP) bits, [9-56](#)

- C**
- cache, [3-15](#)
 - coherency support, [3-81](#)
 - mapping into data banks, [3-35](#)
 - validity of cache lines, [3-17](#)
- Cacheability Protection Lookaside Buffers (CPLBs), [3-15](#), [3-54](#), [3-55](#)
- cache block (definition), [3-84](#)
- cache hit, [3-84](#)
 - address-tag compare, [3-18](#)
 - data cache access, [3-39](#)
 - definition, [3-18](#)
 - processing, [3-18](#)
- cache inhibited accesses, [3-82](#)
- cache line
 - components, [3-15](#)
 - definition, [3-84](#)
 - states, [3-40](#)
- cache miss, [3-84](#)
 - definition, [3-39](#)
 - replacement policy, [3-19](#)
- callback routines, [17-46](#)
- CAN
 - autobaud detection, [10-35](#)
 - bit rate detection, [10-6](#)
- CANWE bit, [18-30](#)
- capacitors, [19-15](#)
- capture mode, *See* WIDTH_CAP mode
- CCLK (core clock), [18-5](#)
 - disabling, [18-22](#)
 - status by operating mode, [18-8](#)
- CDGINV (CDG pin polarity invert) bit, [13-27](#)
- CDG pin polarity invert (CDGINV) bit, [13-27](#)
- circular addressing, [5-68](#)
- clean (definition), [3-85](#)
- clearing interrupt requests, [4-44](#)
- CLKBUFOE bit, [18-30](#)
- CLKIN, [1-31](#), [18-1](#)
- CLKIN (input clock), [18-3](#)
- CLKIN to VCO, changing the multiplier, [18-15](#)
- CLK_SEL bit, [10-14](#), [10-23](#), [10-48](#), [10-57](#)
- clock
 - clock signals, [1-31](#)
 - EBIU, [6-2](#)
 - managing, [19-2](#)
 - RTC, [14-5](#)
 - source for general-purpose timers, [10-6](#)
 - types, [19-2](#)
- clock input (CLKIN) pin, [19-2](#)
- clock rate
 - core timer, [11-1](#)

- clock ratio, changing, [18-6](#)
- clocks
 - internal, [2-5](#)
- CNFG_PEND (config pending) bit, [8-27](#)
- CNT_COMMAND (command) register, [13-25](#), [13-29](#), [A-31](#)
- CNT_CONFIG (configuration) register, [13-24](#), [13-27](#), [A-31](#)
- CNT_COUNTER (counter) register, [13-25](#), [13-32](#), [A-32](#)
- CNT_DEBOUNCE (debounce) register, [13-25](#), [13-31](#), [A-32](#)
- CNTE (counter enable) bit, [13-27](#)
- CNT_IMASK (interrupt mask) register, [13-24](#), [13-28](#), [A-31](#)
- CNT_MAX (maximal count) register, [13-25](#), [13-32](#), [A-32](#)
- CNT_MIN (minimal count) register, [13-25](#), [13-32](#), [A-32](#)
- CNTMODE (counter operating mode) bits, [13-27](#)
- CNT_STATUS (status) register, [13-24](#), [13-29](#), [A-31](#)
- code examples
 - CSYNC, [3-84](#)
 - interrupt enabling and disabling, [3-83](#)
 - load base of MMRs, [3-83](#)
 - restoration of the control register, [3-84](#)
- column address
 - strobe latency, [G-4](#)
- command (CNT_COMMAND) register, [13-25](#), [13-29](#), [A-31](#)
- commands
 - bank activate, [G-1](#)
 - DMA control, [5-40](#), [5-41](#)
 - precharge, [G-18](#)
- COMPLETE (DMA complete) bit, [8-27](#)
- conditional
 - branches, [3-79](#)
- config pending (CNFG_PEND) bit, [8-27](#)
- configuration
 - L1 Instruction Memory, [3-15](#)
 - L1 SRAM, [3-2](#)
 - precautions before changing, [3-13](#)
- configuration (CNT_CONFIG) register, [13-24](#), [13-27](#), [A-31](#)
- congestion, on DMA channels, [5-56](#)
- Connection and arbitration, DMA bus, [2-17](#)
- Content-Addressable Memory (CAM), [3-54](#)
- continuous transition, DMA, [5-36](#)
- control bit summary, general-purpose timers, [10-56](#)
- Controller, DMA, [2-4](#), [2-18](#)
- Control/optimization, DMA traffic, [2-14](#)
- control register
 - data memory, [3-28](#)
 - instruction memory, [3-9](#)
 - restoration, [3-84](#)
- conventions, [-1](#)
- core
 - core clock (CCLK), [18-5](#), [19-2](#)
 - core clock/system clock ratio control, [18-5](#)
 - powering down, [18-22](#)
 - waking from idle state, [4-13](#)
- core and system reset, code example, [8-31](#), [17-154](#)
- Core block diagram, [2-6](#)
- core clock, *See* CCLK
- core event controller (CEC), [4-2](#), [4-6](#)
- core-only software reset, [17-5](#)

Index

- core timer, [-xlvi](#) to [??](#), [11-1](#) to [11-9](#)
 - block diagram, [11-2](#)
 - clock rate, [11-1](#)
 - features, [11-1](#)
 - initialization, [11-3](#)
 - internal interfaces, [11-2](#)
 - interrupts, [11-3](#)
 - low power mode, [11-3](#)
 - operation, [11-3](#)
 - registers, [11-4](#)
 - scaling, [11-7](#)
- core timer control register (TCNTL), [11-3](#), [11-5](#)
- core timer count register (TCOUNT), [11-6](#)
- core timer period register (TPERIOD), [11-7](#)
- core timer scale register (TSCALE), [11-7](#), [11-8](#)
- counter, RTC, [14-2](#)
- counter (CNT_COUNTER) register, [13-25](#), [13-32](#), [A-32](#)
- counter enable (CNTE) bit, [13-27](#)
- counter operating mode (CNTMODE) bits, [13-27](#)
- COUNT_TIMEOUT (host timeout count) bits, [8-29](#)
- count to zero interrupt enable (CZEROIE) bit, [13-28](#)
- count to zero interrupt identifier (CZEROII) bit, [13-29](#)
- count value[15:0] field, [11-6](#)
- count value[31:16] field, [11-6](#)
- COV15IE (bit 15 overflow interrupt enable) bit, [13-28](#)
- COV15II (bit 15 overflow interrupt identifier) bit, [13-29](#)
- COV31IE (bit 31 overflow interrupt enable) bit, [13-28](#)
- COV31II (bit 31 overflow interrupt identifier) bit, [13-29](#)
- CRC32 checksum generation, [17-49](#)
- CrossCore software, [1-36](#)
- crosstalk, [19-15](#)
- CSEL[1:0] field, [18-5](#), [18-27](#)
- CSYNC, [3-79](#)
 - code example, [3-84](#)
- CTYPE bit, [5-79](#)
- CUD and CDZ input disable (INPDIS) bit, [13-27](#)
- CUDINV (CUD pin polarity invert) bit, [13-27](#)
- CUD pin polarity invert (CUDINV) bit, [13-27](#)
- current address field, [5-94](#)
- current address registers
 - (DMAx_CURR_ADDR), [5-93](#)
 - (MDMA_yy_CURR_ADDR), [5-93](#)
- current descriptor pointer field, [5-114](#)
- current descriptor pointer registers
 - (DMAx_CURR_DESC_PTR), [5-113](#)
 - (MDMA_yy_CURR_DESC_PTR), [5-113](#)
- current inner loop count registers
 - (DMAx_CURR_X_COUNT), [5-98](#), [5-99](#)
 - (MDMA_yy_CURR_X_COUNT), [5-98](#), [5-99](#)
- current outer loop count registers
 - (DMAx_CURR_Y_COUNT), [5-105](#)
 - (MDMA_yy_CURR_Y_COUNT), [5-105](#)
- CURR_X_COUNT[15:0] field, [5-99](#)
- CURR_Y_COUNT[15:0] field, [5-106](#)
- customer support, [-xlviii](#)
- CZEROIE (count to zero interrupt enable) bit, [13-28](#)
- CZEROII (count to zero interrupt identifier) bit, [13-29](#)

- CZMEIE (CZM error interrupt enable) bit, [13-28](#)
 - CZMEII (CZM error interrupt identifier) bit, [13-29](#)
 - CZM error interrupt enable (CZMEIE) bit, [13-28](#)
 - CZM error interrupt identifier (CZMEII) bit, [13-29](#)
 - CZMIE (CZM pin interrupt enable) bit, [13-28](#)
 - CZMII (CZM pin interrupt identifier) bit, [13-29](#)
 - CZMINV (CZM pin polarity invert) bit, [13-27](#)
 - CZM pin interrupt enable (CZMIE) bit, [13-28](#)
 - CZM pin interrupt identifier (CZMII) bit, [13-29](#)
 - CZM pin polarity invert (CZMINV) bit, [13-27](#)
 - CZM zeroes counter enable (ZMZC) bit, [13-27](#)
 - CZM zeroes counter interrupt enable (CZMZIE) bit, [13-28](#)
 - CZM zeroes counter interrupt identifier (CZMZII) bit, [13-29](#)
 - CZMZIE (CZM zeroes counter interrupt enable) bit, [13-28](#)
 - CZMZII (CZM zeroes counter interrupt identifier) bit, [13-29](#)
- D**
- DAB, [5-52](#), [5-127](#)
 - clocking, [18-2](#)
 - DAB, DMA Access Bus, [2-5](#), [2-17](#)
 - DAB_TRAFFIC_COUNT[2:0] field, [5-127](#)
 - data cache control instructions, [3-43](#)
 - data-driven interrupts, [5-90](#)
 - data interrupt timing select (DI_SEL) bit, [8-6](#)
 - data memory, L1, [3-28](#)
 - Data Memory Control register (DMEM_CONTROL), [3-28](#), [3-54](#)
 - data operations, CPLB, [3-55](#)
 - DATA_SIZE bit, [8-25](#)
 - Data SRAM
 - L1, [3-31](#)
 - data store format, [3-85](#)
 - data structures, [17-126](#)
 - boot_struct, [17-127](#)
 - buffer_struct, [17-127](#)
 - header_struct, [17-126](#)
 - Data Test Command register (DTEST_COMMAND), [3-45](#)
 - Data Test Data registers (DTEST_DATAx), [3-47](#)
 - Data transfer latency
 - DMA, [2-14](#)
 - day[14:0] field, [14-23](#)
 - day alarm interrupt enable bit, [14-21](#)
 - day counter[14:0] field, [14-21](#)
 - DCB, [5-52](#), [5-128](#)
 - DCB1, [2-11](#)
 - DCB2, [2-10](#), [2-11](#)
 - DCB, DMA Core Bus, [2-5](#), [2-17](#)
 - DCB bus arbitration, [2-18](#)
 - DCBS (L1 Data Cache Bank Select) bit, [3-37](#)
 - DCB_TRAFFIC_COUNT field, [5-128](#)
 - DCB_TRAFFIC_PERIOD field, [5-128](#)
 - DCIE (down count interrupt enable) bit, [13-28](#)
 - DCII (down count interrupt identifier) bit, [13-29](#)
 - DCPLB Address registers (DCPLB_ADDRx), [3-67](#)
 - DCPLB_ADDRx (DCPLB Address registers), [3-67](#)

Index

- DCPLB Data registers (DCPLB_DATAx), [3-65](#)
- DCPLB_DATAx (DCPLB Data registers), [3-65](#)
- DCPLB_FAULT_ADDR (DCPLB Fault Address register), [3-72](#)
- DCPLB Fault Address register (DCPLB_FAULT_ADDR), [3-72](#)
- DCPLB_STATUS (DCPLB Status register), [3-71](#)
- DCPLB Status register (DCPLB_STATUS), [3-71](#)
- DDR SDRAM controller, [1-13](#)
- DEB, [5-52](#), [5-128](#)
- DEB, DMA External Bus, [2-5](#), [2-17](#)
- DEBE (debounce enable) bit, [13-27](#)
- debounce (CNT_DEBOUNCE) register, [13-25](#), [13-31](#), [A-32](#)
- debounce enable (DEBE) bit, [13-27](#)
- DEB_TRAFFIC_COUNT field, [5-128](#)
- DEB_TRAFFIC_PERIOD field, [5-128](#)
- debugging, [1-37](#)
 - test point access, [19-19](#)
- deep sleep mode, [1-33](#), [18-10](#)
- default mapping, peripheral to DMA, [5-10](#)
- descriptor array mode, DMA, [5-23](#), [5-85](#)
- descriptor-based DMA, [5-21](#)
- descriptor chains, DMA, [5-36](#)
- descriptor list mode, DMA, [5-22](#), [5-85](#)
- descriptor queue, [5-69](#)
 - management, [5-68](#)
 - synchronization, [5-68](#), [5-69](#)
- descriptor structures
 - DMA, [5-67](#)
 - MDMA, [5-74](#)
- destination channels, memory DMA, [5-13](#)
- development tools, [1-36](#)
- DF bit, [18-4](#), [18-28](#)
- DFETCH bit, [5-22](#), [5-30](#), [5-88](#)
- DFRESET, [15-60](#), [15-61](#), [15-64](#), [15-67](#), [17-110](#)
- DI_EN bit, [5-22](#), [5-82](#), [5-85](#)
- direct code execution, [17-37](#)
 - initial header, [17-36](#), [17-38](#)
- direct mapped (definition), [3-84](#)
- direct memory access, *See* DMA
- dirty (definition), [3-84](#)
- Disable Interrupts (CLI) instruction, [3-84](#)
- disabling
 - general-purpose timers, [10-41](#)
 - PLL, [18-13](#)
- DI_SEL bit, [5-82](#), [5-85](#)
- DI_SEL (data interrupt), [8-6](#)
- DMA, [5-1](#) to [5-140](#)
 - 1D interrupt-driven, [5-65](#)
 - 1D unsynchronized FIFO, [5-66](#)
 - 2D, polled, [5-66](#)
 - 2D array, example, [5-130](#)
 - 2D interrupt-driven, [5-65](#)
 - autobuffer mode, [5-19](#), [5-37](#), [5-84](#)
 - bandwidth, [5-56](#)
 - block count, [5-47](#)
 - block diagram, [5-6](#), [5-7](#)
 - block done interrupt, [5-51](#)
 - block transfers, [5-17](#), [5-47](#)
 - channel registers, [5-78](#)
 - channels, [5-52](#)
 - channels and control schemes, [5-61](#)
 - channel-specific register names, [5-77](#)
 - congestion, [5-56](#)
 - connecting asynchronous FIFO, [5-48](#)
 - continuous transfers using autobuffering, [5-65](#)
 - continuous transition, [5-36](#)
 - control command restrictions, [5-44](#)
 - control commands, [5-40](#), [5-41](#)
 - data transfers, [5-2](#)
 - default peripheral mapping, [5-10](#)
 - descriptor array, [5-31](#)

- descriptor array mode, [5-23](#), [5-85](#)
- descriptor-based, [5-21](#)
- descriptor-based, initializing, [5-133](#)
- descriptor-based vs. register-based transfers, [5-3](#)
- descriptor chains, [5-36](#)
- descriptor element offsets, [5-24](#)
- descriptor list mode, [5-22](#), [5-85](#)
- descriptor lists, [5-31](#)
- descriptor queue, [5-68](#), [5-69](#)
- descriptors, recommended size, [5-24](#)
- descriptor structures, [5-67](#)
- direction, [5-86](#)
- DMA error interrupt, [5-91](#)
- double buffer scheme, [5-65](#)
- and EBIU, [5-8](#)
- errors, [5-38](#), [5-39](#)
- example connection, receive, [5-50](#)
- example connection, transmit, [5-49](#)
- external interfaces, [5-8](#)
- finish control command, [5-42](#)
- first data memory access, [5-30](#)
- flow chart, [5-27](#), [5-28](#)
- FLOW mode, [5-25](#)
- FLOW value, [5-29](#)
- functions, summary, [5-4](#)
- handshake operation, [5-46](#)
- header file to define descriptor structures
 - example, [5-134](#)
- HMDMA1 block enable example, [5-139](#)
- initializing, [5-25](#)
- internal interfaces, [5-8](#)
- large model mode, [5-85](#)
- latency, [5-33](#)
- mapping to peripherals, [4-14](#), [4-15](#)
- memory conflict, [5-60](#)
- memory DMA, [5-13](#)
- memory DMA streams, [5-14](#)
- memory DMA transfers, [5-9](#)
- memory read, [5-34](#)
- operation flow, [5-25](#)
- orphan access, [5-37](#)
- overflow interrupt, [5-51](#)
- overview, [1-10](#)
- performance considerations, [5-53](#)
- peripheral, [5-10](#)
- peripheral channels, [5-2](#)
- peripheral channels priority, [5-12](#)
- peripheral interrupts, [4-13](#)
- pipelining requests, [5-48](#)
- polling DMA status example, [5-133](#)
- polling registers, [5-62](#)
- prioritization and traffic control, [5-55](#) to [5-61](#)
- programming examples, [5-129](#) to [5-140](#)
- receive, [5-36](#)
- receive restart or finish, [5-45](#)
- refresh, [5-31](#)
- register-based, [5-17](#)
- register-based 2D memory DMA
 - example, [5-130](#)
- register naming conventions, [5-78](#)
- remapping peripheral assignment, [5-11](#)
- request data control command, [5-43](#)
- request data urgent control command, [5-44](#)
- restart control command, [5-42](#)
- round robin operation, [5-58](#)
- single-buffer transfers, [5-64](#)
- small model mode, [5-85](#)
- software management, [5-61](#)
- software-triggered descriptor fetch
 - example, [5-136](#)
- startup, [5-25](#)
- stop mode, [5-18](#), [5-84](#)
- stopping transfers, [5-37](#)
- support for peripherals, [1-3](#)
- switching peripherals from, [5-91](#)
- synchronization, [5-61](#) to [5-72](#)
- synchronized transition, [5-36](#)

Index

- termination without abort, [5-37](#)
- throughput, [5-52](#)
- traffic control, [5-59](#)
- traffic exceeding available bandwidth, [5-56](#)
- transfers, urgent, [5-55](#)
- transmit, [5-34](#)
- transmit restart or finish, [5-45](#)
- triggering transfers, [5-72](#)
- two descriptors in small list flow mode, example, [5-134](#)
- two-dimensional, [5-19](#)
- two-dimensional memory DMA setup example, [5-131](#)
- using descriptor structures example, [5-135](#)
- variable descriptor size, [5-23](#)
- word size, changing, [5-36](#), [5-37](#)
- work units, [5-22](#), [5-31](#), [5-33](#)
- DMA, Related Buses, [2-17](#)
- DMA2D bit, [5-82](#), [5-86](#)
- DMA2D (DMA mode) bit, [8-6](#)
- DMA Access Bus (DAB), [2-5](#), [2-17](#)
- DMA access to L1 or L2 memory, stalls, [2-14](#)
- DMA Bus
 - connection and arbitration, [2-17](#)
- DMACFG field, [5-30](#), [5-73](#)
- DMA channel registers, [5-74](#)
- DMA Code field
 - DMACODE, [17-27](#)
- DMA complete (COMPLETE) bit, [8-27](#)
- DMA configuration registers
 - (DMAx_CONFIG), [5-82](#)
 - (MDMA_yy_CONFIG), [5-82](#)
- DMA controller, [2-4](#), [2-18](#), [5-2](#)
- DMA Core Bus (DCB), [2-5](#), [2-17](#)
- DMA data transfer latency, [2-14](#)
- DMA_DIR (DMA direction) bit, [8-27](#)
- DMA direction (DMA_DIR) bit, [8-27](#)
- DMA_DONE bit, [5-88](#)
- DMA_DONE interrupt, [5-87](#)
- DMAEN bit, [5-26](#), [5-72](#), [5-82](#), [5-86](#)
- DMA_ERR bit, [5-88](#)
- DMA_ERROR interrupt, [5-38](#)
- DMA error interrupts, [5-90](#)
- DMA External Bus (DEB), [2-5](#), [2-17](#)
- DMA performance optimization, [5-51](#)
- DMA queue completion interrupt, [5-71](#)
- DMAR0 pin, [5-8](#)
- DMAR1 pin, [5-8](#)
- DMA ready (READY) bit, [8-27](#)
- DMA registers, [5-74](#), [5-75](#)
- DMA_RUN bit, [5-30](#), [5-69](#), [5-73](#), [5-87](#), [5-88](#)
- DMARx pin, [5-48](#)
- DMA start address field, [5-92](#)
- DMA_TC_CNT (DMA traffic control counter register), [5-127](#)
- DMA_TC_PER (DMA traffic control counter period register), [5-58](#), [5-127](#)
- DMA traffic control counter period register (DMA_TC_PER), [5-127](#)
- DMA traffic control counter register (DMA_TC_CNT), [5-127](#)
- DMA traffic control/optimization, [2-14](#)
- DMA_TRAFFIC_PERIOD field, [5-127](#)
- DMAx_CONFIG (DMA configuration registers), [5-15](#), [5-26](#), [5-34](#), [5-82](#)
- DMAx_CURR_ADDR (current address registers), [5-93](#)
- DMAx_CURR_DESC_PTR (current descriptor pointer registers), [5-113](#)
- DMAx_CURR_X_COUNT (current inner loop count registers), [5-98](#), [5-99](#)
- DMAx_CURR_Y_COUNT (current outer loop count registers), [5-105](#)
- DMAx_IRQ_STATUS (interrupt status registers), [5-87](#), [5-88](#)

- DMAx_NEXT_DESC_PTR (next descriptor pointer registers), 5-25, 5-26, 5-110
- DMAx_PERIPHERAL_MAP (peripheral map registers), 5-79
- DMAx_START_ADDR (start address registers), 5-25, 5-91
- DMAx_X_COUNT (inner loop count registers), 5-96
- DMAx_X_MODIFY (inner loop address increment registers), 5-26, 5-101
- DMAx_Y_COUNT (outer loop count registers), 5-103
- DMAx_Y_MODIFY (outer loop address increment registers), 5-26, 5-108
- DMEM_CONTROL (Data Memory Control register), 3-28, 3-54
- DOUBLE_FAULT bit, 15-34, 17-108
- DOUBLE_RESET, 15-34, 17-108
- DPMC, 18-2, 18-7 to ??
- D Port, 2-4
 - interface, 2-10
- DRQ[1:0] field, 5-56, 5-117, 5-119
- DTEST_COMMAND (Data Test Command register), 3-45
- DTEST_DATAx (Data Test Data registers), 3-47
- dynamic power management, 1-32, 18-1 to ??
- dynamic power management controller, 18-2

- E**
- EAB
 - clocking, 18-2
- EAB, External Access Bus, 2-4, 2-5, 2-25

- EBIU, 1-13, 6-1 to 6-84
 - as slave, 6-8
 - block diagram, 6-5
 - bus errors, 6-9
 - clock, 6-2
 - and DMA, 5-8
 - error detection, 6-8
 - overview, 6-1
 - request priority, 6-2
- EBIU, External Bus Interface Unit, 2-2
- EBIU_AMGCTL (Asynchronous Memory Global Control register), 6-57
- EBIU Pin List (with Multiplexing), 6-6
- ECINIT[15:0] field, 5-123
- ECOUNT[15:0] field, 5-122
- EHR (enable host reads) bit, 8-25
- EHW (enable host writes) bit, 8-25
- elfloader.exe, 17-22
- emulation, and timer counter, 10-49
- EMU_RUN bit, 10-48, 10-57
- enable host reads (EHR) bit, 8-25
- enable host writes (EHW) bit, 8-25
- Enable Interrupts (STI) instruction, 3-84
- enabling
 - general-purpose timers, 10-40
 - interrupts, 4-11
- endian format
 - data and instruction storage, 3-74
- error handler routine, 17-48
- errors
 - DMA, 5-38
 - misalignment of data, 3-81
 - not detected by DMA hardware, 5-39
 - startup, and timers, 10-12
- ERR_TYP[1:0] field, 10-11, 10-47, 10-48, 10-57
- event handling, 4-6

Index

events

- default mapping, [4-16](#)
 - definition, [4-6](#)
 - types of, [4-6](#)
- event vector table (EVT), [4-2](#)
- EVT1 register, [17-7](#)
- exclusive (definition), [3-85](#)
- EXT_CLK mode, [10-36](#) to [10-37](#), [10-52](#)
- control bit and register usage, [10-56](#)
 - flow diagram, [10-37](#)
- External Access Bus (EAB), [2-4](#), [2-5](#), [2-25](#)
- External Bus (DEB), DMA, [2-5](#), [2-17](#)
- external bus interface unit, *See* EBIU
- External Bus Interface Unit (EBIU), [2-2](#)
- External Bus Interface Unit (EBIU)
- Diagram, [6-5](#)
- external crystal, [1-31](#)
- External memory, [2-2](#)
- external memory, [3-53](#)
- design issues, [19-5](#)
- external memory map
- figure, [6-4](#)
- EZ-KIT Lite, [1-39](#)

F

- FIFOEMPTY bit, [8-27](#)
- FIFO flush (HOST_FLUSH) bit, [8-25](#)
- FIFOFULL bit, [8-27](#)
- finish control command, DMA, [5-42](#)
- Flash, [2-2](#)
- flash interface, [19-8](#)
- flex descriptors, [5-3](#)
- FLOW[2:0] field, [5-31](#), [5-33](#), [5-67](#), [5-82](#),
[5-84](#)
- FLOW bit, [8-6](#)

flow charts

- DMA, [5-27](#), [5-28](#)
 - general-purpose timers interrupt structure, [10-10](#)
 - timer EXT_CLK mode, [10-37](#)
 - timer PWM_OUT mode, [10-15](#)
 - timer WDT_CAP mode, [10-28](#)
- FLOW mode, DMA, [5-25](#)
- FLOW value, DMA, [5-29](#)
- FLUSH instruction, [3-43](#)
- FLUSHINV instruction, [3-43](#)
- frame interrupt enable (FRM_INT_EN)
- bit, [7-42](#)
- frame interrupt status (FRM_INT_STAT)
- bit, [7-42](#)
- FREQ[1:0] field, [18-20](#), [18-30](#)
- FRM_INT_EN (frame interrupt enable)
- bit, [7-42](#)
- FRM_INT_STAT (frame interrupt status)
- bit, [7-42](#)
- FULL_ON bit, [18-29](#)
- full on mode, [1-32](#), [18-8](#)

G

- GAIN[1:0] field, [18-19](#), [18-30](#)
- gain levels, [18-19](#)
- general-purpose interrupts, [4-6](#), [4-7](#)
- general-purpose ports, [1-14](#), [9-1](#) to [9-68](#)
- general-purpose timers, [10-1](#) to [10-67](#)
- aborting, [10-26](#)
 - and startup errors, [10-12](#)
 - autobaud mode, [10-35](#)
 - block diagram, [10-4](#)
 - buffer registers, [10-52](#)
 - capture mode, [10-8](#)
 - clock source, [10-6](#)
 - code examples, [10-58](#)
 - control bit summary, [10-56](#)
 - counter, [10-7](#)
 - disable timing, [10-26](#)

disabling, [10-41](#)
 enabling, [10-7](#), [10-38](#), [10-40](#)
 error detection, [10-11](#)
 EXT_CLK mode, [10-52](#)
 external interface, [10-5](#)
 features, [10-2](#)
 flow diagram for EXT_CLK mode, [10-37](#)
 generating maximum frequency, [10-19](#)
 illegal states, [10-11](#), [10-12](#)
 internal interface, [10-6](#)
 internal timer structure, [10-5](#)
 interrupts, [10-7](#), [10-8](#), [10-18](#), [10-32](#)
 interrupt setup, [10-60](#)
 interrupt structure, [10-10](#)
 measurement report, [10-29](#), [10-30](#), [10-31](#)
 non-overlapping clock pulses, [10-64](#)
 output pad disable, [10-16](#)
 overflow, [10-7](#)
 periodic interrupt requests, [10-61](#)
 port setup, [10-58](#)
 preventing errors in PWM_OUT mode, [10-54](#)
 programming model, [10-38](#)
 PULSE_HI toggle mode, [10-19](#)
 PWM mode, [10-8](#)
 PWM_OUT mode, [10-14](#) to [10-26](#), [10-52](#)
 registers, [10-39](#)
 signal generation, [10-59](#)
 single pulse generation, [10-16](#)
 size of register accesses, [10-39](#)
 stopping in PWM_OUT mode, [10-24](#)
 three timers with same period, [10-20](#)
 two timers with non-overlapping clocks, [10-21](#)
 waveform generation, [10-17](#)
 WIDTH_CAP mode, [10-27](#), [10-52](#)

WIDTH_CAP mode configuration, [10-66](#)
 WIDTH_CAP mode flow diagram, [10-28](#)
 glueless connection, [19-7](#)
 GPIO, [9-1](#) to [9-68](#)
 interrupt request, [4-45](#)
 pins, [9-2](#)
 ground plane, [19-15](#)
 GU_MULT4 (multiply row by 4) bit, [7-44](#)
 GU_TRANS (transparent color - G/U) bits, [7-47](#)

H

handshake MDMA, [5-16](#), [5-46](#)
 interrupts, [5-50](#)
 handshake MDMA control registers (HMDMAx_CONTROL), [5-117](#), [5-119](#)
 handshake MDMA current block count registers (HMDMAx_BCOUNT), [5-120](#), [5-121](#)
 handshake MDMA current edge count registers (HMDMAx_ECOUNTER), [5-121](#), [5-122](#)
 handshake MDMA edge count overflow interrupt registers (HMDMAx_ECOVERFLOW), [5-125](#)
 handshake MDMA edge count overflow interrupt registers (HMDMAx_ECOVERFLOW), [5-125](#)
 handshake MDMA edge count urgent registers (HMDMAx_ECURGENT), [5-124](#)
 handshake MDMA initial block count registers (HMDMAx_BCINIT), [5-120](#)

Index

- handshake MDMA initial edge count registers
(HMDMAx_ECINIT), [5-123](#)
 - handshake MDMA initial edge count registers (HMDMAx_ECINIT), [5-123](#)
 - handshake memory DMA, [5-3](#)
 - hardware reset, [17-7](#)
 - Harvard architecture, [3-5](#)
 - header checksum field
HDRCHK, [17-31](#)
 - header signature
HDRSGN, [17-31](#)
 - hibernate state, [1-33](#), [18-10](#), [18-22](#)
 - high frequency design considerations, [19-5](#)
 - HIRQ (host interrupt request) bit, [8-27](#)
 - HMDMA, [5-16](#)
 - HMDMAEN bit, [5-46](#), [5-48](#), [5-119](#)
 - HMDMAx_BCINIT (handshake MDMA configuration registers), [5-47](#), [5-120](#)
 - HMDMAx_BCOUNT (handshake MDMA current block count registers), [5-47](#), [5-120](#), [5-121](#)
 - HMDMAx_CONTROL (handshake MDMA control registers), [5-8](#), [5-117](#), [5-119](#)
 - HMDMAx_ECINIT (handshake MDMA initial edge count registers), [5-48](#), [5-123](#)
 - HMDMAx_ECOUNT (handshake MDMA current edge count registers), [5-48](#), [5-121](#), [5-122](#)
 - HMDMAx_ECOVERFLOW (handshake MDMA edge count overflow interrupt registers), [5-125](#)
 - HMDMAx_ECURGENT (handshake MDMA edge count urgent registers), [5-124](#)
 - HOST acknowledge mode timeout (HOST_TIMEOUT) register, [8-29](#)
 - HOST_CONFIG (HOST configuration) word, [8-6](#)
 - HOST configuration word (HOST_CONFIG), [8-6](#)
 - HOST_CONTROL (HOST control) register, [8-25](#)
 - HOST control (HOST_CONTROL) register, [8-25](#)
 - host enable (HOST_EN) bit, [8-25](#)
 - HOST_END (host endianness) bit, [8-25](#)
 - host endianness (HOST_END) bit, [8-25](#)
 - HOST_EN (host enable) bit, [8-25](#)
 - HOST_FLUSH (FIFO flush) bit, [8-25](#)
 - host handshake (HSHK) bit, [8-27](#)
 - host interrupt request (HIRQ) bit, [8-27](#)
 - host ready override (HRDY_OVR) bit, [8-25](#)
 - HOST_STATUS (HOST status) register, [8-27](#)
 - HOST status (HOST_STATUS) register, [8-27](#)
 - host timeout count (COUNT_TIMEOUT) bits, [8-29](#)
 - HOST_TIMEOUT (HOST acknowledge mode timeout) register, [8-29](#)
 - host timeout (TIMEOUT) bit, [8-27](#)
 - hours[3:0] field, [14-21](#), [14-23](#)
 - hours[4] bit, [14-21](#), [14-23](#)
 - hours event flag bit, [14-22](#)
 - hours interrupt enable bit, [14-21](#)
 - HRDY_OVR (host ready override) bit, [8-25](#)
 - HSHK (host handshake) bit, [8-27](#)
- ## I
- I²C bus standard, [1-15](#)
 - I²S, [1-19](#)
 - ICIE (illegal gray/binary code interrupt enable) bit, [13-28](#)

- ICII (illegal gray/binary code interrupt identifier) bit, [13-29](#)
- ICPLB Address registers (ICPLB_ADDRx), [3-69](#)
- ICPLB_ADDRx (ICPLB Address registers), [3-69](#)
- ICPLB Data registers (ICPLB_DATAx), [3-63](#)
- ICPLB_DATAx (ICPLB Data registers), [3-63](#)
- ICPLB Fault Address register (ICPLB_FAULT_ADDR), [3-72](#)
- ICPLB_FAULT_ADDR (ICPLB Fault Address register), [3-72](#)
- ICPLB_STATUS (ICPLB Status register), [3-72](#)
- ICPLB Status register (ICPLB_STATUS), [3-72](#)
- idle state
 - waking from, [4-13](#)
- image data format (IMG_FORM) bit, [7-38](#)
- image FIFO status (IMG_STAT) bits, [7-38](#)
- IMEM_CONTROL (Instruction Memory Control register), [3-9](#), [3-54](#)
- IMG_FORM (image data format) bit, [7-38](#)
- IMG_STAT (image FIFO status) bits, [7-38](#)
- index (definition), [3-85](#)
- INIT bit, [17-40](#)
- initcall address/symbol command, [17-41](#)
- initcode routines, [17-39](#)
- initializing
 - DMA, [5-25](#)
- init initcode.dxe command, [17-41](#)
- inner loop address increment registers (DMAx_X_MODIFY), [5-101](#) (MDMA_yy_X_MODIFY), [5-101](#)
- inner loop count registers (DMAx_X_COUNT), [5-96](#) (MDMA_yy_X_COUNT), [5-96](#)
- INPDIS (CUD and CDZ input disable) bit, [13-27](#)
- input clock, *See* CLKIN
- input delay bit, [18-28](#)
- Inserting Wait States using ARDY (figure), [6-80](#)
- instruction cache
 - coherency, [3-21](#)
- instruction fetches, [3-55](#)
- Instruction Memory Control register (IMEM_CONTROL), [3-9](#), [3-54](#)
- instructions, [1-35](#)
 - interlocked pipeline, [3-76](#)
 - load/store, [3-75](#)
 - See also* instructions by name
 - stored in memory, [3-75](#)
 - synchronizing, [3-78](#)
- Instruction Test Command register (ITEST_COMMAND), [3-25](#)
- Instruction Test Data registers (ITEST_DATAx), [3-26](#)
- Instruction Test registers, [3-24](#) to [3-27](#)
- Interface
 - D Port, [2-10](#)
 - On-Chip L2 Memory, [2-11](#)
- interfaces
 - internal memory, [6-8](#)
 - RTC, [14-3](#)
- internal bank, [G-12](#)
- internal boot ROM, [17-1](#)
- internal clocks, [2-5](#)
- internal interfaces, [2-1](#)
- internal memory, [1-6](#), [3-5](#)
 - interfaces, [6-8](#)
- internal supply regulator, shutting off, [18-22](#)
- interrupt
 - enabling and disabling, [3-83](#)
 - priority watermark, [3-42](#)

Index

- interrupt handler and DMA
 - synchronization, 5-69
 - interrupt mask (CNT_IMASK) register,
 - 13-24, 13-28, A-31
 - interrupt mode (INT_MODE) bit, 8-25
 - Interrupt Priority register (IPRIO), 3-42
 - interrupt request lines, peripheral, 4-2
 - interrupts, 4-1 to 4-46
 - clearing requests, 4-44
 - configuring and servicing, 19-2
 - control of system, 4-6
 - core timer, 11-3
 - default mapping, 4-7
 - definition, 4-6
 - determining source, 4-12
 - DMA channels, 4-13
 - DMA_ERROR, 5-38
 - DMA error, 5-91
 - DMA overflow, 5-51
 - DMA queue completion, 5-71
 - enabling, 4-11
 - general-purpose, 4-6, 4-7
 - general-purpose timers, 10-7, 10-8, 10-18, 10-32
 - generated by peripheral, 4-22
 - handshake MDMA, 5-50
 - initialization, 4-22
 - inputs and outputs, 4-10
 - mapping, 4-11
 - mask function, 4-14
 - multiple sources, 4-24
 - peripheral, 4-6, 4-10, 4-10 to 4-22
 - peripheral IDs, 4-16
 - peripheral interrupt events, 4-16
 - prioritization, 4-11
 - processing, 4-22
 - programming examples, 4-44 to 4-46
 - reset, 17-9
 - routing overview, 4-3, 4-4, 4-5
 - RTC, 14-16
 - shared, 4-11
 - software, 4-10
 - to wake core from idle, 4-13
 - use in managing a descriptor queue, 5-68
 - interrupt service routine, determining
 - source of interrupt, 4-12
 - interrupt status registers
 - (DMAx_IRQ_STATUS), 5-87, 5-88
 - (MDMA_yy_IRQ_STATUS), 5-87, 5-88
 - INT_MODE (interrupt mode) bit, 8-25
 - invalid cache line (definition), 3-85
 - I/O memory space, 1-9
 - IPRIO (Interrupt Priority register), 3-42
 - IRQ_ENA bit, 10-48, 10-56, 10-58
 - ISR and multiple interrupt sources, 4-24
 - ITEST_COMMAND (Instruction Test Command register), 3-25
 - ITEST_DATAx (Instruction Test Data registers), 3-26
 - ITEST register
 - ITEST_COMMAND, 3-24
 - ITEST_DATA0, 3-24
 - ITEST_DATA1, 3-24
 - ITEST registers, 3-25
 - ITHR[15:0] field, 5-125
- ## J
- JTAG, 1-38
- ## L
- L1 data memory, 1-6
 - L1 Data Memory Architecture, 3-33
 - L1 Data SRAM, 3-31
 - L1 instruction memory, 1-6
 - subbanks, 3-13
 - L1 Instruction Memory Bank Architecture, 3-14

- L1 memory. *See* Level 1 (L1) memory;
 - Level 1 (L1) Data Memory; Level 1 (L1) Instruction Memory
 - L1 or L2 memory, DMA access stalls, [2-14](#)
 - L1 scratchpad RAM, [1-6](#)
 - L2 bus, [2-5](#)
 - L2 memory
 - access bus arbitration, [2-13](#)
 - access latency and throughput, [2-14](#)
 - arbitration, [2-12](#)
 - interface, [2-11](#)
 - interface control logic clock rate, [2-12](#)
 - L2 port
 - access priority, [2-13](#)
 - access request, arbitration priority, [2-13](#)
 - large descriptor mode, DMA, [5-22](#)
 - large model mode, DMA, [5-85](#)
 - Latency
 - DMA data transfer, [2-14](#)
 - L2 memory access, [2-14](#)
 - latency
 - DMA, [5-33](#)
 - least recently used algorithm (LRU),
 - definition, [3-85](#)
 - Level 1 (L1) Data Memory, [3-28](#)
 - configuration, [3-7](#)
 - sub-banks, [3-32](#)
 - Level 1 (L1) Instruction Memory, [3-8](#)
 - architecture, [3-15](#)
 - configuration, [3-15](#)
 - DAG reference exception, [3-12](#)
 - dual-port capability, [3-12](#)
 - instruction cache, [3-15](#)
 - sub-bank organization, [3-8](#)
 - Level 1 (L1) memory
 - See also* Level 1 (L1) Data Memory; Level 1 (L1) Instruction Memory
 - address alignment, [3-12](#)
 - architecture, [3-5](#)
 - definition, [3-85](#)
 - L1 Data SRAM, [3-7](#)
 - L1 Instruction SRAM, [3-6](#)
 - scratchpad data SRAM, [3-7](#)
 - Level 2 (L2) memory, [3-49](#)
 - latency, [3-50](#), [3-51](#)
 - latency with cache off, [3-51](#)
 - latency with cache on, [3-50](#)
 - off-chip, [3-53](#)
 - lines per frame) bits, [7-39](#)
 - lines per frame (LPF) bit, [7-39](#)
 - little endian (definition), [3-85](#)
 - load, speculative execution, [3-79](#)
 - loader file, [17-22](#)
 - loader utility, [17-22](#)
 - load operation, [3-75](#)
 - load ordering, [3-77](#)
 - LOCKCNT[15:0] field, [18-29](#)
- ## M
- manual
 - conventions, [-1](#)
 - mapping
 - default interrupt, [4-16](#)
 - peripheral to DMA, [4-14](#), [4-15](#)
 - MAXCIE (max Count Interrupt Enable), [13-28](#)
 - MAXCIE (max count interrupt enable) bit, [13-28](#)
 - MAXCII (max count interrupt identifier) bit, [13-29](#)
 - max count interrupt enable (MAXCIE) bit, [13-28](#)
 - max count interrupt identifier (MAXCII) bit, [13-29](#)

Index

- maximal count (CNT_MAX) register,
 - 13-25, 13-32, A-32
- MBDI bit, 5-51, 5-119
- MDMA controllers, 5-13
- MDMA_ROUND_ROBIN_COUNT[4:0] field, 5-59, 5-127
- MDMA_ROUND_ROBIN_PERIOD field, 5-58, 5-59, 5-127
- MDMA_yy_CONFIG (DMA configuration registers), 5-82
- MDMA_yy_CURR_ADDR (current address registers), 5-93
- MDMA_yy_CURR_DESC_PTR (current descriptor pointer registers), 5-113
- MDMA_yy_CURR_X_COUNT (current inner loop count registers), 5-98, 5-99
- MDMA_yy_CURR_Y_COUNT (current outer loop count registers), 5-105
- MDMA_yy_IRQ_STATUS (interrupt status registers), 5-87, 5-88
- MDMA_yy_NEXT_DESC_PTR (next descriptor pointer registers), 5-110
- MDMA_yy_PERIPHERAL_MAP (peripheral map registers), 5-79
- MDMA_yy_START_ADDR (start address registers), 5-91
- MDMA_yy_X_COUNT (inner loop count registers), 5-96
- MDMA_yy_X_MODIFY (inner loop address increment registers), 5-101
- MDMA_yy_Y_COUNT (outer loop count registers), 5-103
- MDMA_yy_Y_MODIFY (outer loop address increment registers), 5-108
- measurement report, general-purpose timers, 10-29, 10-30, 10-31
- Memory
 - access, latency and throughput, 2-14
 - external, 2-2
 - interface, 2-11
- memory
 - See also* cache; Level 1 (L1) memory;
 - Level 1 (L1) Data Memory; Level 1 (L1) Instruction Memory; Level 2 (L2) memory
 - architecture, 1-5
 - asynchronous interface, 19-7
 - asynchronous region, 6-2
 - configurations, 1-5
 - external, 3-53
 - how instructions are stored, 3-75
 - internal, 1-6
 - internal bank, G-12
 - I/O space, 1-9
 - L1 data, 1-6, 3-28
 - L1 Data SRAM, 3-31
 - L1 instruction, 1-6
 - L1 scratchpad, 1-6
 - Level 2 (L2), 3-49
 - management, 3-54
 - on-chip, 1-6
 - Page Descriptor Table, 3-57
 - protection and properties, 3-54
 - structure, 1-5
 - terminology, 3-84
 - transaction model, 3-74
- Memory Architecture, 3-2
- memory architecture, 3-2
- memory conflict, DMA, 5-60
- memory DMA, 5-13
 - bandwidth, 5-54
 - buffers, 5-15
 - channels, 5-13
 - descriptor structures, 5-74
 - handshake operation, 5-16
 - timing, 5-55
 - transfer operation, starting, 5-15
 - transfer performance, 2-24
 - transfers, 5-3, 5-9
 - word size, 5-14

Memory Management Unit (MMU), [3-54](#)
Memory Map, [3-4](#)
memory map, external (figure), [6-4](#)
memory-mapped registers, *See* MMRs
memory-mapped registers (MMRs), [3-83](#),
[3-84](#)
memory page, [3-55](#)
 attributes, [3-56](#)
memory-to-memory transfer, [5-14](#)
MINCIE (min count interrupt enable) bit,
[13-28](#)
MINCII (min count interrupt identifier)
 bit, [13-29](#)
min count interrupt enable (MINCIE) bit,
[13-28](#)
min count interrupt identifier (MINCII)
 bit, [13-29](#)
minimal count (CNT_MIN) register,
[13-25](#), [13-32](#), [A-32](#)
minutes[5:0] field, [14-21](#), [14-23](#)
minutes event flag bit, [14-22](#)
minutes interrupt enable bit, [14-21](#)
MMR Port, [2-4](#)
MMRs, [1-9](#)
 address range, [A-3](#)
 width, [A-3](#)
modified (definition), [3-84](#)
MSEL[5:0] field, [18-4](#), [18-28](#)
multiple interrupt sources, [4-24](#)
multiply row by 4 (BV_MULT4) bit, [7-45](#)
multiply row by 4 (GU_MULT4) bit, [7-44](#)
multiply row by 4 (RY_MULT4) bit, [7-43](#),
[17-27](#), [17-111](#), [17-112](#), [17-113](#),
[17-114](#), [17-116](#), [17-117](#), [17-118](#),
[17-119](#), [17-120](#), [17-121](#), [17-122](#),
[17-123](#), [17-125](#), [17-131](#), [17-132](#)
MUX_y (port x bit y) bits, [9-37](#)

N
NDPH bit, [5-29](#)

NDPL bit, [5-29](#)
NDSIZE[3:0] field, [5-23](#), [5-82](#), [5-85](#)
 legal values, [5-40](#)
next descriptor pointer registers
 (DMA_x_NEXT_DESC_PTR), [5-110](#)
 (MDMA_{yy}_NEXT_DESC_PTR),
 [5-110](#)
nFlags variable, [17-132](#)
NOPREBOOT, [15-60](#), [15-61](#), [15-64](#),
[15-67](#), [17-110](#)

O

offsets, DMA descriptor elements, [5-24](#)
OI bit, [5-119](#)
OIE bit, [5-119](#)
onboard regulation, bypassing, [18-20](#)
on-chip memory, [1-6](#)
on-chip switching regulator controller,
[18-16](#)
open page, [G-1](#)
operating modes, [18-8](#)
 active, [1-32](#), [18-9](#)
 deep sleep, [1-33](#), [18-10](#)
 full on, [1-32](#), [18-8](#)
 hibernate state, [1-33](#), [18-10](#)
 sleep, [1-32](#), [18-9](#)
 transition, [18-11](#), [18-12](#)
optimization, of DMA performance, [5-51](#)
ordering
 loads and stores, [3-77](#)
 weak and strong, [3-77](#)
oscilloscope probes, [19-19](#)
OTP_ALTERNATE_HWAIT, [17-116](#)
OTP_EBIU_AMBCTL, [17-120](#)
OTP_EBIU_AMG, [17-119](#)
OTP_EBIU_DDRCTL, [17-123](#)
OTP_EBIU_DDRCTL0, [17-123](#)
OTP_EBIU_DDRCTL2, [17-125](#)
OTP_EBIU_DDRCTL3, [17-125](#)
OTP_EBIU_FCTL, [17-120](#)

Index

OTP_ENA_CLKOUT, [17-116](#)
OTP_INVALID, [17-116](#)
OTP_LOAD_PAGE17H, [17-116](#)
OTP_LOAD_PAGE17L, [17-116](#)
OTP memory
 bfrom_OtpCommand(), [16-9](#)
 BFROM_OTP_READ, [16-17](#)
 bfrom_OtpRead(), [16-9](#)
 BFROM_OTP_WRITE, [16-19](#)
 bfrom_OtpWrite(), [16-9](#)
 BFROM_TOP_COMMAND, [16-15](#)
 error correction, [16-5](#)
 map, [16-3](#)
 overview, [16-1](#)
OTP_SET_BMODES, [17-116](#)
OTP_SET_CALIB, [17-116](#)
OTP_SET_EBIU_ASYNC, [17-116](#)
OTP_SET_EBIU_SYNC, [17-116](#)
OTP_SET_FCTL, [17-119](#)
OTP_SET_MODE, [17-119](#)
OTP_SET_PLL, [17-116](#)
OTP_SET_VR, [17-116](#)
OTP_SPI_BAUD, [17-117](#)
OTP_SPI_FASTREAD, [17-117](#)
OTP_TARGET_PLL_CTL, [17-118](#)
OTP_TARGET_PLL_DIV, [17-117](#)
OTP_TARGET_VER_CTL, [17-118](#)
OTP_TWI_CLKDIV, [17-117](#)
OTP_TWI_PRESCALE, [17-117](#)
OTP_TWI_TYPE, [17-116](#)
OTP_USB_CALIB, [17-121](#)
OUT_DIS bit, [10-47](#), [10-48](#), [10-57](#)
outer loop address increment registers
 (DMAx_Y_MODIFY), [5-108](#)
 (MDMA_yy_Y_MODIFY), [5-108](#)
outer loop count registers
 (DMAx_Y_COUNT), [5-103](#)
 (MDMA_yy_Y_COUNT), [5-103](#)
OUT_FORM (output data format) bit,
 [7-38](#)
output data format (OUT_FORM) bit,
 [7-38](#)
output delay bit, [18-28](#)
output pad disable, timer, [10-16](#)
outputs, programmable pins, [19-18](#)
overflow interrupt, DMA, [5-51](#)
overlay A enable (OVR_A_EN) bit, [7-38](#)
overlay A horizontal end (A_HEND) bits,
 [7-40](#)
overlay A horizontal start (A_HSTART)
 bits, [7-40](#)
overlay A transparency (A_TRANSP) bits,
 [7-42](#)
overlay A vertical end (A_VEND) bits,
 [7-41](#)
overlay A vertical start (A_VSTART) bits,
 [7-41](#)
overlay B enable (OVR_B_EN) bit, [7-38](#)
overlay B horizontal end (B_HEND) bits,
 [7-40](#)
overlay B horizontal start (B_HSTART)
 bits, [7-40](#)
overlay B transparency (B_TRANSP) bits,
 [7-42](#)
overlay B vertical end (B_VEND) bits,
 [7-41](#)
overlay B vertical start (B_VSTART) bits,
 [7-41](#)
overlay data format (OVR_FORM) bit,
 [7-38](#)
overlay FIFO status (OVR_STAT) bits,
 [7-38](#)
overlay interrupt enable (OVR_INT_EN)
 bit, [7-42](#)
overlay interrupt status
 (OVR_INT_STAT) bit, [7-42](#)
OVR_A_EN (overlay A enable) bit, [7-38](#)
OVR_B_EN (overlay B enable) bit, [7-38](#)
OVR_FORM (overlay data format) bit,
 [7-38](#)

OVR_INT_EN (overlay interrupt enable) bit, [7-42](#)

OVR_INT_STAT (overlay interrupt status) bit, [7-42](#)

OVR_STAT (overlay FIFO status) bits, [7-38](#)

P

PAB

clocking, [18-2](#)

PAB, Peripheral Access Bus, [2-4](#), [2-5](#), [2-15](#)

page 0x14, [17-115](#), [17-119](#)

page 0x15H, [17-120](#)

Page 0x16L, [17-122](#)

page 0x16L, [17-124](#)

PDWN bit, [18-28](#)

Performance

DAB, DCB, and DEB, [2-23](#)

DAB bus, [2-23](#)

DCB bus, [2-23](#)

DEB bus, [2-25](#)

performance

DMA, [5-53](#)

memory DMA, [5-54](#)

memory DMA transfers, [2-24](#)

optimization, DMA, [5-51](#)

PERIOD_CNT bit, [10-14](#), [10-23](#), [10-29](#), [10-48](#), [10-56](#)

period value[15:0] field, [11-7](#)

period value[31:16] field, [11-7](#)

Peripheral

DMA, [2-18](#)

Group 1, [2-4](#)

Peripheral Access Bus (PAB), [2-4](#), [2-5](#), [2-15](#)

peripheral DMA, [5-10](#)

peripheral DMA channels, [5-52](#)

peripheral DMA start address registers, [5-91](#)

peripheral DMA transfers, [5-2](#)

peripheral error interrupts, [5-90](#)

peripheral interrupt request lines, [4-2](#)

peripheral interrupts, [4-6](#), [4-10](#), [4-10](#) to [4-22](#)

peripheral map registers

(DMAx_PERIPHERAL_MAP), [5-79](#)

(MDMA_yy_PERIPHERAL_MAP), [5-79](#)

peripherals, [1-2](#)

and buses, [1-3](#)

configuring for an IVG priority, [4-27](#)

default mapping to DMA, [5-10](#)

and DMA controller, [5-40](#)

DMA support, [1-3](#)

interrupt events, [4-16](#)

interrupt generated by, [4-22](#)

interrupt IDs, [4-16](#)

interrupts, clearing, [4-44](#)

level-sensitivity of interrupts, [4-46](#)

list of, [1-2](#)

mapping to DMA, [4-14](#), [4-15](#)

remapping DMA assignment, [5-11](#)

switching from DMA to non-DMA, [5-91](#)

used to wake from idle, [4-13](#)

PHYWE bit, [18-30](#)

pin assignment (PINTx_ASSIGN) registers, [9-56](#)

pin interrupt edge clear (PINTx_EDGE_CLEAR) registers, [9-52](#)

pin interrupt edge set (PINTx_EDGE_SET) registers, [9-51](#)

pin interrupt invert clear (PINTx_INVERT_CLEAR) registers, [9-55](#)

pin interrupt invert set (PINTx_INVERT_SET) registers, [9-54](#)

pin interrupt latch (PINTx_LATCH) registers, [9-50](#)

Index

- pin interrupt mask clear
 - (PINT_x_MASK_CLEAR) registers, [9-47](#)
- pin interrupt mask set
 - (PINT_x_MASK_SET) registers, [9-46](#)
- pin interrupt pin state
 - (PINT_x_PINSTATE) registers, [9-53](#)
- pin interrupt request (PINT_x_REQUEST) registers, [9-48](#)
- pin interrupt x (PIQ_x) bits, [9-46](#), [9-47](#), [9-48](#), [9-50](#), [9-51](#), [9-52](#), [9-53](#), [9-54](#), [9-55](#)
- pins, [19-1](#)
 - unused, [19-18](#)
- PINT_x_ASSIGN (pin assignment) registers, [9-56](#)
- PINT_x_EDGE_CLEAR (pin interrupt edge clear) registers, [9-52](#)
- PINT_x_EDGE_SET (pin interrupt edge set) registers, [9-51](#)
- PINT_x_INVERT_CLEAR (pin interrupt invert clear) registers, [9-55](#)
- PINT_x_INVERT_SET (pin interrupt invert set) registers, [9-54](#)
- PINT_x_LATCH (pin interrupt latch) registers, [9-50](#)
- PINT_x_MASK_CLEAR (pin interrupt mask clear) registers, [9-47](#)
- PINT_x_MASK_SET (pin interrupt mask set) registers, [9-46](#)
- PINT_x_PINSTATE (pin interrupt pin state) registers, [9-53](#)
- PINT_x_REQUEST (pin interrupt request) registers, [9-48](#)
- pipeline
 - interlocked, [3-76](#)
- pipeline, lengths of, [5-63](#)
- pipelining
 - DMA requests, [5-48](#)
- PIQ_x (pin interrupt x) bits, [9-46](#), [9-47](#), [9-48](#), [9-50](#), [9-51](#), [9-52](#), [9-53](#), [9-54](#), [9-55](#)
- PIXC_AHEND (PIXC overlay A horizontal end) register, [7-36](#), [7-40](#), [A-16](#)
- PIXC_AHSTART (PIXC overlay A horizontal start) register, [7-36](#), [7-40](#), [A-16](#)
- PIXC_ATRANSP (PIXC overlay A transparency) register, [7-36](#), [7-42](#), [A-16](#)
- PIXC_AVEND (PIXC overlay A vertical end) register, [7-36](#), [7-41](#), [A-16](#)
- PIXC_AVSTART (PIXC overlay A vertical start) register, [7-36](#), [7-41](#), [A-16](#)
- PIXC_BHEND (PIXC overlay B horizontal end) register, [7-36](#), [7-40](#), [A-16](#)
- PIXC_BHSTART (PIXC overlay B horizontal start) register, [7-36](#), [7-40](#), [A-16](#)
- PIXC_BTRANSP (PIXC overlay B transparency) register, [7-36](#), [7-42](#), [A-16](#)
- PIXC_BVCON (PIXC B/V conversion coefficients) register, [7-36](#), [7-45](#), [A-16](#)
- PIXC B/V conversion coefficients (PIXC_BVCON) register, [7-36](#), [7-45](#), [A-16](#)
- PIXC_BVEND (PIXC overlay B vertical end) register, [7-36](#), [7-41](#), [A-16](#)
- PIXC_BVSTART (PIXC overlay B vertical start) register, [7-36](#), [7-41](#), [A-16](#)
- PIXC_CCBIAS (PIXC color conversion bias) register, [7-36](#), [7-46](#), [A-17](#)
- PIXC color conversion bias (PIXC_CCBIAS) register, [7-36](#), [7-46](#), [A-17](#)

- PIXC control (PIXC_CTL) register, 7-35, 7-38, A-15
- PIXC_CTL (PIXC control) register, 7-35, 7-38, A-15
- PIXC_EN (pixel compositor enable) bit, 7-38
- PIXC_GUCON (PIXC G/U conversion coefficients) register, 7-36, 7-44, A-16
- PIXC G/U conversion coefficients (PIXC_GUCON) register, 7-36, 7-44, A-16
- PIXC interrupt status (PIXC_INTRSTAT) register, 7-36, 7-42, A-16
- PIXC_INTRSTAT (PIXC interrupt status) register, 7-36, 7-42, A-16
- PIXC lines per frame (PIXC_LPF) register, 7-35, 7-39, A-15
- PIXC_LPF (PIXC lines per frame) register, 7-35, 7-39, A-15
- PIXC overlay A horizontal end (PIXC_AHEND) register, 7-36, 7-40, A-16
- PIXC overlay A horizontal start (PIXC_AHSTART) register, 7-36, 7-40, A-16
- PIXC overlay A transparency (PIXC_ATRANSF) register, 7-36, 7-42, A-16
- PIXC overlay A vertical end (PIXC_AVEND) register, 7-36, 7-41, A-16
- PIXC overlay A vertical start (PIXC_AVSTART) register, 7-36, 7-41, A-16
- PIXC overlay B horizontal end (PIXC_BHEND) register, 7-36, 7-40, A-16
- PIXC overlay B horizontal start (PIXC_BHSTART) register, 7-36, 7-40, A-16
- PIXC overlay B transparency (PIXC_BTRANSP) register, 7-36, 7-42, A-16
- PIXC overlay B vertical end (PIXC_BVEND) register, 7-36, 7-41, A-16
- PIXC overlay B vertical start (PIXC_BVSTART) register, 7-36, 7-41, A-16
- PIXC pixels per line (PIXC_PPL) register, 7-35, 7-39, A-15
- PIXC_PPL (PIXC pixels per line) register, 7-35, 7-39, A-15
- PIXC_RYCON (PIXC R/Y conversion coefficients) register, 7-36, 7-43, 17-27, 17-111, 17-112, 17-113, 17-114, 17-116, 17-117, 17-118, 17-119, 17-120, 17-121, 17-122, 17-123, 17-125, 17-131, 17-132, A-16
- PIXC R/Y conversion coefficients (PIXC_RYCON) register, 7-36, 7-43, 17-27, 17-111, 17-112, 17-113, 17-114, 17-116, 17-117, 17-118, 17-119, 17-120, 17-121, 17-122, 17-123, 17-125, 17-131, 17-132, A-16
- PIXC_TC (PIXC transparent color) register, 7-36, 7-47, A-17
- PIXC transparent color (PIXC_TC) register, 7-36, 7-47, A-17
- pixel compositor enable (PIXC_EN) bit, 7-38
- pixels per line value (PPL) bits, 7-39

Index

- PLL, 18-1 to ??
 - active mode, 18-9
 - applying power to the PLL, 18-13
 - block diagram, 18-3
 - BYPASS bit, 18-9
 - bypassing onboard regulation, 18-20
 - CCLK derivation, 18-3
 - changing clock ratio, 18-6
 - clock dividers, 18-4
 - clocking to SDRAM, 18-10
 - clock multiplier ratios, 18-3
 - configuration, 18-3
 - control bits, 18-11
 - design, 18-2
 - disabled, 18-13
 - divide frequency, 18-4
 - DMA access, 18-9
 - dynamic power management controller (DPMC), 18-7
 - enabled, 18-13
 - enabled but bypassed, 18-9
 - maximum performance mode, 18-8
 - multiplier select (MSEL) field, 18-4
 - operating modes, operational characteristics, 18-8
 - operating mode transitions, 18-14
 - PDWN bit, 18-11
 - PLL_LOCKED bit, 18-15
 - PLL_OFF bit, 18-13
 - PLL status (table), 18-8
 - power domains, 18-16
 - powering down core, 18-22
 - power savings by operating mode (table), 18-8
 - relocking after changes, 18-15
 - removing power to the PLL, 18-13
 - RTC interrupt, 18-10, 18-16
 - SCLK derivation, 18-2, 18-3
 - sleep mode, 18-9, 18-15
 - STOPCK bit, 18-11
 - voltage control, 18-7, 18-20
- PLL control register (PLL_CTL), 18-26, 18-28
- PLL_CTL (PLL control register), 18-4, 18-26, 18-28
- PLL divide register (PLL_DIV), 18-27
- PLL_DIV (PLL divide register), 18-5, 18-26, 18-27
- PLL_LOCKCNT (PLL lock count register), 18-27, 18-29
- PLL lock count register (PLL_LOCKCNT), 18-29
- PLL_LOCKED bit, 18-29
- PLL_OFF bit, 18-28
- PLL_STAT (PLL status register), 18-26, 18-29
- PLL status register (PLL_STAT), 18-29
- PMAP[3:0] field, 5-10, 5-55, 5-79
- polling DMA registers, 5-62
- Port access priority, 2-13
- Port access request, arbitration priority, 2-13
- port data clear (PORTx_CLEAR) registers, 9-44
- port data (PORTx) registers, 9-42
- port data set (PORTx_SET) registers, 9-43
- PORT_DIR bit, 13-32
- port direction clear (PORTx_DIR_CLEAR) registers, 9-40
- port direction set (PORTx_DIR_SET) registers, 9-39
- port F
 - and general-purpose timers, 10-5
 - timer port setup, 10-58
- port function enable (PORTx_FER) registers, 9-36
- port input enable (PORTx_INEN) registers, 9-41
- ports, 1-14

- port x bit y (Pxy) bits, [9-36](#), [9-39](#), [9-40](#),
[9-41](#), [9-42](#), [9-43](#), [9-44](#)
 - PORTx_CLEAR (port data clear) registers,
[9-44](#)
 - PORTx_DIR_CLEAR (port direction
clear) registers, [9-40](#)
 - PORTx_DIR_SET (port direction set)
registers, [9-39](#)
 - PORTx_FER (port function enable)
registers, [9-36](#)
 - PORTx_INEN (port input enable)
registers, [9-41](#)
 - port x multiplexer control (PORTx_MUX)
registers, [9-37](#)
 - PORTx_MUX (port x multiplexer control)
registers, [9-37](#)
 - port x mux y (MUXy) bits, [9-37](#)
 - PORTx (port data) registers, [9-42](#)
 - PORTx_SET (port data set) registers, [9-43](#)
 - power dissipation, [18-16](#)
 - power domains, [18-16](#)
 - powering down core, [18-22](#)
 - power management, [1-32](#), [18-1](#) to ??
 - power supply management, [18-16](#)
 - PPI
 - GP output, [13-3](#), [13-5](#), [13-6](#), [13-9](#),
[13-10](#), [13-20](#), [13-22](#)
 - PPL (pixel per line value) bits, [7-39](#)
 - P Port, [2-4](#)
 - interface, [2-9](#)
 - preboot, controlled by OTP programming,
[17-4](#)
 - preboot routine, [17-10](#)
 - precharge command, [G-18](#)
 - PREFETCH instruction, [3-6](#), [3-43](#)
 - PREN bit, [14-23](#)
 - prescaler, RTC, [14-2](#)
 - prescaler enable register (RTC_PREN),
[14-23](#)
 - priorities
 - peripheral DMA operations, [5-57](#)
 - prioritization
 - DMA, [5-55](#) to [5-61](#)
 - interrupts, [4-11](#)
 - Priority
 - L2 port access, [2-13](#)
 - Sys L2 port access, [2-13](#)
 - probes, oscilloscope, [19-19](#)
 - processor block diagram, [1-4](#)
 - Processor Bus Hierarchy, [2-3](#)
 - Processor Core and L1 Memory Block
Diagram, [2-7](#)
 - programmable outputs, [19-18](#)
 - programming model
 - cache memory, [3-5](#)
 - PS bit, [5-119](#)
 - PULSE_HI bit, [10-17](#), [10-19](#), [10-48](#),
[10-56](#)
 - PULSE_HI toggle mode, [10-19](#)
 - pulse width count and capture mode, *See*
WDTH_CAP mode
 - pulse width modulation mode, *See*
PWM_OUT mode
 - PWM_CLK clock, [10-24](#)
 - PWM_OUT mode, [10-14](#) to [10-26](#), [10-52](#)
 - control bit and register usage, [10-56](#)
 - error prevention, [10-54](#)
 - externally clocked, [10-23](#)
 - PULSE_HI toggle mode, [10-19](#)
 - stopping the timer, [10-24](#)
 - Pxy (port x bit y) bits, [9-36](#), [9-39](#), [9-40](#),
[9-41](#), [9-42](#), [9-43](#), [9-44](#)
- ## Q
- query semaphore, [19-4](#)
 - quick boot, [17-43](#)

Index

R

RBC bit, [5-47](#), [5-119](#)

READY (DMA ready) bit, [8-27](#)

real-time clock, *See* RTC

register-based DMA, [5-17](#)

registers

See also registers by name

diagram conventions, [-li](#)

rotary counter, [13-24](#), [A-31](#)

system, [A-3](#)

regulator controller, switching, [18-16](#)

REP bit, [5-8](#), [5-48](#), [5-119](#)

replacement policy, [3-39](#)

definition, [3-85](#)

request data control command, DMA, [5-43](#)

request data urgent control command,

DMA, [5-44](#)

resampling mode (UDS_MOD) bit, [7-38](#)

reset

effect on memory configuration, [3-30](#)

RESET_DOUBLE, [15-34](#), [17-108](#)

RESET_DOUBLE bit, [15-34](#), [17-108](#)

RESET pin, [17-6](#)

resets

core and system, [8-31](#), [17-154](#)

core-only software, [17-5](#)

hardware, [17-7](#)

interrupts, [17-9](#)

software, [17-6](#)

system software, [17-4](#)

watchdog timer, [17-6](#)

RESET_SOFTWARE, [15-34](#), [17-108](#)

RESET_SOFTWARE bit, [15-34](#), [17-108](#)

reset vector, [17-1](#)

reset vector addresses, [17-2](#)

RESET_WDOG, [15-34](#), [17-108](#)

RESET_WDOG bit, [12-5](#), [15-34](#), [15-38](#),
[17-108](#)

resource sharing, with semaphores, [19-3](#)

restart control command, DMA, [5-42](#)

restart or finish control command, receive,
[5-45](#)

restart or finish control command,
transmit, [5-45](#)

restrictions

DMA control commands, [5-44](#)

DMA work unit, [5-33](#)

RETI register, [17-9](#)

rotary counter registers, [13-24](#), [A-31](#)

round robin operation, MDMA, [5-58](#)

routing of interrupts, [4-3](#), [4-4](#), [4-5](#)

RTC, [1-29](#), [14-1](#) to [14-28](#)

alarm clock features, [14-2](#)

alarm feature, [14-27](#)

clock rate, [14-5](#)

clock requirements, [14-5](#)

code examples, [14-24](#)

counters, [14-2](#)

digital watch features, [14-1](#)

disabling prescaler, [14-6](#)

enabling prescaler, [14-5](#), [14-24](#)

interfaces, [14-3](#)

interrupt structure, [14-16](#)

prescaler, [14-2](#)

programming model, [14-7](#)

registers, table, [14-20](#)

state transitions, [14-17](#)

stopwatch, [14-2](#), [14-25](#)

synchronization, [14-6](#)

test mode, [14-6](#)

RTC alarm register (RTC_ALARM),
[14-23](#)

RTC_ALARM (RTC alarm register),
[14-20](#), [14-23](#)

RTC_ICTL (RTC interrupt control
register), [14-20](#), [14-21](#)

RTC interrupt control register
(RTC_ICTL), [14-21](#)

RTC interrupt status register
(RTC_ISTAT), [14-22](#)

- RTC_ISTAT (RTC interrupt status register), [14-20](#), [14-22](#)
 - RTC_PREN bit, [14-5](#)
 - RTC_PREN (prescaler enable register), [14-5](#), [14-20](#), [14-23](#)
 - RTC_STAT (RTC status register), [14-20](#), [14-21](#)
 - RTC status register (RTC_STAT), [14-21](#)
 - RTC stopwatch count register (RTC_SWCNT), [14-22](#)
 - RTC_SWCNT (RTC stopwatch count register), [14-20](#), [14-22](#)
 - RY_MULT4 (multiply row by 4) bit, [7-43](#), [17-27](#), [17-111](#), [17-112](#), [17-113](#), [17-114](#), [17-116](#), [17-117](#), [17-118](#), [17-119](#), [17-120](#), [17-121](#), [17-122](#), [17-123](#), [17-125](#), [17-131](#), [17-132](#)
 - RY_TRANS (transparent color - R/Y) bits, [7-47](#)
- S**
- scale value[7:0] field, [11-8](#)
 - scaling, of core timer, [11-7](#)
 - SCLK, [18-5](#)
 - derivation, [18-2](#)
 - disabling, [18-22](#)
 - status by operating mode (table), [18-8](#)
 - scratchpad memory, and booting, [17-24](#)
 - scratchpad SRAM, [3-7](#)
 - SDRAM, [2-2](#)
 - banks, [3-53](#)
 - bank size, [6-2](#)
 - memory banks, [6-4](#)
 - memory space, [6-2](#)
 - sizes supported, [3-53](#)
 - seconds (1 Hz) event flag bit, [14-22](#)
 - seconds (1Hz) interrupt enable bit, [14-21](#)
 - seconds[5:0] field, [14-21](#), [14-23](#)
 - semaphores, [19-3](#)
 - example code, [19-4](#)
 - query, [19-4](#)
 - set associative (definition), [3-85](#)
 - set (definition), [3-85](#)
 - shared interrupts, [4-11](#)
 - SIC_IAR0 (system interrupt assignment register 0), [4-28](#), [8-25](#), [8-27](#), [8-29](#)
 - SIC_IAR1 (system interrupt assignment register 1), [4-28](#)
 - SIC_IAR2 (system interrupt assignment register 2), [4-29](#)
 - SIC_IAR3 (system interrupt assignment register 3), [4-29](#), [4-30](#), [4-31](#), [4-32](#), [4-33](#)
 - SIC_IMASK (system interrupt mask register), [4-11](#), [4-35](#), [4-36](#), [4-37](#), [4-38](#), [4-39](#), [4-40](#), [4-41](#), [4-42](#), [4-43](#)
 - SIC_ISR (system interrupt status register), [4-12](#)
 - SIC_IWR (system interrupt wakeup-enable register), [4-13](#)
 - signal integrity, [19-14](#)
 - sine wave input, [1-31](#)
 - single pulse generation, timer, [10-16](#)
 - size of accesses, timer registers, [10-39](#)
 - slaves
 - EBIU, [6-8](#)
 - sleep mode, [1-32](#), [18-9](#)
 - small descriptor mode, DMA, [5-22](#)
 - small model mode, DMA, [5-85](#)
 - software interrupts, [4-10](#)
 - software management of DMA, [5-61](#)
 - software reset, [17-6](#), [17-107](#)
 - software reset register (SWRST), [15-34](#), [17-108](#)
 - software watchdog timer, [1-30](#), [12-1](#)
 - source channels, memory DMA, [5-13](#)
 - SPI, [1-21](#)
 - slave boot mode, [17-76](#)

Index

- SPORT, 1-19
- SRAM
 - interface, 19-7
 - L1 data, 3-31
 - L1 Data Memory, 3-7
 - L1 instruction access, 3-12
 - L1 Instruction Memory, 3-6
 - scratchpad, 3-7
- SSEL[3:0] field, 18-5, 18-27
- Stalls
 - DMA access to L1 or L2 memory, 2-14
- stalls
 - pipeline, 3-76
- start address registers
 - (DMAx_START_ADDR), 5-91
 - (MDMA_yy_START_ADDR), 5-91
- state transitions, RTC, 14-17
- status (CNT_STATUS) register, 13-24, 13-29, A-31
- STI. *See* Enable Interrupts (STI)
- STOPCK bit, 18-28
- stop mode, DMA, 5-18, 5-84
- stopping DMA transfers, 5-37
- stopwatch count[15:0] field, 14-22
- stopwatch function, RTC, 14-2
- store operation, 3-75
- store ordering, 3-77
- streams, memory DMA, 5-14
- strong ordering requirement, 3-83
- subbanks
 - L1 instruction memory, 3-13
- supervisor mode, 17-9
- support, technical or customer, -xlvi
- switching frequency values, 18-20
- switching regulator controller, 18-16
- SWRESET, 15-60, 15-61, 15-64, 15-67, 17-110
- SWRST, software reset register, 17-107, 17-109
- SWRST (software reset register), 15-34, 17-108
- SYNC, 3-79
- SYNC bit, 5-34, 5-36, 5-73, 5-82, 5-85
- synchronization
 - interrupt-based methods, 5-62
 - of descriptor queue, 5-68
 - of DMA, 5-61 to 5-72
- synchronized transition, DMA, 5-36
- SYSCR (system reset configuration register), 15-60, 15-61, 15-64, 15-67, 17-109, 17-110
- System
 - DMA access request, 2-12
 - L2 bus, 2-5
 - overview, 2-8
- system clock (SCLK), 18-2
 - managing, 19-2
- system design, 19-1 to 19-20
 - high frequency considerations, 19-5
 - recommendations and suggestions, 19-15
 - recommended reading, 19-19
- system interrupt assignment register 0 (SIC_IAR0), 4-28, 8-25, 8-27, 8-29
- system interrupt assignment register 1 (SIC_IAR1), 4-28
- system interrupt assignment register 2 (SIC_IAR2), 4-29
- system interrupt assignment register 3 (SIC_IAR3), 4-29, 4-30, 4-31, 4-32, 4-33
- system interrupt controller (SIC), 4-2, 4-6 registers, 4-26
- system interrupt mask register (SIC_IMASK), 4-11, 4-35, 4-36, 4-37, 4-38, 4-39, 4-40, 4-41, 4-42, 4-43
- system interrupt processing, 4-22
- system interrupts, 4-6

system interrupt status register (SIC_ISR),
4-12

system interrupt wakeup-enable register
(SIC_IWR), 4-13

system peripheral clock, *See* SCLK

system peripherals, 1-2

SYSTEM_RESET, 15-34, 17-108

system reset, 17-1 to ??

SYSTEM_RESET[2:0] field, 15-34,
15-38, 17-108

system reset configuration register
(SYSCR), 15-60, 15-61, 15-64,
15-67, 17-109, 17-110

system software reset, 17-4

T

TACIx pins, 10-6, 10-35

TACLKx pins, 10-6

tag (definition), 3-85

target address, 17-31

TAUTORLD bit, 11-3, 11-5

TC_EN (transparent color enable) bit,
7-38

TCNTL (core timer control register), 11-3,
11-5

TCOUNT (core timer count register),
11-3, 11-6

technical support, -xlviii

termination, DMA, 5-37

test point access, 19-19

TESTSET instruction, 19-3

throughput

- achieved by interlocked pipeline, 3-76
- achieved by SRAM, 3-5
- DMA, 5-52
- from DMA system, 5-51

Throughput for L2 memory access, 2-14

TIMDISx bit, 10-41, 10-42, 10-43

TIMENx bit, 10-40, 10-41

TIMEOUT (host timeout) bit, 8-27

timer configuration registers
(TIMERx_CONFIG), 10-47, 10-48

timer counter[15:0] field, 10-51

timer counter[31:16] field, 10-51

timer counter registers
(TIMERx_COUNTER), 10-49,
10-51

TIMER_DISABLE bit, 10-56

timer disable register (TIMER_DISABLE),
10-41, 10-42, 10-43

TIMER_DISABLE (timer disable register),
10-41, 10-42, 10-43

TIMER_ENABLE bit, 10-56

timer enable register (TIMER_ENABLE),
10-40, 10-41

TIMER_ENABLE (timer enable register),
10-40, 10-41

timer period[15:0] field, 10-54

timer period[31:16] field, 10-54

timer period registers
(TIMERx_PERIOD), 10-52, 10-54

timers, 1-21, 10-1 to 10-67

- core, -xlvii to ??, 11-1 to 11-9
- EXT_CLK mode, 10-36 to 10-37
- watchdog, 1-30, 12-1 to 12-11

timer status register (TIMER_STATUS),
10-43, 10-45, 10-46

TIMER_STATUS (timer status register),
10-43, 10-45, 10-46

timer width[15:0] field, 10-55

timer width[31:16] field, 10-55

timer width registers (TIMERx_WIDTH),
10-52, 10-55

TIMERx_CONFIG (timer configuration
registers), 10-47, 10-48

TIMERx_COUNTER (timer counter
registers), 10-7, 10-49, 10-51

TIMERx_PERIOD (timer period
registers), 10-52, 10-54

Index

TIMER_x_WIDTH (timer width registers),
10-52, 10-55

TIMIL_x bits, 10-7, 10-45

timing

- memory DMA, 5-55

TIN_SEL bit, 10-35, 10-48, 10-57

TINT bit, 11-3, 11-5

TMODE[1:0] field, 10-14, 10-48, 10-56

TMPWR bit, 11-3, 11-5

TMRCLK input, 10-6

TMREN bit, 11-3, 11-5

TMR pin, 10-57

TMR_x pins, 10-5, 10-19, 10-35

TOGGLE_HI bit, 10-48, 10-57

TOGGLE_HI mode, 10-19

tools, development, 1-36

TOVF_ERR_x bits, 10-7, 10-11, 10-18,
10-45, 10-47, 10-58

TPERIOD (core timer period register),
11-7

traffic control, DMA, 5-55 to 5-61

Traffic control/optimization, DMA, 2-14

Transfer latency

- DMA data, 2-14

transfer rate

- memory DMA channels, 5-52
- peripheral DMA channels, 5-52

transitions

- continuous DMA, 5-33
- DMA work unit, 5-33
- operating mode, 18-11, 18-12
- synchronized DMA, 5-33

transparent color - B/V (BV_TRANS) bits,
7-47

transparent color enable (TC_EN) bit,
7-38

transparent color - G/U (GU_TRANS)
bits, 7-47

transparent color - R/Y (RY_TRANS) bits,
7-47

triggering DMA transfers, 5-72

TRUN_x bits, 10-24, 10-43, 10-45, 10-58

TSCALE (core timer scale register), 11-7,
11-8

TWI, 1-15

- I²C compatibility, 1-15
- master boot mode, 17-79
- slave boot mode, 17-83

two-dimensional DMA, 5-19

two-wire interface, *See* TWI

U

UART

- autobaud detection, 10-35
- bit rate detection, 10-6

UCIE (up count interrupt enable) bit,
13-28

UCII (up count interrupt identifier) bit,
13-29

UDS_MOD (resampling mode) bit, 7-38

universal asynchronous
receiver/transmitter, *See* UART

unused pins, 19-18

urgent DMA transfers, 5-55

user mode, 17-9

UTE bit, 5-50, 5-119

UTHE[15:0] field, 5-124

V

Valid bit

- clearing, 3-43
- figure, 3-27
- function, 3-17
- in cache-line replacement, 3-19
- in instruction cache invalidation, 3-23

valid (definition), 3-86

VCO, multiplication factors, 18-4

VDK, 1-38

victim (definition), 3-86

- VisualDSP++, 1-36, 17-22
 - debugger, 1-37
 - VLEV[3:0] field, 18-21, 18-30
 - voltage, 18-16
 - changing, 18-20
 - control, 18-7
 - dynamic control, 18-16
 - voltage controlled oscillator (VCO), 18-3
 - voltage level values, 18-21
 - voltage regulator, 1-34
 - voltage regulator control register (VR_CTL), 18-18, 18-30
 - VR_CTL (voltage regulator control register), 18-18, 18-27, 18-30
- W**
- WAKE bit, 18-30
 - wakeup function, 4-14
 - watchdog control register (WDOG_CTL), 12-8, 12-9
 - watchdog count[15:0] field, 12-7
 - watchdog count[31:16] field, 12-7
 - watchdog count register (WDOG_CNT), 12-6, 12-7
 - watchdog status[15:0] field, 12-8
 - watchdog status[31:16] field, 12-8
 - watchdog status register (WDOG_STAT), 12-7, 12-8
 - watchdog timer, 1-30, 12-1 to 12-11
 - block diagram, 12-3
 - disabling, 12-5
 - and emulation mode, 12-2
 - features, 12-1
 - registers, 12-6
 - and reset, 12-5
 - reset, 17-6
 - starting, 12-4
 - zero value, 12-5
 - watermark level (WM_LVL) bits, 7-38
 - waveform generation, pulse width modulation, 10-17
 - Way
 - 1-Way associative (direct-mapped), 3-84
 - definition, 3-86
 - L1 instruction memory as 4-Way set-associative, 3-6
 - priority in cache-line replacement, 3-20
 - WDEN[7:0] field, 12-8
 - WDEV[1:0] field, 12-4, 12-8
 - WDOG_CNT (watchdog count register), 12-4, 12-6, 12-7
 - WDOG_CTL (watchdog control register), 12-8, 12-9
 - WDOG_STAT (watchdog status register), 12-4, 12-7, 12-8
 - WDRESET, 15-60, 15-61, 15-64, 15-67, 17-110
 - WDSIZE[1:0] field, 5-82, 5-86
 - WDTH_CAP mode, 10-27, 10-52
 - control bit and register usage, 10-56
 - WM_LVL (watermark level) bits, 7-38
 - WNR bit, 5-82, 5-86, 8-6
 - work unit
 - completion, 5-31
 - DMA, 5-22
 - interrupt timing, 5-34
 - restrictions, 5-33
 - transitions, 5-33
 - write, 3-86
 - write back (definition), 3-86
 - write buffer depth, 3-42
 - write complete interrupt enable bit, 14-21
 - write pending status bit, 14-22
 - WURESET, 15-60, 15-61, 15-64, 15-67, 17-110
- X**
- X_COUNT[15:0] field, 5-96
 - X_MODIFY[15:0] field, 5-101

Y

Y_COUNT[15:0] field, [5-103](#)

Y_MODIFY[15:0] field, [5-108](#)

Z

ZMZC (CZM zeroes counter enable) bit,
[13-27](#)

ZMZC (CZM zeroes counter enable)
bit, [13-27](#)

