ADSP-BF537 Blackfin® Processor Hardware Reference

Revision 2.0, December 2005

Part Number
82-000555-01

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

Copyright Information

© 2005 Analog Devices, Inc., ALL RIGHTS RESERVED. This document may not be reproduced in any form without prior, express written consent from Analog Devices, Inc.

Printed in the USA.

Disclaimer

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

Trademark and Service Mark Notice

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

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

CONTENTS

PREFACE

Purpose of This Manual xliii
Intended Audience xliii
Manual Contents xliv
What's New in This Manual xlvii
Technical or Customer Support xlvii
Supported Processors xlviii
Product Information xlix
MyAnalog.com xlix
Processor Product Information xlix
Related Documents l
Online Technical Documentation li
Accessing Documentation From VisualDSP++ lii
Accessing Documentation From Windows lii
Accessing Documentation From the Web liii
Printed Manuals liii
VisualDSP++ Documentation Set liii
Hardware Tools Manuals liv
Processor Manuals liv
Data Sheets liv

Conventions lv
Register Diagram Conventions lvi

INTRODUCTION

Peripherals 1-1
Memory Architecture 1-4
Internal Memory 1-6
External Memory 1-6
I/O Memory Space 1-7
DMA Support 1-7
External Bus Interface Unit 1-9
PC133 SDRAM Controller 1-9
Asynchronous Controller 1-9
Ports 1-10
General-Purpose I/O (GPIO) 1-10
Two-Wire Interface 1-11
Controller Area Network 1-12
Ethernet MAC 1-14
Parallel Peripheral Interface 1-14
SPORT Controllers 1-16
Serial Peripheral Interface (SPI) Port 1-18
Timers 1-18
UART Ports 1-19
Real-Time Clock 1-20
Watchdog Timer 1-21

Clock Signals 1-22
Dynamic Power Management 1-23
Full On Mode (Maximum Performance) 1-23
Active Mode (Moderate Power Savings) 1-23
Sleep Mode (High Power Savings) 1-23
Deep Sleep Mode (Maximum Power Savings) 1-24
Hibernate State 1-24
Voltage Regulation 1-24
Boot Modes 1-25
Instruction Set Description 1-27
Development Tools 1-29

CHIP BUS HIERARCHY

Overview 2-1
Interface Overview 2-3
Internal Clocks 2-4
Core Bus Overview 2-4
Peripheral Access Bus (PAB) 2-6
PAB Arbitration 2-6
PAB Agents (Masters, Slaves) 2-6
PAB Performance 2-7
DMA Access Bus (DAB), DMA Core Bus (DCB), DMA External Bus (DEB) 2-8
DAB Arbitration 2-8
DAB Bus Agents (Masters) 2-9
DAB, DCB, and DEB Performance 2-10

External Access Bus (EAB) 2-10
Arbitration of the External Bus 2-11
DEB/EAB Performance 2-11

MEMORY

Memory Architecture 3-1
L1 Instruction SRAM 3-5
L1 Data SRAM 3-7
L1 Data Cache 3-8
Boot ROM 3-8
External Memory 3-8
Processor-Specific MMRs 3-8
DMEM_CONTROL Register 3-9
DTEST_COMMAND Register 3-10

SYSTEM INTERRUPTS

Overview 4-1
Features 4-1
Interfaces 4-2
Description of Operation 4-2
Events and Sequencing 4-2
System Peripheral Interrupts 4-7
Programming Model 4-14
System Interrupt Initialization 4-14
System Interrupt Processing Summary 4-14

System Interrupt Controller Registers 4-16
SIC_IARx Registers 4-18
SIC_IMASK Register 4-20
SIC_ISR Register 4-21
SIC_IWR Register 4-22
Programming Examples 4-23
Clearing Interrupt Requests 4-23

DIRECT MEMORY ACCESS

Overview and Features 5-1
DMA Controller Overview 5-5
External Interfaces 5-6
Internal Interfaces 5-6
Peripheral DMA 5-7
Memory DMA 5-9
Handshaked Memory DMA Mode 5-11
Modes of Operation 5-12
Register-based DMA Operation 5-12
Stop Mode 5-13
Autobuffer Mode 5-14
Two-Dimensional DMA Operation 5-14
Examples of Two-Dimensional DMA 5-15
Descriptor-based DMA Operation 5-16
Descriptor List Mode 5-17
Descriptor Array Mode 5-18

Variable Descriptor Size 5-18
Mixing Flow Modes 5-19
Functional Description 5-20
DMA Operation Flow 5-20
DMA Startup 5-20
DMA Refresh 5-25
Work Unit Transitions 5-27
DMA Transmit and MDMA Source 5-28
DMA Receive 5-30
Stopping DMA Transfers 5-31
DMA Errors (Aborts) 5-32
DMA Control Commands 5-34
Restrictions 5-38
Transmit Restart or Finish 5-38
Receive Restart or Finish 5-39
Handshaked Memory DMA Operation 5-40
Pipelining DMA Requests 5-41
HMDMA Interrupts 5-43
DMA Performance 5-44
DMA Throughput 5-45
Memory DMA Timing Details 5-48
Static Channel Prioritization 5-48
Temporary DMA Urgency 5-50
Memory DMA Priority and Scheduling 5-51
Traffic Control 5-53

Programming Model 5-55
Synchronization of Software and DMA 5-56
Single-buffer DMA Transfers 5-58
Continuous Transfers Using Autobuffering 5-58
Descriptor Structures 5-60
Descriptor Queue Management 5-61
Descriptor Queue Using Interrupts on Every Descriptor 5-62
Descriptor Queue Using Minimal Interrupts 5-63
Software Triggered Descriptor Fetches 5-65
DMA Registers 5-67
DMA Channel Registers 5-68
DMAx_PERIPHERAL_MAP/ MDMA_yy_PERIPHERAL_MAP Registers 5-71
DMAx_CONFIG/MDMA_yy_CONFIG Registers 5-74
DMAx_IRQ_STATUS/ MDMA_yy_IRQ_STATUS Registers 5-78
DMAx_START_ADDR/ MDMA_yy_START_ADDR Registers 5-82
DMAx_CURR_ADDR/MDMA_yy_CURR_ADDR Registers 5-83
DMAx_X_COUNT/MDMA_yy_X_COUNT Registers 5-85
DMAx_CURR_X_COUNT/ MDMA_yy_CURR_X_COUNT Registers 5-86
DMAx_X_MODIFY/ MDMA_yy_X_MODIFY Registers 5-88

DMAx_Y_COUNT/
MDMA_yy_Y_COUNT Registers 5-90
DMAx_CURR_Y_COUNT/
MDMA_yy_CURR_Y_COUNT Registers 5-91
DMAx_Y_MODIFY/MDMA_yy_Y_MODIFY
Registers 5-93
DMAx_NEXT_DESC_PTR/
MDMA_yy_NEXT_DESC_PTR Registers 5-94
DMAx_CURR_DESC_PTR/
MDMA_yy_CURR_DESC_PTR Registers 5-96
HMDMA Registers 5-98
HMDMAx_CONTROL Registers 5-99
HMDMAx_BCINIT Registers 5-101
HMDMAx_BCOUNT Registers 5-101
HMDMAx_ECOUNT Registers 5-102
HMDMAx_ECINIT Registers 5-103
HMDMAx_ECURGENT Registers 5-104
HMDMAx_ECOVERFLOW Registers 5-104
DMA Traffic Control Registers 5-105
DMA_TC_PER Register 5-105
DMA_TC_CNT Register 5-106
Programming Examples 5-107
Register-Based 2D Memory DMA 5-107
Initializing Descriptors in Memory 5-111
Software-Triggered Descriptor Fetch Example 5-114
Handshaked Memory DMA Example 5-116

EXTERNAL BUS INTERFACE UNIT

EBIU Overview 6-1
Block Diagram 6-4
Internal Memory Interfaces 6-5
Registers 6-6
Shared Pins 6-6
System Clock 6-7
Error Detection 6-7
Bus Request and Grant 6-8
Operation 6-8
AMC Overview and Features 6-9
Features 6-9
Asynchronous Memory Interface 6-9
Asynchronous Memory Address Decode 6-10
AMC Pin Description 6-10
AMC Description of Operation 6-11
Avoiding Bus Contention 6-11
External Access Extension 6-12
AMC Functional Description 6-12
Programmable Timing Characteristics 6-12
Asynchronous Reads 6-13
Asynchronous Writes 6-14
Adding External Access Extension 6-16
Byte Enables 6-18

AMC Programming Model 6-18
AMC Registers 6-21
EBIU_AMGCTL Register 6-21
EBIU_AMBCTL0 and EBIU_AMBCTL1 Registers 6-21
AMC Programming Examples 6-24
SDC Overview and Features 6-25
Features 6-25
SDRAM Configurations Supported 6-26
SDRAM External Bank Size 6-27
SDC Address Mapping 6-27
Internal SDRAM Bank Select 6-29
Parallel Connection of SDRAMs 6-29
SDC Interface Overview 6-30
SDC Pin Description 6-30
SDRAM Performance 6-31
SDC Description of Operation 6-32
Definition of SDRAM Architecture Terms 6-32
Refresh 6-32
Row Activation 6-32
Column Read/Write 6-32
Row Precharge 6-33
Internal Bank 6-33
External Bank 6-33
Memory Size 6-33

Burst Length 6-33
Burst Type 6-34
CAS Latency 6-34
Data I/O Mask Function 6-34
SDRAM Commands 6-34
Mode Register Set (MRS) command 6-34
Extended Mode Register Set (EMRS) command 6-34
Bank Activate command 6-35
Read/Write command 6-35
Precharge/Precharge All Command 6-35
Auto-refresh command 6-35
Enter Self-Refresh Mode 6-35
Exit Self-Refresh Mode 6-36
SDC Timing Specs 6-36
tMRD 6-36
tRAS 6-36
CL 6-37
tRCD 6-37
tRRD 6-37
tWR 6-37
tRP 6-38
tRC 6-38
tRFC 6-38

tXSR 6-38
tREF 6-39
tREFI 6-39
SDC Functional Description 6-39
SDC Operation 6-39
SDC Address Muxing 6-42
Multibank Operation 6-43
Core and DMA Arbitration 6-44
Changing System Clock During Runtime 6-44
Changing Power Management During Runtime 6-46
Deep Sleep Mode 6-46
Hibernate State 6-46
Shared SDRAM 6-46
SDC Commands 6-47
Mode Register Set Command 6-49
Extended Mode Register Set Command (Mobile SDRAM) 6-50
Bank Activation Command 6-51
Read/Write Command 6-51
Write Command With Data Mask 6-51
Single Precharge Command 6-52
Precharge All Command 6-52
Auto-Refresh Command 6-52

Self-Refresh Mode 6-53
Self-Refresh Entry Command 6-53
Self-Refresh Exit Command 6-54
No Operation Command 6-54
SDC SA10 Pin 6-55
SDC Programming Model 6-55
SDC Configuration 6-56
Example SDRAM System Block Diagrams 6-58
SDC Register Definitions 6-60
EBIU_SDRRC Register 6-60
EBIU_SDBCTL Register 6-62
Using SDRAMs With Systems Smaller than 16M byte 6-65
EBIU_SDGCTL Register 6-67
EBIU_SDSTAT Register 6-78
SDC Programming Examples 6-79

PARALLEL PERIPHERAL INTERFACE

Overview 7-1
Features 7-1
Interface Overview 7-2
Description of Operation 7-5
Functional Description 7-6
ITU-R 656 Modes 7-6
ITU-R 656 Background 7-6
ITU-R 656 Input Modes 7-10

Entire Field 7-10
Active Video Only 7-11
Vertical Blanking Interval (VBI) only 7-11
ITU-R 656 Output Mode 7-12
Frame Synchronization in ITU-R 656 Modes 7-12
General-Purpose PPI Modes 7-13
Data Input (RX) Modes 7-15
No Frame Syncs 7-16
1, 2, or 3 External Frame Syncs 7-16
2 or 3 Internal Frame Syncs 7-17
Data Output (TX) Modes 7-18
No Frame Syncs 7-18
1 or 2 External Frame Syncs 7-18
1, 2, or 3 Internal Frame Syncs 7-19
Frame Synchronization in GP Modes 7-20
Modes With Internal Frame Syncs 7-20
Modes With External Frame Syncs 7-21
Programming Model 7-23
DMA Operation 7-23
PPI Registers 7-26
PPI_CONTROL Register 7-26
PPI_STATUS Register 7-30
PPI_DELAY Register 7-34
PPI_COUNT Register 7-34
PPI_FRAME Register 7-35

Programming Examples 7-36
Data Transfer Scenarios 7-39

ETHERNET MAC

Overview 8-1
Features 8-1
Interface Overview 8-2
External Interface 8-4
Clocking 8-4
Pins 8-5
Internal Interface 8-6
Power Management 8-7
Description of Operation 8-7
Protocol 8-7
MII Management Interface 8-7
Operation 8-10
MII Management Interface Operation 8-10
Receive DMA Operation 8-11
Frame Reception and Filtering 8-13
RX Automatic Pad Stripping 8-17
RX DMA Data Alignment 8-18
RX DMA Buffer Structure 8-18
RX Frame Status Buffer 8-19
RX Frame Status Classification 8-20

RX IP Frame Checksum Calculation 8-21
RX DMA Direction Errors 8-22
Transmit DMA Operation 8-24
Flexible Descriptor Structure 8-27
TX DMA Data Alignment 8-27
Late Collisions 8-28
TX Frame Status Classification 8-29
TX DMA Direction Errors 8-30
Power Management 8-31
Ethernet Operation in the Sleep State 8-33
Magic Packet Detection 8-34
Remote Wake-up Filters 8-35
Ethernet Event Interrupts 8-38
RX/TX Frame Status Interrupt Operation 8-42
RX Frame Status Register Operation at Startup and Shutdown 8-43
TX Frame Status Register Operation at Startup and Shutdown 8-43
MAC Management Counters 8-43
Programming Model 8-46
Configure MAC Pins 8-46
Multiplexing Scheme 8-47
CLKBUF 8-47
Configure Interrupts 8-47

Configure MAC Registers 8-48
MAC Address 8-48
MII Station Management 8-48
Configure PHY 8-50
Receive and Transmit Data 8-50
Receiving Data 8-51
Transmitting Data 8-51
Ethernet MAC Register Definitions 8-51
Control-Status Register Group 8-63
EMAC_OPMODE Register 8-64
EMAC_ADDRLO Register 8-70
EMAC_ADDRHI Register 8-71
EMAC_HASHLO and EMAC_HASHHI Registers 8-72
EMAC_STAADD Register 8-76
EMAC_STADAT Register 8-78
EMAC_FLC Register 8-78
EMAC_VLAN1 and EMAC_VLAN2 Registers 8-80
EMAC_WKUP_CTL Register 8-81
EMAC_WKUP_FFMSK0, EMAC_WKUP_FFMSK1, EMAC_WKUP_FFMSK2, and EMAC_WKUP_FFMSK3 Registers 8-84
EMAC_WKUP_FFCMD Register 8-89
EMAC_WKUP_FFOFF Register 8-91
EMAC_WKUP_FFCRC0 and EMAC_WKUP_FFCRC1 Registers 8-92

System Interface Register Group 8-93
EMAC_SYSCTL Register 8-93
EMAC_SYSTAT Register 8-95
Ethernet MAC Frame Status Registers 8-97
EMAC_RX_STAT Register 8-97
EMAC_RX_STKY Register 8-103
EMAC_RX_IRQE Register 8-107
EMAC_TX_STAT Register 8-108
EMAC_TX_STKY Register 8-112
EMAC_TX_IRQE Register 8-114
EMAC_MMC_RIRQS Register 8-115
EMAC_MMC_RIRQE Register 8-117
EMAC_MMC_TIRQS Register 8-119
EMAC_MMC_TIRQE Register 8-121
MAC Management Counter Registers 8-123
EMAC_MMC_CTL Register 8-124
Programming Examples 8-125
Ethernet Structures 8-126
MAC Address Setup 8-129
PHY Control Routines 8-130

CAN MODULE

Overview 9-1
Interface Overview 9-2
CAN Mailbox Area 9-4

CAN Mailbox Control 9-6
CAN Protocol Basics 9-7
CAN Operation 9-9
Bit Timing 9-9
Transmit Operation 9-12
Retransmission 9-13
Single Shot Transmission 9-15
Auto-Transmission 9-15
Receive Operation 9-15
Data Acceptance Filter 9-18
Remote Frame Handling 9-19
Watchdog Mode 9-19
Time Stamps 9-20
Temporarily Disabling Mailboxes 9-21
Functional Operation 9-22
CAN Interrupts 9-23
Mailbox Interrupts 9-23
Global Interrupt 9-24
Event Counter 9-26
CAN Warnings and Errors 9-28
Programmable Warning Limits 9-28
CAN Error Handling 9-28
Error Frames 9-29
Error Levels 9-31

Debug and Test Modes 9-33
Low Power Features 9-37
CAN Built-In Suspend Mode 9-37
CAN Built-In Sleep Mode 9-38
CAN Wakeup From Hibernate State 9-38
Register Definitions 9-39
Global Registers 9-42
CAN_CONTROL Register 9-43
CAN_STATUS Register 9-44
CAN_DEBUG Register 9-45
CAN_CLOCK Register 9-45
CAN_TIMING Register 9-46
CAN_INTR Register 9-46
CAN_GIM Register 9-47
CAN_GIS Register 9-47
CAN_GIF Register 9-48
Mailbox/Mask Registers 9-48
CAN_AMxx Registers 9-49
CAN_MBxx_ID1 Registers 9-53
CAN_MBxx_ID0 Registers 9-55
CAN_MBxx_TIMESTAMP Registers 9-57
CAN_MBxx_LENGTH Registers 9-59
CAN_MBxx_DATAx Registers 9-61

Mailbox Control Registers 9-68
CAN_MCx Registers 9-69
CAN_MDx Registers 9-70
CAN_RMPx Register 9-71
CAN_RMLx Register 9-72
CAN_OPSSx Register 9-73
CAN_TRSx Registers 9-74
CAN_TRRx Registers 9-75
CAN_AAx Register 9-76
CAN_TAx Register 9-77
CAN_MBTD Register 9-78
CAN_RFHx Registers 9-78
CAN_MBIMx Registers 9-79
CAN_MBTIFx Registers 9-80
CAN_MBRIFx Registers 9-81
Universal Counter Registers 9-82
CAN_UCCNF Register 9-83
CAN_UCCNT Register 9-84
CAN_UCRC Register 9-84
Error Registers 9-85
CAN_CEC Register 9-85
CAN_ESR Register 9-85
CAN_EWR Register 9-86

Programming Examples 9-87
CAN Setup Code 9-87
Initializing and Enabling CAN Mailboxes 9-89
Initiating CAN Transfers and Processing Interrupts 9-90

SPI COMPATIBLE PORT CONTROLLERS

Overview 10-1
Features 10-1
Interface Overview 10-3
External Interface 10-4
Serial Peripheral Interface Clock Signal (SCK) 10-4
Master Out Slave In (MOSI) 10-5
Master In Slave Out (MISO) 10-5
Serial Peripheral Interface Slave Select Input Signal 10-6
Serial Peripheral Interface Slave Select Enable Output Signals 10-7
Slave Select Inputs 10-10
Use of FLS Bits in SPI_FLG for Multiple Slave SPI Systems 10-10
Internal Interfaces 10-12
DMA Functionality 10-12
SPI Transmit Data Buffer 10-13
SPI Receive Data Buffer 10-14
Description of Operation 10-14
SPI Transfer Protocols 10-14

SPI General Operation 10-17
SPI Control 10-18
Clock Signals 10-19
SPI Baud Rate 10-20
Error Signals and Flags 10-20
Mode Fault Error (MODF) 10-21
Transmission Error (TXE) 10-22
Reception Error (RBSY) 10-22
Transmit Collision Error (TXCOL) 10-22
Interrupt Output 10-23
Functional Description 10-23
Master Mode Operation 10-24
Transfer Initiation From Master (Transfer Modes) 10-25
Slave Mode Operation 10-26
Slave Ready for a Transfer 10-27
Programming Model 10-28
Beginning and Ending an SPI Transfer 10-28
Master Mode DMA Operation 10-30
Slave Mode DMA Operation 10-32
SPI Registers 10-40
Programming Examples 10-45
Core Generated Transfer 10-45
Initialization Sequence 10-45
Starting a Transfer 10-46

Post Transfer and Next Transfer 10-47
Stopping 10-48
DMA Transfer 10-48
DMA Initialization Sequence 10-49
SPI Initialization Sequence 10-50
Starting a Transfer 10-51
Stopping a Transfer 10-51

TWO WIRE INTERFACE CONTROLLER

Overview 11-1
Interface Overview 11-2
External Interface 11-3
Serial Clock signal (SCL) 11-3
Serial data signal (SDA) 11-4
TWI Pins 11-4
Internal Interfaces 11-5
Description of Operation 11-6
TWI Transfer Protocols 11-6
Clock Generation and Synchronization 11-6
Bus Arbitration 11-7
Start and Stop Conditions 11-8
General Call Support 11-9
Fast Mode 11-10
TWI General Operation 11-10
TWI Control 11-10

Clock Signal 11-11
Error Signals and Flags 11-12
TWI Master Status 11-12
TWI Slave Status 11-15
TWI FIFO Status 11-16
TWI Interrupt Status 11-17
Functional Description 11-20
General Setup 11-20
Slave Mode 11-20
Master Mode Clock Setup 11-22
Master Mode Transmit 11-22
Master Mode Receive 11-23
Repeated Start Condition 11-24
Transmit/Receive Repeated Start Sequence 11-24
Receive/Transmit Repeated Start Sequence 11-26
Programming Model 11-28
Register Descriptions 11-30
TWI_CONTROL Register 11-30
TWI_CLKDIV Register 11-30
TWI_SLAVE_CTL Register 11-31
TWI_SLAVE_ADDR Register 11-33
TWI_SLAVE_STAT Register 11-33
TWI_MASTER_CTL Register 11-34
TWI_MASTER_ADDR Register 11-37
TWI_MASTER_STAT Register 11-38

TWI_FIFO_CTL Register 11-38
TWI_FIFO_STAT Register 11-40
TWI_INT_MASK Register 11-40
TWI_INT_STAT Register 11-43
TWI_XMT_DATA8 Register 11-43
TWI_XMT_DATA16 Register 11-44
TWI_RCV_DATA8 Register 11-45
TWI_RCV_DATA16 Register 11-46
Programming Examples 11-47
Master Mode Setup 11-47
Slave Mode Setup 11-52
Electrical Specifications 11-59

SPORT CONTROLLERS

Overview 12-1
Features 12-2
Interface Overview 12-3
SPORT Pin/Line Terminations 12-9
Description of Operation 12-10
SPORT Operation 12-10
SPORT Disable 12-10
Setting SPORT Modes 12-11
Stereo Serial Operation 12-12
Multichannel Operation 12-15
Multichannel Enable 12-18

Frame Syncs in Multichannel Mode 12-19
The Multichannel Frame 12-20
Multichannel Frame Delay 12-21
Window Size 12-21
Window Offset 12-22
Other Multichannel Fields in SPORTx_MCMC2 12-22
Channel Selection Register 12-23
Multichannel DMA Data Packing 12-24
Support for H.100 Standard Protocol 12-25
2X Clock Recovery Control 12-26
Functional Description 12-26
Clock and Frame Sync Frequencies 12-26
Maximum Clock Rate Restrictions 12-28
Word Length 12-28
Bit Order 12-28
Data Type 12-28
Companding 12-29
Clock Signal Options 12-30
Frame Sync Options 12-30
Framed Versus Unframed 12-31
Internal Versus External Frame Syncs 12-32
Active Low Versus Active High Frame Syncs 12-33
Sampling Edge for Data and Frame Syncs 12-33
Early Versus Late Frame Syncs (Normal Versus Alternate Timing) 12-35

Data Independent Transmit Frame Sync 12-37
Moving Data Between SPORTs and Memory 12-38
SPORT RX, TX, and Error Interrupts 12-38
PAB Errors 12-39
Timing Examples 12-39
SPORT Registers 12-45
Register Writes and Effective Latency 12-47
SPORTx_TCR1 and SPORTx_TCR2 Registers 12-47
SPORTx_RCR1 and SPORTx_RCR2 Registers 12-52
Data Word Formats 12-57
SPORTx_TX Register 12-58
SPORTx_RX Register 12-60
SPORTx_STAT Register 12-63
SPORTx_TCLKDIV and SPORTx_RCLKDIV Registers 12-64
SPORTx_TFSDIV and SPORTx_RFSDIV Register 12-65
SPORTx_MCMCn Registers 12-66
SPORTx_CHNL Register 12-67
SPORTx_MRCSn Registers 12-68
SPORTx_MTCSn Registers 12-70
Programming Examples 12-72
SPORT Initialization Sequence 12-73
DMA Initialization Sequence 12-75
Interrupt Servicing 12-77
Starting a Transfer 12-78

UART PORT CONTROLLERS

Overview 13-1
Features 13-1
Interface Overview 13-2
External Interface 13-2
Internal Interface 13-3
Description of Operation 13-4
UART Transfer Protocol 13-4
UART Transmit Operation 13-5
UART Receive Operation 13-6
IrDA Transmit Operation 13-8
IrDA Receive Operation 13-9
Interrupt Processing 13-10
Bit Rate Generation 13-12
Autobaud Detection 13-13
Programming Model 13-15
Non-DMA Mode 13-15
DMA Mode 13-17
Mixing Modes 13-18
UART Registers 13-19
UARTx_LCR Registers 13-21
UARTx_MCR Registers 13-23
UARTx_LSR Registers 13-24
UARTx_THR Registers 13-26

UARTx_RBR Registers 13-26
UARTx_IER Registers 13-27
UARTx_IIR Registers 13-29
UARTx_DLL and UARTx_DLH Registers 13-30
UARTx_SCR Registers 13-31
UARTx_GCTL Registers 13-31
Programming Examples 13-32

GENERAL-PURPOSE PORTS

Overview 14-1
Features 14-2
Interface Overview 14-3
External Interface 14-4
Port F Structure 14-4
Port G Structure 14-5
Port H Structure 14-6
Port J Structure 14-7
Internal Interfaces 14-8
Performance/Throughput 14-9
Description of Operation 14-9
Operation 14-9
General-Purpose I/O Modules 14-10
GPIO Interrupt Processing 14-14
Programming Model 14-20
Memory-Mapped GPIO Registers 14-22

Programming Examples 14-34

GENERAL-PURPOSE TIMERS

Overview and Features 15-1
Features 15-1
Interface Overview 15-2
External Interface 15-4
Internal Interface 15-6
Description of Operation 15-6
Interrupt Processing 15-8
Illegal States 15-10
Modes of Operation 15-13
Pulse Width Modulation (PWM_OUT) Mode 15-13
Output Pad Disable 15-15
Single Pulse Generation 15-15
Pulse Width Modulation Waveform Generation 15-16
PULSE_HI Toggle Mode 15-18
Externally Clocked PWM_OUT 15-22
Using PWM_OUT Mode With the PPI 15-23
Stopping the Timer in PWM_OUT Mode 15-23
Pulse Width Count and Capture (WDTH_CAP) Mode 15-26
Autobaud Mode 15-34
External Event (EXT_CLK) Mode 15-34
Programming Model 15-36

Timer Registers 15-37
TIMER_ENABLE Register 15-38
TIMER_DISABLE Register 15-39
TIMER_STATUS Register 15-40
TIMERx_CONFIG Registers 15-43
TIMERx_COUNTER Registers 15-45
TIMERx_PERIOD and TIMERx_WIDTH Registers 15-46
Summary 15-50
Programming Examples 15-52

CORE TIMER

Overview and Features 16-1
Timer Overview 16-1
External Interfaces 16-2
Internal Interfaces 16-2
Description of Operation 16-2
Interrupt Processing 16-3
Core Timer Registers 16-4
TCNTL Register 16-4
TCOUNT Register 16-5
TPERIOD Register 16-6
TSCALE Register 16-7
Programming Examples 16-7

WATCHDOG TIMER

Overview and Features 17-1
Interface Overview 17-2
External Interface 17-3
Internal Interface 17-3
Description of Operation 17-3
Register Definitions 17-5
WDOG_CNT Register 17-5
WDOG_STAT Register 17-6
WDOG_CTL Register 17-7
Programming Examples 17-9

REAL-TIME CLOCK

Overview 18-1
Interface Overview 18-2
Description of Operation 18-3
RTC Clock Requirements 18-4
Prescaler Enable 18-4
RTC Programming Model 18-6
Register Writes 18-7
Write Latency 18-8
Register Reads 18-9
Deep Sleep 18-9
Event Flags 18-10

Setting Time of Day 18-12
Using the Stopwatch 18-13
Interrupts 18-14
State Transitions Summary 18-16
Register Definitions 18-19
RTC_STAT Register 18-20
RTC_ICTL Register 18-20
RTC_ISTAT Register 18-21
RTC_SWCNT Register 18-21
RTC_ALARM Register 18-22
RTC_PREN Register 18-22
Programming Examples 18-23
Enable RTC Prescaler 18-23
RTC Stopwatch For Exiting Deep Sleep Mode 18-24
RTC Alarm to Come Out of Hibernate State 18-26

SYSTEM RESET AND BOOTING

Reset and Powerup 19-2
Hardware Reset 19-2
System Reset Configuration Register (SYSCR) 19-4
Software Resets and Watchdog Timer 19-5
Software Reset Register (SWRST) 19-6
Core-Only Software Reset 19-7
Core and System Reset 19-8
Reset Vector 19-8

Servicing Reset Interrupts 19-9
Booting Process 19-11
Header Information 19-13
Host Wait Feedback Strobe (HWAIT) 19-19
Final Initialization 19-21
Initialization Code 19-22
Multi-Application (Multi-DXE) Management 19-26
User-callable Boot ROM Functions 19-27
Booting a Different Application 19-27
Determining Boot Stream Start Addresses 19-29
Specific Blackfin Boot Modes 19-34
Bypass (No-Boot) Mode (BMODE = 000) 19-34
8-Bit Flash/PROM Boot (BMODE = 001) 19-36
16-Bit Flash/PROM Boot (BMODE = 001) 19-40
SPI Master Mode Boot from SPI Memory (BMODE = 011) 19-42
SPI Memory Detection Routine 19-44
SPI Slave Mode Boot From SPI Host (BMODE = 100) 19-48
TWI Master Boot Mode (BMODE = 101) 19-53
TWI Slave Boot Mode (BMODE = 110) 19-55
UART Slave Mode Boot via Master Host (BMODE = 111) 19-56
Blackfin Loader File Viewer 19-59

DYNAMIC POWER MANAGEMENT

Phase Locked Loop and Clock Control 20-1
PLL Overview 20-2
PLL Clock Multiplier Ratios 20-3
Core Clock/System Clock Ratio Control 20-5
Dynamic Power Management Controller 20-7
Operating Modes 20-8
Dynamic Power Management Controller States 20-8
Full On Mode 20-9
Active Mode 20-9
Sleep Mode 20-9
Deep Sleep Mode 20-10
Hibernate State 20-11
Operating Mode Transitions 20-11
Programming Operating Mode Transitions 20-14
PLL Programming Sequence 20-15
PLL Programming Sequence Continues 20-17
Dynamic Supply Voltage Control 20-18
Power Supply Management 20-18
Controlling the Voltage Regulator 20-19
Changing Voltage 20-21
Powering Down the Core (Hibernate State) 20-22
PLL Registers 20-24
PLL_DIV Register 20-25
PLL_CTL Register 20-26

PLL_STAT Register 20-26
PLL_LOCKCNT Register 20-27
VR_CTL Register 20-27
Programming Examples 20-28
Active Mode to Full On Mode 20-29
Full On Mode to Active Mode 20-30
In the Full On Mode, Change CLKIN to VCO Multiplier From 31x to 2x 20-31
Setting Wakeups and Entering Hibernate State 20-32
Changing Internal Voltage Levels 20-33

SYSTEM DESIGN

Pin Descriptions 21-1
Managing Clocks 21-1
Managing Core and System Clocks 21-2
Configuring and Servicing Interrupts 21-2
Semaphores 21-2
Example Code for Query Semaphore 21-3
Data Delays, Latencies and Throughput 21-4
Bus Priorities 21-4
External Memory Design Issues 21-5
Example Asynchronous Memory Interfaces 21-5
Avoiding Bus Contention 21-7
High Frequency Design Considerations 21-8
Signal Integrity 21-8
Decoupling Capacitors and Ground Planes 21-10

5 Volt Tolerance 21-11
Resetting the Processor 21-12
Recommendations for Unused Pins 21-12
Programmable Outputs 21-12
Test Point Access 21-12
Oscilloscope Probes 21-13
Recommended Reading 21-13

SYSTEM MMR ASSIGNMENTS

Dynamic Power Management Registers A-2
System Reset and Interrupt Control Registers A-2
Watchdog Timer Registers A-3
Real-Time Clock Registers A-4
UART0 Controller Registers A-4
SPI Controller Registers A-5
Timer Registers A-6
Ports Registers A-8
SPORT0 Controller Registers A-12
SPORT1 Controller Registers A-14
External Bus Interface Unit Registers A-16
DMA/Memory DMA Control Registers A-17
PPI Registers A-19
UART1 Controller Registers A-20
CAN Registers A-21
Ethernet MAC Registers A-29

Handshake MDMA Control Registers A-34
Core Timer Registers A-36
Processor-Specific Memory Registers A-36

TEST FEATURES

JTAG Standard B-1
Boundary-Scan Architecture B-2
Instruction Register B-4
Public Instructions B-5
EXTEST – Binary Code 00000 ..... B-5
SAMPLE/PRELOAD – Binary Code 10000 ..... B-6
BYPASS – Binary Code 11111 ..... B-6
Boundary-Scan Register B-6

GLOSSARY

INDEX

PREFACE

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

Purpose of This Manual

The *ADSP-BF537 Blackfin Processor Hardware Reference* provides architectural information about the ADSP-BF534, ADSP-BF536, and ADSP-BF537 processors. The architectural descriptions cover functional blocks, buses, and ports, including all features and processes that they support. For programming information, see the *ADSP-BF53x/BF56x Blackfin Processor Programming Reference.* For timing, electrical, and package specifications, see the *ADSP-BF534 Embedded Processor Data Sheet* or the *ADSP-BF536/ADSP-BF537 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 L1memories 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, 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, "Parallel Peripheral Interface"
  Describes the Parallel Peripheral Interface (PPI) of the processor. The PPI is a half-duplex, bidirectional port accommodating up to 16 bits of data and is used for digital video and data converter applications.

- Chapter 8, "Ethernet MAC"
  Describes the Ethernet Media Access Controller (MAC) peripheral that is available on ADSP-BF536 and ADSP-BF537 processors. 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.

- Chapter 9, "CAN Module"
  Describes the CAN module, a low bit rate serial interface intended for use in applications where bit rates are typically up to 1Mbit/s.

- Chapter 10, "SPI Compatible Port Controllers"
  Describes the Serial Peripheral Interface (SPI) port that provides an I/O interface to a variety of SPI compatible peripheral devices.

- Chapter 11, "Two Wire Interface Controller"
  Describes the Two Wire Interface (TWI) controller, which allows a device to interface to an Inter IC bus as specified by the *Philips I^2C Bus Specification version 2.1* dated January 2000.

- Chapter 12, "SPORT Controllers"
  Describes the two independent, synchronous Serial Port Controllers (SPORT0 and SPORT1) that provide an I/O interface to a variety of serial peripheral devices.

- Chapter 13, "UART Port Controllers"
  Describes the two Universal Asynchronous Receiver/Transmitter ports (UART0 and UART1) that convert data between serial and parallel formats. The UARTs support the half-duplex IrDA® SIR protocol as a mode-enabled feature.

- Chapter 14, "General-Purpose Ports"
  Describes the general-purpose I/O ports, including the structure of each port, multiplexing, configuring the pins, and generating interrupts.

- Chapter 15, "General-Purpose Timers"
  Describes the eight general-purpose timers.
- Chapter 16, "Core Timer"
  Describes the core timer.
- Chapter 17, "Watchdog Timer"
  Describes the watchdog timer.
- Chapter 18, "Real-Time Clock"
  Describes a set of digital watch features of the processor, including time of day, alarm, and stopwatch countdown.
- Chapter 19, "System Reset and Booting"
  Describes the booting methods, booting process and specific boot modes for the processor.
- Chapter 20, "Dynamic Power Management"
  Describes the clocking, including the PLL, and the dynamic power management controller.
- Chapter 21, "System Design"
  Describes how to use the processor as part of an overall system. It 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.

- Appendix B, "Test Features"
  Describes test features for the processor, discusses the JTAG standard, boundary-scan architecture, instruction and boundary registers, and public instructions.

- "Glossary"
  Contains definitions of terms used in this book, including acronyms.

This hardware reference is a companion document to the *ADSP-BF53x/BF56x Blackfin Processor Programming Reference*.

What's New in This Manual

This is Revision 2.0 of the *ADSP-BF537 Blackfin Processor Hardware Reference*. Peripheral chapters have been expanded and reorganized, and modifications and corrections based on errata reports against the Preliminary version of this manual have been made.

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-BF53x 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, and ADSP-2136x.

Product Information

You can obtain product information from the Analog Devices Web site, from the product CD-ROM, or from the printed publications (manuals).

Analog Devices is online at www.analog.com. Our Web site provides information about a broad range of products—analog integrated circuits, amplifiers, converters, and digital signal processors.

MyAnalog.com

MyAnalog.com is a free feature of the Analog Devices Web site that allows customization of a Web page to display only the latest information on products you are interested in. You can also choose to receive weekly e-mail notifications containing updates to the Web pages that meet your interests. MyAnalog.com provides access to books, application notes, data sheets, code examples, and more.

Registration

Visit www.myanalog.com to sign up. Click **Register** to use MyAnalog.com. Registration takes about five minutes and serves as a means to select the information you want to receive.

If you are already a registered user, just log on. Your user name is your e-mail address.

Processor Product Information

For information on embedded processors and DSPs, visit our Web site at www.analog.com/processors, which provides access to technical publications, data sheets, application notes, product overviews, and product announcements.

You may also obtain additional information about Analog Devices and its products in any of the following ways.

- E-mail questions or requests for information to
  `processor.support@analog.com (World wide support)`
  `processor.europe@analog.com (Europe support)`
  `processor.china@analog.com (China support)`
- Fax questions or requests for information to
  1-781-461-3010 (North America)
  +49-89-76903-157 (Europe)
- Access the FTP Web site at
  `ftp ftp.analog.com` (or `ftp 137.71.25.69`)
  `ftp://ftp.analog.com`

Related Documents

The following publications that describe the ADSP-BF534, ADSP-BF536, and ADSP-BF537 Blackfin processor (and related processors) can be ordered from any Analog Devices sales office:

- *ADSP-BF53x/ADSP-BF56x Blackfin Processor Programming Reference*
- *ADSP-BF536/ADSP-BF537 Blackfin Embedded Processor Data Sheet*
- *ADSP-BF534 Blackfin Embedded Processor Data Sheet*

For information on product related development software and Analog Devices processors, see these publications:

- *VisualDSP++ User's Guide for Blackfin Processors*
- *VisualDSP++ C/C++ Compiler and Library Manual for Blackfin Processors*

- *VisualDSP++ Assembler and Preprocessor Manual for Blackfin Processors*
- *VisualDSP++ Linker and Utilities Manual for Blackfin Processors*
- *VisualDSP++ Kernel (VDK) User's Guide*

Visit the Technical Library Web site to access all processor and tools manuals and data sheets:

http://www.analog.com/processors/technical_library

Online Technical Documentation

Online documentation comprises the VisualDSP++ Help system, software tools manuals, hardware tools manuals, processor manuals, the Dinkum Abridged C++ library, and Flexible License Manager (FlexLM) network license manager software documentation. You can easily search across the entire VisualDSP++ documentation set for any topic of interest. For easy printing, supplementary .PDF files of most manuals are also provided.

Each documentation file type is described as follows.

File	Description
.CHM	Help system files and manuals in Help format
.HTM or .HTML	Dinkum Abridged C++ library and FlexLM network license manager software documentation. Viewing and printing the .HTML files requires a browser, such as Internet Explorer 4.0 (or higher).
.PDF	VisualDSP++ and processor manuals in Portable Documentation Format (PDF). Viewing and printing the .PDF files requires a PDF reader, such as Adobe Acrobat Reader (4.0 or higher).

If documentation is not installed on your system as part of the software installation, you can add it from the VisualDSP++ CD-ROM at any time by running the Tools installation. Access the online documentation from the VisualDSP++ environment, Windows® Explorer, or the Analog Devices Web site.

Accessing Documentation From VisualDSP++

From the VisualDSP++ environment:

- Access VisualDSP++ online Help from the Help menu's **Contents**, **Search**, and **Index** commands.
- Open online Help from context-sensitive user interface items (toolbar buttons, menu commands, and windows).

Accessing Documentation From Windows

In addition to any shortcuts you may have constructed, there are many ways to open VisualDSP++ online Help or the supplementary documentation from Windows.

Help system files (`.CHM`) are located in the `Help` folder, and `.PDF` files are located in the `Docs` folder of your VisualDSP++ installation CD-ROM. The `Docs` folder also contains the Dinkum Abridged C++ library and the FlexLM network license manager software documentation.

Using Windows Explorer

- Double-click the `vdsp-help.chm` file, which is the master Help system, to access all the other `.CHM` files.
- Double-click any file that is part of the VisualDSP++ documentation set.

Using the Windows Start Button

- Access VisualDSP++ online Help by clicking the **Start** button and choosing **Programs**, **Analog Devices**, **VisualDSP++**, and **VisualDSP++ Documentation**.
- Access the `.PDF` files by clicking the **Start** button and choosing **Programs**, **Analog Devices**, **VisualDSP++**, **Documentation for Printing**, and the name of the book.

Accessing Documentation From the Web

Download manuals at the following Web site:
http://www.analog.com/processors/technical_library

Select a processor family and book title. Download archive (`.ZIP`) files, one for each manual. Use any archive management software, such as WinZip, to decompress downloaded files.

Printed Manuals

For general questions regarding literature ordering, call the Literature Center at **1-800-ANALOGD** (**1-800-262-5643**) and follow the prompts.

VisualDSP++ Documentation Set

To purchase VisualDSP++ manuals, call **1-603-883-2430**. The manuals may be purchased only as a kit.

If you do not have an account with Analog Devices, you are referred to Analog Devices distributors. For information on our distributors, log onto http://www.analog.com/salesdir.

Hardware Tools Manuals

To purchase EZ-KIT Lite® and In-Circuit Emulator (ICE) manuals, call 1-603-883-2430. The manuals may be ordered by title or by product number located on the back cover of each manual.

Processor Manuals

Hardware reference and instruction set reference manuals may be ordered through the Literature Center at **1-800-ANALOGD** (**1-800-262-5643**), or downloaded from the Analog Devices Web site. Manuals may be ordered by title or by product number located on the back cover of each manual.

Data Sheets

All data sheets (preliminary and production) may be downloaded from the Analog Devices Web site. Only production (final) data sheets (Rev. 0, A, B, C, and so on) can be obtained from the Literature Center at **1-800-ANALOGD** (**1-800-262-5643**); they also can be downloaded from the Web site.

To have a data sheet faxed to you, call the Analog Devices Faxback System at **1-800-446-6212**. Follow the prompts and a list of data sheet code numbers will be faxed to you. If the data sheet you want is not listed, check for it on the Web site.

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 $\overline{\text{SMS3}}$, $\overline{\text{SMS2}}$, $\overline{\text{SMS1}}$, and $\overline{\text{SMS0}}$).
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 devices 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.

Table P-1. Short Form of Register Names

Pattern	Description	Examples
TIMERx_CONFIG	The *x* refers to multiple instances of the peripheral.	TIMER0_CONFIG TIMER1_CONFIG TIMER2_CONFIG
SIC_IARn	The *n* refers to multiple registers within the same peripheral or within the same core component.	SIC_IAR2 ICPLB_DATA15
SPORTx_TCRn	The combination of *x* and *n* indicates multiple instances of the peripheral *and* 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

- 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.

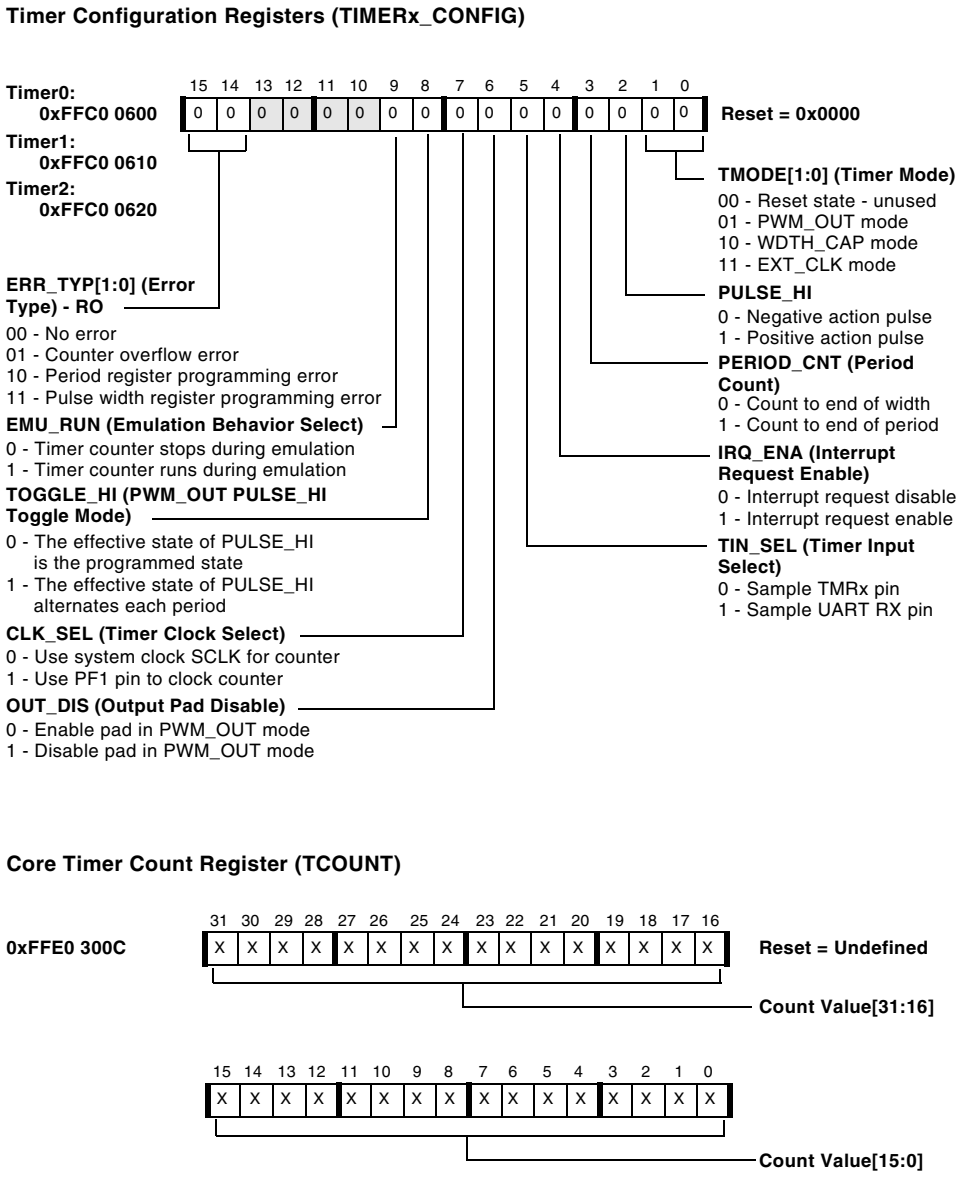


Figure P-1. Register Diagram Examples

1 INTRODUCTION

The ADSP-BF534, ADSP-BF536, and ADSP-BF537 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-BF536 and ADSP-BF537 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 ADSP-BF534 processor is pin compatible with the ADSP-BF536 and ADSP-BF537, but it does not include the embedded Ethernet controller like the ADSP-BF536 and ADSP-BF537 devices.

Peripherals

The processor system peripherals include:

- IEEE 802.3-compliant 10/100 Ethernet MAC (Not included on the ADSP-BF534)
- Controller Area Network (CAN) 2.0B interface
- Parallel Peripheral Interface (PPI), supporting ITU-R 656 video data formats
- Two dual-channel, full-duplex synchronous Serial Ports (SPORTs), supporting eight stereo I^2S channels
- 12 peripheral DMAs (2 mastered by the Ethernet MAC on ADSP-BF536 and ADSP-BF537 processors)

- Two memory-to-memory DMAs with handshake DMA
- Event handler with 32 interrupt inputs
- Serial Peripheral Interface (SPI)-compatible
- Two UARTs with IrDA® support
- Two-Wire Interface (TWI) controller
- Eight 32-bit timer/counters with PWM support
- Real-Time Clock (RTC) and watchdog timer
- 32-bit core timer
- 48 General-Purpose I/Os (GPIOs), 8 with high current drivers
- On-chip PLL capable of 1x to 63x frequency multiplication
- Debug/JTAG interface

These peripherals are connected to the core via several high bandwidth buses, as shown in Figure 1-1 for the ADSP-BF534 and in Figure 1-2 for the ADSP-BF536 and ADSP-BF537.

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 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.

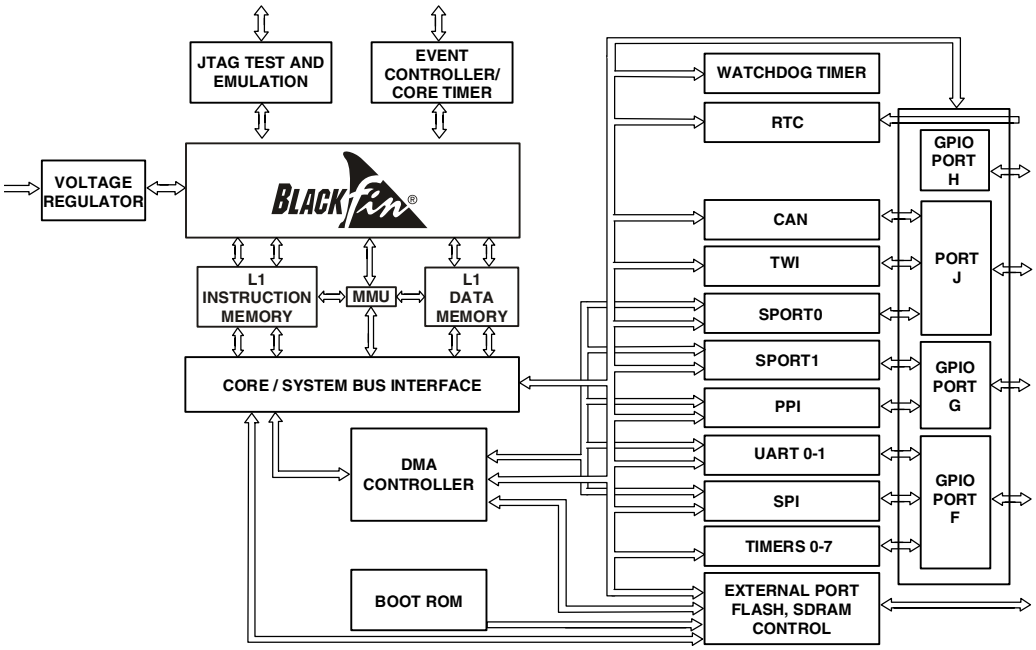


Figure 1-1. ADSP-BF534 Processor Block Diagram

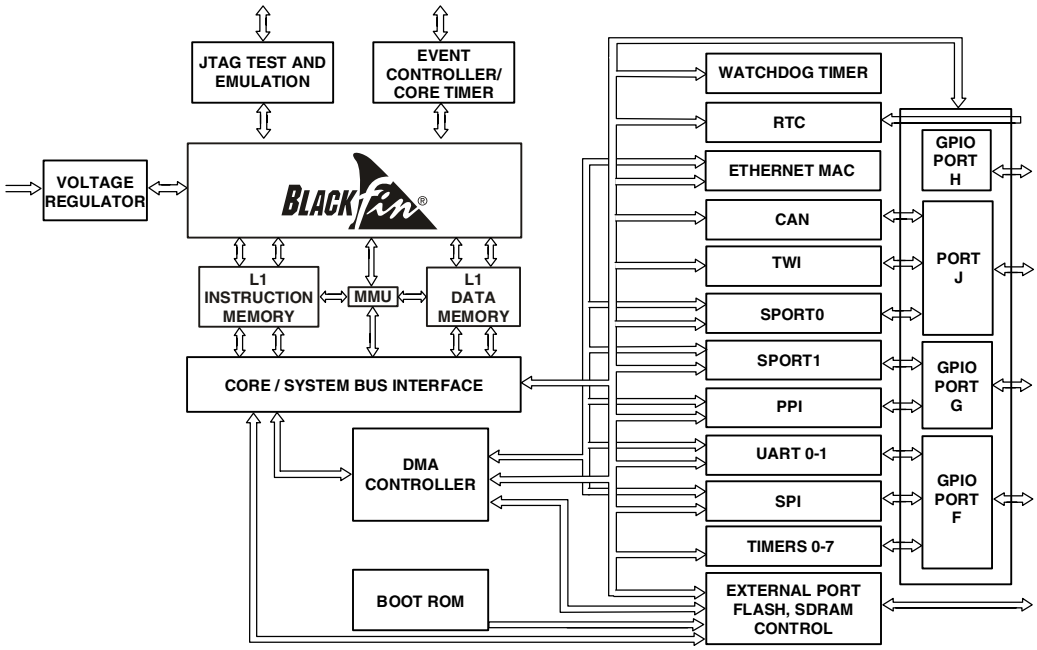


Figure 1-2. ADSP-BF536 and ADSP-BF537 Processor Block Diagram

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-BF534, ADSP-BF536, and ADSP-BF537 processors.

Table 1-1. Memory Configurations

Type of Memory	ADSP-BF534	ADSP-BF536	ADSP-BF537
Instruction SRAM/cache, lockable by way or line	16K byte	16K byte	16K byte
Instruction SRAM	48K byte	48K byte	48K byte
Data SRAM/cache	32K byte	16K byte	32K byte
Data SRAM	32K byte	16K byte	32K byte
Data scratchpad SRAM	4K byte	4K byte	4K byte
Total	**132K byte**	**100K byte**	**132K byte**

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

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

Internal Memory

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

- L1 instruction memory, consisting of SRAM and a 4-way set-associative cache. This memory is accessed at full processor speed.
- L1 data memory, consisting of SRAM and/or a 2-way set-associative cache. This memory block is accessed at full processor speed.
- L1 scratchpad RAM, which runs at the same speed as the L1 memories but is only accessible as data SRAM and cannot be configured as cache memory.

External Memory

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

The PC133-compliant SDRAM controller can be programmed to interface to up to 512M bytes of SDRAM.

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

I/O Memory Space

Blackfin 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.

DMA Support

The processor has multiple, independent DMA controllers that support automated data transfers with minimal overhead for the core. DMA transfers can occur between the internal memories and any of its DMA-capable peripherals. Additionally, DMA transfers can be accomplished between any of the DMA-capable peripherals and external devices connected to the external memory interfaces, including the SDRAM controller and the asynchronous memory controller. DMA-capable peripherals include the SPORTs, SPI ports, UARTs, and PPI. For the ADSP-BF536 and ADSP-BF537 processors, Ethernet is also a DMA-capable peripheral. Each individual DMA-capable peripheral has at least one dedicated DMA channel.

The DMA controller supports 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 +/- 32K elements. Furthermore, the column step size can be less

than the row step size, allowing implementation of interleaved data-streams. This feature is especially useful in video applications where data can be de-interleaved on the fly.

Examples of DMA types supported include:

- A single, linear buffer that stops upon completion
- A circular, auto-refreshing buffer that interrupts on each full or fractionally full buffer
- 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, there is a separate memory DMA channel provided for transfers between the various memories of the system. This enables transfers of blocks of data between any of the memories—including external SDRAM, ROM, SRAM, and flash memory—with minimal processor intervention. Memory DMA transfers can be controlled by a very flexible descriptor-based methodology or by a standard register-based autobuffer mechanism.

The ADSP-BF534, ADSP-BF536, and ADSP-BF537 processors also include a handshake DMA capability via dual external DMA request pins when used in conjunction with the External Bus Interface Unit (EBIU). This functionality can be used when a high speed interface is required for external FIFOs and high bandwidth communications peripherals such as USB 2.0. It allows control of the number of data transfers for MDMA. The number of transfers per edge is programmable. This feature can be programmed to allow MDMA to have an increased priority on the external bus relative to the core.

External Bus Interface Unit

The External Bus Interface Unit (EBIU) on the processor interfaces with a wide variety of industry-standard memory devices. The controller consists of an SDRAM controller and an asynchronous memory controller.

PC133 SDRAM Controller

The SDRAM controller provides an interface to a single bank of industry-standard SDRAM devices or DIMMs. Fully compliant with the PC133 SDRAM standard, the bank can be configured to contain between 16M and 512M bytes of memory.

A set of programmable timing parameters is available to configure the SDRAM bank to support slower memory devices. The memory bank is 16 bits wide for minimum device count and lower system cost.

Asynchronous Controller

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

Ports

Because of the rich set of peripherals, the ADSP-BF534, ADSP-BF536, and ADSP-BF537 processor groups the many peripheral signals to four ports—port F, port G, port H, and port J. Most of the associated pins are shared by multiple signals. The ports function as multiplexer controls. Eight of the pins (port F7–0) offer high source/high sink current capabilities.

General-Purpose I/O (GPIO)

The ADSP-BF534, ADSP-BF536, and ADSP-BF537 processor has 48 bi-directional, general-purpose I/O (GPIO) pins allocated across three separate GPIO modules—PORTFIO, PORTGIO, and PORTHIO, associated with port F, port G, and port H, respectively. Port J does not provide GPIO functionality. Each GPIO-capable pin shares functionality with other ADSP-BF534, ADSP-BF536, and ADSP-BF537 processor peripherals via a multiplexing scheme; however, the GPIO functionality is the default state of the device upon powerup. Neither GPIO output or input drivers are active by default. Each general-purpose port pin can be individually controlled by manipulation of the port control, status, and interrupt registers:

- GPIO direction control register – Specifies the direction of each individual GPIO pin as input or output.
- GPIO control and status registers – The ADSP-BF534, ADSP-BF536, and ADSP-BF537 processor employs a “write one to modify” mechanism that allows any combination of individual GPIO pins to be modified in a single instruction, without affecting the level of any other GPIO pins. Four control registers are provided. One register is written in order to set pin values, one register is written in order to clear pin values, one register is written in

order to toggle pin values, and one register is written in order to specify a pin value. Reading the GPIO status register allows software to interrogate the sense of the pins.

- GPIO interrupt mask registers – The two GPIO interrupt mask registers allow each individual GPIO pin to function as an interrupt to the processor. Similar to the two GPIO control registers that are used to set and clear individual pin values, one GPIO interrupt mask register sets bits to enable interrupt function, and the other GPIO interrupt mask register clears bits to disable interrupt function. GPIO pins defined as inputs can be configured to generate hardware interrupts, while output pins can be triggered by software interrupts.
- GPIO interrupt sensitivity registers – The two GPIO interrupt sensitivity registers specify whether individual pins are level- or edge-sensitive and specify—if edge-sensitive—whether just the rising edge or both the rising and falling edges of the signal are significant. One register selects the type of sensitivity, and one register selects which edges are significant for edge-sensitivity.

Two-Wire Interface

The Two-Wire Interface (TWI) is fully compatible with the widely used I^2C bus standard. It was designed with a high level of functionality and is compatible with multi-master, multi-slave 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 I^2C bus protocol. The *Philips I^2C Bus Specification version 2.1* covers many variants of I^2C. The TWI controller includes these features:

- Simultaneous master and slave operation on multiple device systems
- Support for multi-master data arbitration
- 7-bit addressing
- 100 kbits/second and 400 kbits/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 lock-up
- 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 Controller Area Network (CAN) module is a low bit rate serial interface intended for use in applications where bit rates are typically up to 1 Mbit/second. The CAN protocol incorporates a data CRC check, message error tracking, and fault node confinement as means to improve network reliability to the level required for control applications.

The interface to the CAN bus is a simple two-wire line. See Figure 9-1 on page 9-2 for a symbolic representation of the CAN transceiver interconnection. The Blackfin processor's CANTX output and CANRX input pins are connected to an external CAN transceiver's TX and RX pins, respectively.

Key features of the CAN module are:

- Conforms to the CAN 2.0B (active) standard
- Supports both standard (11-bit) and extended (29-bit) identifiers
- Supports data rates of up to 1 Mbit/second
- 32 mailboxes (8 transmit, 8 receive, 16 configurable)
- Dedicated acceptance mask for each mailbox
- Data filtering (first 2 bytes) can be used for acceptance filtering (Device Net mode)
- Error status and warning registers
- Transmit priority by identifier
- Universal counter module
- Readable receive and transmit pin values

These modes support ADC/DAC connections, as well as video communication with hardware signalling. Many of the modes support more than one level of frame synchronization. If desired, a programmable delay can be inserted between assertion of a frame sync and reception/transmission of data.

Ethernet MAC

The Ethernet Media Access Controller (MAC) peripheral for the ADSP-BF536 and ADSP-BF537 processors provides a 10/100 Mbit/second Ethernet interface, compliant with IEEE Std. 802.3-2002, between a Media Independent Interface (MII) and the Blackfin peripheral subsystem. The MAC operates in both half-duplex and full-duplex modes. It provides programmable enhanced features designed to minimize bus utilization and pre- or post-message processing. The connection to the external physical layer device (PHY) is achieved via the MII or a Reduced Media Independent Interface (RMII). The RMII provides data buses half as wide (2 bit vs. 4 bit) as those of an MII, operating at double the frequency.

The MAC is clocked internally from the `CLKIN` pin on the processor. A buffered version of this clock can also be used to drive the external PHY via the `CLKBUF` pin. A 25 MHz source should be used with an MII PHY. A 50 MHz clock source is required to drive an RMII PHY.

Parallel Peripheral Interface

The processor provides a Parallel Peripheral Interface (PPI) that can connect directly to parallel A/D and D/A converters, ITU-R 601/656 video encoders and decoders, and other general-purpose peripherals. The PPI consists of a dedicated input clock pin and three multiplexed frame sync pins. The input clock supports parallel data rates up to half the system clock rate.

In ITU-R 656 modes, the PPI receives and parses a data stream of 8-bit or 10-bit data elements. On-chip decode of embedded preamble control and synchronization information is supported.

Three distinct ITU-R 656 modes are supported:

- Active video only - The PPI does not read in any data between the End of Active Video (EAV) and Start of Active Video (SAV) preamble symbols, or any data present during the vertical blanking intervals. In this mode, the control byte sequences are not stored to memory; they are filtered by the PPI.
- Vertical blanking only - The PPI only transfers Vertical Blanking Interval (VBI) data, as well as horizontal blanking information and control byte sequences on VBI lines.
- Entire field - The entire incoming bitstream is read in through the PPI. This includes active video, control preamble sequences, and ancillary data that may be embedded in horizontal and vertical blanking intervals.

Though not explicitly supported, ITU-R 656 output functionality can be achieved by setting up the entire frame structure (including active video, blanking, and control information) in memory and streaming the data out the PPI in a frame sync-less mode. The processor's 2D DMA features facilitate this transfer by allowing the static frame buffer (blanking and control codes) to be placed in memory once, and simply updating the active video information on a per-frame basis.

The general-purpose modes of the PPI are intended to suit a wide variety of data capture and transmission applications. The modes are divided into four main categories, each allowing up to 16 bits of data transfer per `PPI_CLK` cycle:

- Data receive with internally generated frame syncs
- Data receive with externally generated frame syncs
- Data transmit with internally generated frame syncs
- Data transmit with externally generated frame syncs

These modes support ADC/DAC connections, as well as video communication with hardware signalling. Many of the modes support more than one level of frame synchronization. If desired, a programmable delay can be inserted between assertion of a frame sync and reception/transmission of data.

SPORT Controllers

The processor incorporates two dual-channel synchronous serial ports (SPORT0 and SPORT1) for serial and multiprocessor communications. The SPORTs support these features:

- Bidirectional, I^2S capable operation

  Each SPORT has two sets of independent transmit and receive pins, which enable eight channels of I^2S 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.

- 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 processor has an SPI-compatible port that enables the processor to communicate with multiple SPI-compatible devices.

The SPI interface uses three pins for transferring data: two data pins and a clock pin. An SPI chip select input pin lets other SPI devices select the processor, 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, and it has an integrated DMA controller, configurable to support either transmit or receive datastreams. 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 nine general-purpose programmable timer units in the processor. Eight timers have an external pin that can be configured either as a Pulse Width Modulator (PWM) or timer output, as an input to clock the timer, or as a mechanism for measuring pulse widths of external events. These timer units can be synchronized to an external clock input connected to the `PF1` pin, an external clock input to the `PPI_CLK` pin, or to the internal `SCLK`.

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

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

In addition to the eight general-purpose programmable timers, a 9th timer is also provided. This extra timer is clocked by the internal processor clock and is typically used as a system tick clock for generation of operating system periodic interrupts.

UART Ports

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

- 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 UARTs' baud rate, serial data format, error code generation and status, and interrupts 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

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

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.

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.

The 32.768 kHz input clock frequency is divided down to a 1 Hz signal by a prescaler. The counter function of the timer consists of four counters: a 60 second counter, a 60 minute counter, a 24 hours counter, and a 32768 day counter.

When enabled, the alarm function generates an interrupt when the output of the timer matches the programmed value in the alarm control register. There are two alarms. The first alarm is for a time of day. The second alarm is for a day and time of that day.

The stopwatch function counts down from a programmed value, with one minute resolution. When the stopwatch is enabled and the counter 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 CPU and the peripherals. After a reset, software can determine if the watchdog was the source of the hardware reset by interrogating a status bit in the watchdog control register.

The timer is clocked by the system clock (SCLK), at a maximum frequency of f_{SCLK}.

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 (1x to 63x) multiplication factor (bounded by specified minimum and maximum VCO frequencies). The default multiplier is 10x, but it can be modified by a software instruction sequence. On-the-fly frequency changes can be made by simply writing to the PLL_DIV register.

All on-chip peripherals are clocked by the system clock (SCLK). The system clock frequency is programmable by means of the SSEL[3:0] bits of the PLL_DIV register.

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 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. In this mode, the CLKIN to VCO multiplier ratio can be changed, although the changes are not realized until the full on mode is entered. DMA access is available to appropriately configured L1 memories.

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 Power Savings)

The sleep mode reduces power dissipation by disabling the clock to the processor core (CCLK). The PLL and system clock (SCLK), however, continue to operate in this mode. Typically an external event or RTC activity will wake up the processor. When in the sleep mode, assertion of any

interrupt causes the processor to sense the value of the bypass bit (BYPASS) in the PLL control register (PLL_CTL). If bypass is disabled, the processor transitions to the full on mode. If bypass is enabled, the processor transitions to the active mode.

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

Deep Sleep Mode (Maximum Power Savings)

The deep sleep mode maximizes 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

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 (0.8 V to 1.2 V) from an external 2.25 V to 3.6 V supply. Figure 1-3 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.

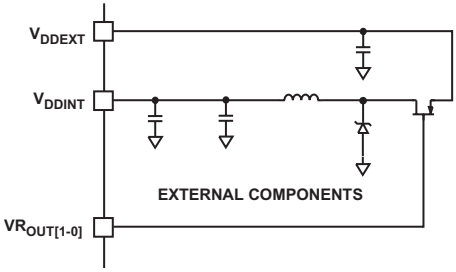


Figure 1-3. Voltage Regulator Circuit

Boot Modes

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

- Execute from 16-bit external memory – Execution starts from address 0x2000 0000 with 16-bit packing. The boot ROM is bypassed in this mode. All configuration settings are set for the slowest device possible (3-cycle hold time; 15-cycle R/W access times; 4-cycle setup).
- Boot from 8-bit and 16-bit external flash memory – The 8-bit or 16-bit flash boot routine located in boot ROM memory space is set up using asynchronous memory bank 0. All configuration settings are set for the slowest device possible (3-cycle hold time; 15-cycle

R/W access times; 4-cycle setup). The boot ROM evaluates the first byte of the boot stream at address 0x2000 0000. If it is 0x40, 8-bit boot is performed. A 0x60 byte is required for 16-bit boot.

- Boot from serial SPI memory (EEPROM or flash). Eight-, 16-, or 24-bit addressable devices are supported as well as AT45DB041, AT45DB081, and AT45DB161 data flash devices from Atmel. The SPI uses the `PF10` output pin to select a single SPI EEPROM/flash device, submits a read command and successive address bytes (0x00) until a valid 8-, 16-, or 24-bit, or Atmel addressable device is detected, and begins clocking data into the processor.
- Boot from SPI host device – The Blackfin processor operates in SPI slave mode and is configured to receive the bytes of the .LDR file from an SPI host (master) agent. To hold off the host device from transmitting while the boot ROM is busy, the Blackfin processor asserts a flag pin to signal the host device not to send any more bytes until the flag is deasserted. The flag is chosen by the user and this information is transferred to the Blackfin processor via bits 8:5 of the FLAG header.
- Boot from UART – Using an autobaud handshake sequence, a boot-stream-formatted program is downloaded by the host. The host agent selects a baud rate within the UART's clocking capabilities. When performing the autobaud, the UART expects a "@" (boot stream) character (eight bits data, one start bit, one stop bit, no parity bit) on the `RXD` pin to determine the bit rate. It then replies with an acknowledgement which is composed of 4 bytes: 0xBF, the value of `UART_DLL`, the value of `UART_DLH`, 0x00. The host can then download the boot stream. When the processor needs to hold off the host, it deasserts `CTS`. Therefore, the host must monitor this signal.

- Boot from serial TWI memory (EEPROM/flash) – The Blackfin processor operates in master mode and selects the TWI slave with the unique id 0xA0. It submits successive read commands to the memory device starting at two byte internal address 0x0000 and begins clocking data into the processor. The TWI memory device should comply with *Philips I^2C Bus Specification* version 2.1 and have the capability to auto-increment its internal address counter such that the contents of the memory device can be read sequentially.
- Boot from TWI host – The TWI host agent selects the slave with the unique id 0x5F. The processor replies with an acknowledgement and the host can then download the boot stream. The TWI host agent should comply with *Philips I^2C Bus Specification* version 2.1. An I^2C multiplexer can be used to select one processor at a time when booting multiple processors from a single TWI.

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

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

Instruction Set Description

The ADSP-BF53x processor family assembly language instruction set employs an algebraic syntax designed for ease of coding and readability. Refer to the *ADSP-BF53x/BF56x 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 microcontrollers, this instruction set is very efficient when compiling C and C++ source code. In addition, the architecture supports both user (algorithm/application code) and supervisor (O/S kernel, device drivers, debuggers, ISRs) modes of operation, allowing multiple levels of access to core resources.

The assembly language, which takes advantage of the 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-BF53x 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 has been 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

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, pre-emptive, 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-BF537 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 other factors that determine the system organization. Following an overview and a list of key features is a block diagram of the chip bus hierarchy and a description of its operation. The chapter concludes with details about the system interconnects and associated system buses.

Overview

The ADSP-BF534, ADSP-BF536, and ADSP-BF537 Blackfin processors feature a powerful chip bus hierarchy on which all data movement between the processor core, internal memory, external memory, and its rich set of peripherals occurs. The chip bus hierarchy includes the controllers for system interrupts, test/emulation, and clock and power management. Synchronous clock domain conversion is provided to support clock domain transactions between the core and the system.

The processor system includes:

- The peripheral set (timers, real-time clock, CAN, TWI, Ethernet MAC (ADSP-BF536 and ADSP-BF537), GPIOs, UARTs, SPORTs, PPI, watchdog timer, and SPI)
- The External Bus Interface Unit (EBIU)
- The Direct Memory Access (DMA) controller
- The interfaces between these, the system, and the optional external (off-chip) resources

The following sections describe the on-chip interfaces between the system and the peripherals via the:

- Peripheral Access Bus (PAB)
- DMA Access Bus (DAB)
- DMA Core Bus (DCB)
- DMA External Bus (DEB)
- External Access Bus (EAB)

The External Bus Interface Unit (EBIU) is the primary chip pin bus and is discussed in Chapter 6, “External Bus Interface Unit”.

Interface Overview

Figure 2-1 shows the core processor and system boundaries as well as the interfaces between them.

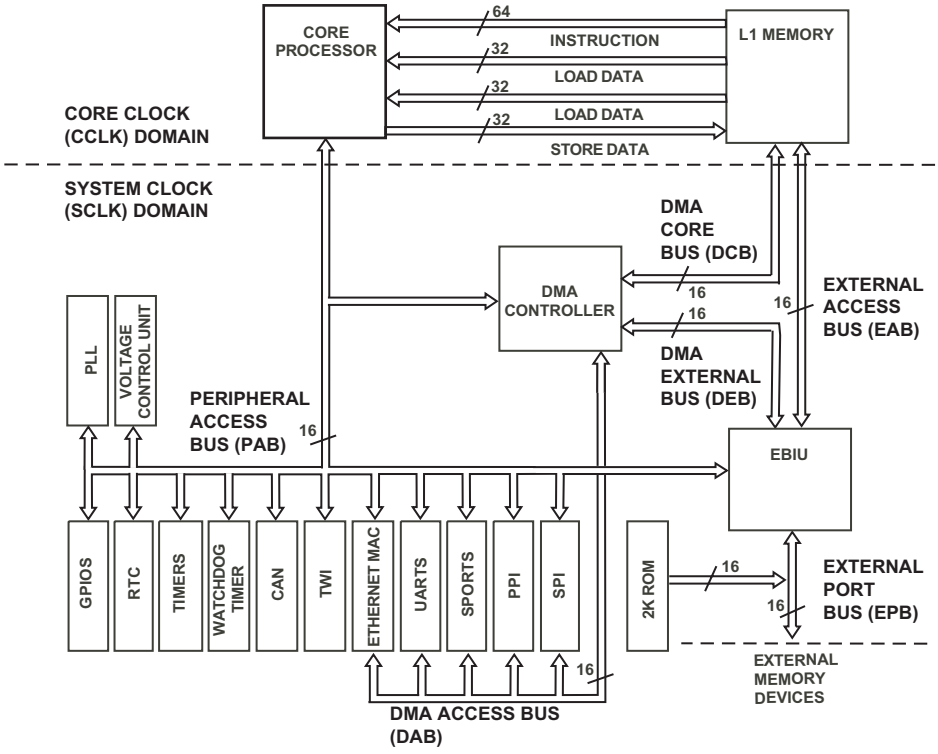


Figure 2-1. Processor Bus Hierarchy

Internal Clocks

The core processor clock (CCLK) rate is highly programmable with respect to CLKIN. The CCLK rate is divided down from the Phase Locked Loop (PLL) output rate. This divider ratio is set using the CSEL parameter of the PLL divide register.

The PAB, the DAB, the EAB, the DCB, the DEB, the EPB, and the EBIU run at system clock frequency (SCLK domain). This divider ratio is set using the SSEL parameter of the PLL divide register and must be set so that these buses run as specified in the processor data sheet, and slower than or equal to the core clock frequency.

These buses can also be cycled at a programmable frequency to reduce power consumption, or to allow the core processor to run at an optimal frequency. Note all synchronous peripherals derive their timing from the SCLK. For example, the UART clock rate is determined by further dividing this clock frequency.

Core Bus Overview

For the purposes of this discussion, level 1 memories (L1) are included in the description of the core; they have full bandwidth access from the processor core with a 64-bit instruction bus and two 32-bit data buses.

Figure 2-2 shows the core processor and its interfaces to the peripherals and external memory resources.

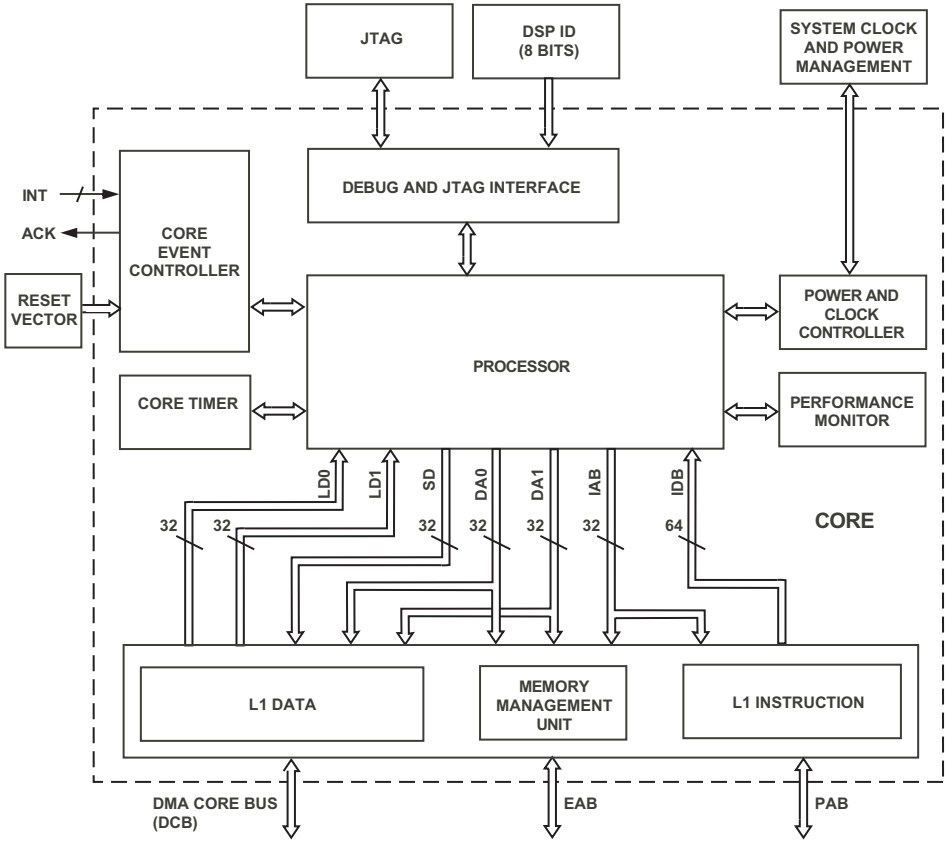


Figure 2-2. Core Block Diagram

The core can generate up to three simultaneous off-core accesses per cycle.

The core bus structure between the processor and L1 memory runs at the full core frequency and has data paths up to 64 bits.

When the instruction request is filled, the 64-bit read can contain a single 64-bit instruction or any combination of 16-, 32-, or 64-bit (partial) instructions.

When cache is enabled, four 64-bit read requests are issued to support 32-byte line fill burst operations. These requests are pipelined so that each transfer after the first is filled in a single, consecutive cycle.

Peripheral Access Bus (PAB)

The processor has a dedicated low latency peripheral bus that keeps core stalls to a minimum and allows for manageable interrupt latencies to time-critical peripherals. All peripheral resources accessed through the PAB are mapped into the system MMR space of the processor memory map. The core accesses system MMR space through the PAB bus.

The core processor has byte addressability, but the programming model is restricted to only 32-bit (aligned) access to the system MMRs. Byte accesses to this region are not supported.

PAB Arbitration

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

PAB Agents (Masters, Slaves)

The processor core can master bus operations on the PAB. All peripherals have a peripheral bus slave interface which allows the core to access control and status state. These registers are mapped into the system MMR space of the memory map. Appendix B lists system MMR addresses.

The slaves on the PAB bus are:

- System event controller
- Clock and power management controller

- Watchdog timer
- Real-time clock (RTC)
- Timer 0–7
- SPORT0–1
- SPI
- Ports
- UART0–1
- PPI
- TWI
- CAN
- Ethernet MAC
- Asynchronous memory controller (AMC)
- SDRAM controller (SDC)
- DMA controller

PAB Performance

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

For example, the core can transfer up to 32 bits per access to the PAB slaves. With the core clock running at 2x the frequency of the system clock, the first and subsequent system MMR read or write accesses take four core clocks (CCLK) of latency.

The PAB has a maximum frequency of SCLK.

DMA Access Bus (DAB), DMA Core Bus (DCB), DMA External Bus (DEB)

The DAB, DCB, and DEB buses provide a means for DMA-capable peripherals to gain access to on-chip and off-chip memory with little or no degradation in core bandwidth to memory.

DAB Arbitration

Sixteen DMA channels and bus masters support the DMA-capable peripherals in the processor system. The twelve peripheral DMA channel controllers can transfer data between peripherals and internal or external memory. Both the read and write channels of the dual-stream memory DMA controller access their descriptor lists through the DAB.

The DCB has priority over the core processor on arbitration into L1 configured as SRAM. For off-chip memory, the core (by default) has priority over the DEB for accesses to the EPB. The processor has a programmable priority arbitration policy on the DAB. Table 2-1 shows the default arbitration priority. In addition, by setting the `CDPRIO` bit in the `EBIU_AMGCTL` register, all DEB transactions to the EPB have priority over core accesses to external memory. Use of this bit is application-dependent. For example, if you are polling a peripheral mapped to asynchronous memory with long access times, by default the core will “win” over DMA requests. By setting the `CDPRIO` bit, the core would be held off until DMA requests were serviced.

Table 2-1. DAB, DCB, and DEB Arbitration Priority

DAB, DCB, DEB Master	Default Arbitration Priority
PPI receive/transmit	0 - highest
Ethernet receive	1
Ethernet transmit	2
SPORT0 receive	3
SPORT0 transmit	4
SPORT1 receive	5
SPORT1 transmit	6
SPI receive/transmit	7
UART0 receive	8
UART0 transmit	9
UART1 receive	10
UART1 transmit	11
MDMA stream 0 destination	12
MDMA stream 0 source	13
MDMA stream 1 destination	14
MDMA stream 1 source	15 - lowest

DAB Bus Agents (Masters)

All peripherals capable of sourcing a DMA access are masters on this bus, as shown in Table 2-1. A single arbiter supports a programmable priority arbitration policy for access to the DAB.

When two or more DMA master channels are actively requesting the DAB, bus utilization is considerably higher due to the DAB's pipelined design. Bus arbitration cycles are concurrent with the previous DMA access's data cycles.

DAB, DCB, and DEB Performance

The processor DAB supports data transfers of 16 bits or 32 bits. The data bus has a 16-bit width with a maximum frequency as specified in the processor data sheet.

The DAB has a dedicated port into L1 memory. No stalls occur as long as the core access and the DMA access are not to the same memory bank (4K byte size for L1). 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. Latencies caused by these stalls are in addition to any arbitration latencies.

The core processor and the DAB must arbitrate for access to external memory through the EBIU. This additional arbitration latency added to the latency required to read off-chip memory devices can significantly degrade DAB throughput, potentially causing peripheral data buffers to underflow or overflow. If you use DMA peripherals other than the memory DMA controller, and you target external memory for DMA accesses, you need to carefully analyze your specific traffic patterns. Make sure that isochronous peripherals targeting internal memory have enough allocated bandwidth and the appropriate maximum arbitration latencies.

External Access Bus (EAB)

The EAB provides a way for the processor core to directly access off-chip memory.

Arbitration of the External Bus

Arbitration for use of external port bus interface resources is required because of possible contention between the potential masters of this bus. A fixed-priority arbitration scheme is used. That is, core accesses via the EAB will be of higher priority than those from the DMA external bus (DEB).

DEB/EAB Performance

The DEB and the EAB support single word accesses of either 8-bit or 16-bit data types. The DEB and the EAB operate at the same frequency as the PAB and the DAB, up to the maximum `SCLK` frequency specified in the processor data sheet.

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 2-2 shows many types of 16-bit memory DMA transfers. In the table, it is assumed that no other DMA activity is conflicting with ongoing operations. 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 via the EAB), a 32-bit access to SDRAM (of the form `R0 = [P0];` where P0 points to an address in SDRAM) is always more efficient than executing two 16-bit accesses (of

the form `R0 = W[P0++]`; where P0 points to an address in SDRAM). In this example, a 32-bit SDRAM read takes 10 `SCLK` cycles while two 16-bit reads take 9 `SCLK` cycles each.

Table 2-2. Performance of DMA Access to External Memory

Source	Destination	Approximate SCLKs For n Words (from start of DMA to interrupt at end)
16-bit SDRAM	L1 data memory	n + 14
L1 data memory	16-bit 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)
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 SDRAM	16-bit 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

3 MEMORY

This chapter discusses memory population specific to the ADSP-BF534, ADSP-BF536, and ADSP-BF537 processors. Functional memory architecture is described in the *ADSP-BF53x/BF56x Blackfin Processor Programming Reference*.

Memory Architecture

Figure 3-1 on page 3-3 provides an overview of the ADSP-BF534 processor system memory map. Figure 3-2 on page 3-4 shows this information for the ADSP-BF536 processor, and Figure 3-3 on page 3-5 for the ADSP-BF537 processor. For a detailed discussion of how to use them, see the *ADSP-BF53x/BF56x Blackfin Processor Programming Reference*. Note the architecture does not define a separate I/O space. All resources are mapped through the flat 32-bit address space. The memory is byte-addressable.

As shown in Table 3-1, the ADSP-BF534, ADSP-BF536, and ADSP-BF537 processors offer a variety of instruction and data memory configurations.

Table 3-1. Memory Configurations

Type of Memory	ADSP-BF534	ADSP-BF536	ADSP-BF537
Instruction SRAM/Cache, lockable by Way or line	16K byte	16K byte	16K byte
Instruction SRAM	48K byte	48K byte	48K byte
Data SRAM/Cache	32K byte	16K byte	32K byte
Data SRAM	32K byte	16K byte	32K byte
Data Scratchpad SRAM	4K byte	4K byte	4K byte
Total	**132K byte**	**100K byte**	**132K byte**

The upper portion of internal memory space is allocated to the core and system MMRs. Accesses to this area are allowed only when the processor is in supervisor or emulation mode (see the Operating Modes and States chapter of the *ADSP-BF53x/BF56x Blackfin Processor Programming Reference*).

Within the external memory map, four banks of asynchronous memory space and one bank of SDRAM memory are available. Each of the asynchronous banks is 1M byte and the SDRAM bank is up to 512M byte.

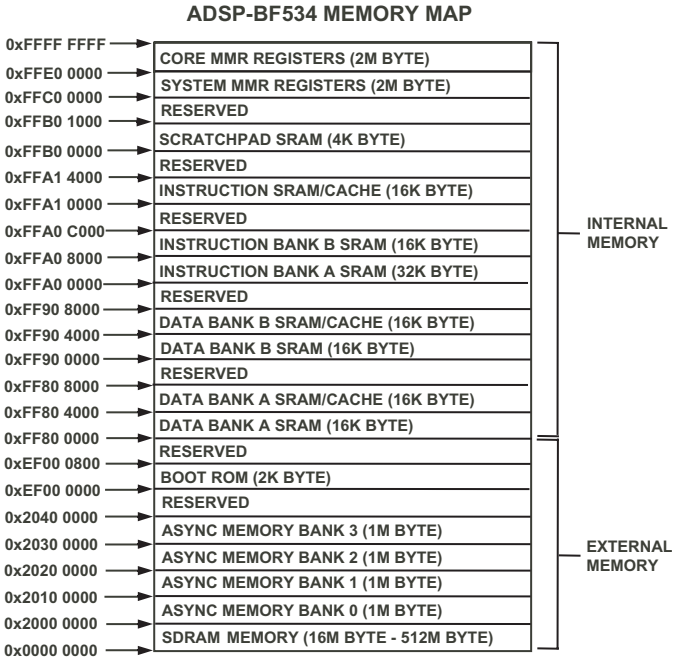


Figure 3-1. ADSP-BF534 Memory Map

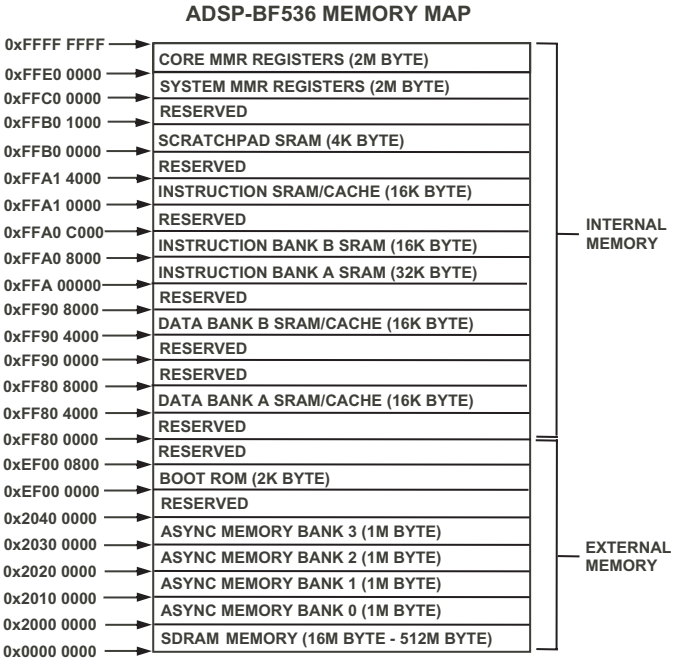


Figure 3-2. ADSP-BF536 Memory Map

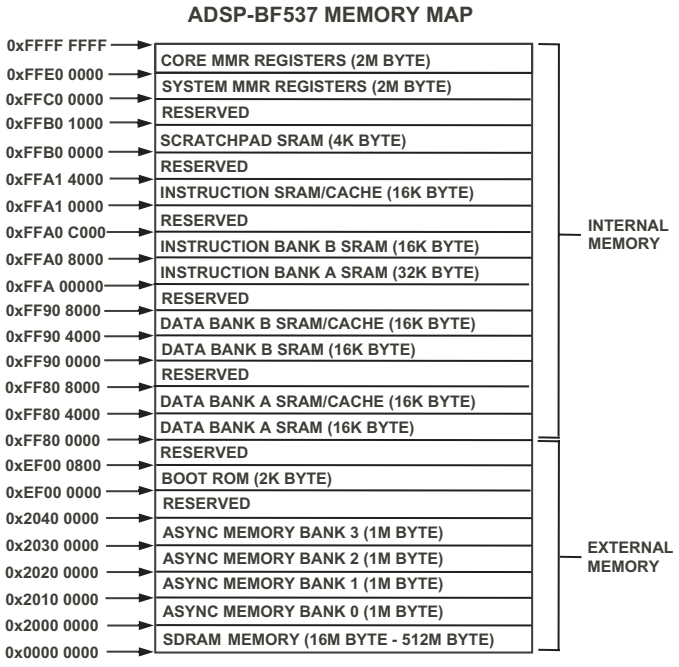


Figure 3-3. ADSP-BF537 Memory Map

L1 Instruction SRAM

The processor core reads the instruction memory through the 64-bit wide instruction fetch bus. All addresses from this bus are 64-bit aligned. Each instruction fetch can return any combination of 16-, 32- or 64-bit instructions (for example, four 16-bit instructions, two 16-bit instructions and one 32-bit instruction, or one 64-bit instruction).

Table 3-2 lists the memory start locations of the L1 instruction memory subbanks.

Table 3-2. L1 Instruction Memory Subbanks

Memory Subbank	Memory Start Location ADSP-BF534, ADSP-BF536, ADSP-BF537
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

L1 Data SRAM

Table 3-3 shows how the subbank organization is mapped into memory.

Table 3-3. L1 Data Memory SRAM Subbank Start Addresses

Memory Bank and Subbank	ADSP-BF534 and ADSP-BF537	ADSP-BF536
Data Bank A, Subbank 0	0xFF80 0000	-
Data Bank A, Subbank 1	0xFF80 1000	-
Data Bank A, Subbank 2	0xFF80 2000	-
Data Bank A, Subbank 3	0xFF80 3000	-
Data Bank A, Subbank 4	0xFF80 4000	0xFF80 4000
Data Bank A, Subbank 5	0xFF80 5000	0xFF80 5000
Data Bank A, Subbank 6	0xFF80 6000	0xFF80 6000
Data Bank A, Subbank 7	0xFF80 7000	0xFF80 7000
Data Bank B, Subbank 0	0xFF90 0000	-
Data Bank B, Subbank 1	0xFF90 1000	-
Data Bank B, Subbank 2	0xFF90 2000	-
Data Bank B, Subbank 3	0xFF90 3000	-
Data Bank B, Subbank 4	0xFF90 4000	0xFF90 4000
Data Bank B, Subbank 5	0xFF90 5000	0xFF90 5000
Data Bank B, Subbank 6	0xFF90 6000	0xFF90 6000
Data Bank B, Subbank 7	0xFF90 7000	0xFF90 7000

L1 Data Cache

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. For the ADSP-BF534 and ADSP-BF537, the upper 16K byte is used.

Boot ROM

The lowest 2K byte of internal memory space is occupied by the boot ROM starting from address 0xEF00 0000. This 16-bit boot ROM is not part of the L1 memory module. Read accesses take one SCLK cycle and no wait states are required. The read-only memory can be read by the core as well as by DMA. It can be cached and protected by CPLD blocks like external memory. The boot ROM not only contains boot-strap loader code, it also provides some subfunctions that are user-callable at runtime. For more information, see Chapter 19, "System Reset and Booting".

External Memory

The external memory space is shown in Figure 3-1 on page 3-3. One of the memory regions is dedicated to SDRAM support. The size of the SDRAM bank is programmable and can range in size from 16M byte to 512M byte. The start address of the bank is 0x0000 0000.

Each of the next four banks contains 1M byte and is dedicated to support asynchronous memories. The start address of the asynchronous memory bank is 0x2000 0000.

Processor-Specific MMRs

The complete set of memory-related MMRs is described in the *ADSP-BF53x/BF56x Blackfin Processor Programming Reference*. Several MMRs have bit definitions specific to the processors described in this manual. These registers are described in the following sections.

DMEM_CONTROL Register

The data memory control register (DMEM_CONTROL), shown in Figure 3-4, contains control bits for the L1 data memory.

Data Memory Control Register (DMEM_CONTROL)

0xFFE0 0004 **Reset = 0x0000 1001**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

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

PORT_PREF1 (DAG1 Port Preference)
0 - DAG1 non-cacheable fetches use port A
1 - DAG1 non-cacheable fetches use port B

PORT_PREF0 (DAG0 Port Preference)
0 - DAG0 non-cacheable fetches use port A
1 - DAG0 non-cacheable fetches use port B

DCBS (L1 Data Cache Bank Select)
Valid only when DMC[1:0] = 11. Determines whether Address bit A[14] or A[23] is used to select the L1 data cache bank.
0 - Address bit 14 is used to select Bank A or B for cache access. If bit 14 of address is 1, select L1 Data Memory Data Bank A; if bit 14 of address is 0, select L1 Data Memory Data Bank B.
1 - Address bit 23 is used to select Bank A or B for cache access. If bit 23 of address is 1, select L1 Data Memory Data Bank A; if bit 23 of address is 0, select L1 Data Memory Data Bank B.

ENDCPLB (Data Cacheability Protection Lookaside Buffer Enable)
0 - CPLBs disabled. Minimal address checking only
1 - CPLBs enabled

DMC[1:0] (L1 Data Memory Configure)
For ADSP-BF534 and ADSP-BF537:
00 - Both data banks are SRAM, also invalidates all cache lines if previously configured as cache
01 - Reserved
10 - Data Bank A is lower 16K byte SRAM, upper 16K byte cache
Data Bank B is SRAM
11 - Both data banks are lower 16K byte SRAM, upper 16K byte cache
For ADSP-BF536:
00 - Data Bank A is SRAM, also invalidates all cache lines if previously configured as cache
01 - Reserved
10 - Data Bank A is cache
11 - Both data banks are lower 16K byte SRAM, upper 16K byte cache

Figure 3-4. L1 Data Memory Control Register

DTEST_COMMAND Register

When the data test command register (DTEST_COMMAND) is written to, the L1 cache data or tag arrays are accessed, and the data is transferred through the data test data registers (DTEST DATA[1:0]). This register is shown in Figure 3-5.

The data/instruction access bit allows direct access via the DTEST_COMMAND MMR to L1 instruction SRAM.

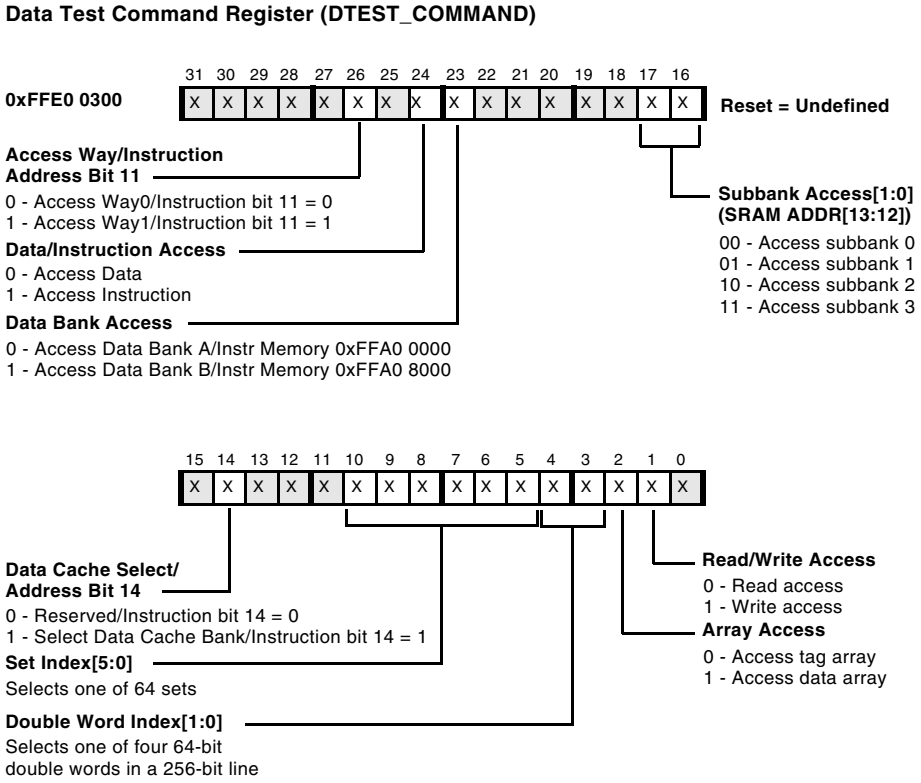


Figure 3-5. Data Test Command Register

4 SYSTEM INTERRUPTS

This chapter discusses the System Interrupt Controller (SIC), which is specific to the ADSP-BF534, ADSP-BF536, ADSP-BF537 derivatives. While this chapter does refer to features of the Core Event Controller (CEC), it does not cover all aspects of it. Please refer to the *ADSP-BF53x/BF56x Blackfin Processor Programming Reference* for more information on the CEC.

Overview

The processor system has numerous peripherals, which therefore require many supporting interrupts.

Features

The Blackfin 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 provides an overview of how the individual peripheral interrupt request lines connect to the SIC. It also shows how the four interrupt assignment registers (SIC_IARx) control the assignment to the nine available peripheral request inputs of the CEC.

The memory-mapped ILAT, IMASK, and IPEND registers are part of the CEC controller. The interrupt requests sourced by the Ethernet MAC (MAC) shown in Figure 4-1 are not available on ADSP-BF534 parts.

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

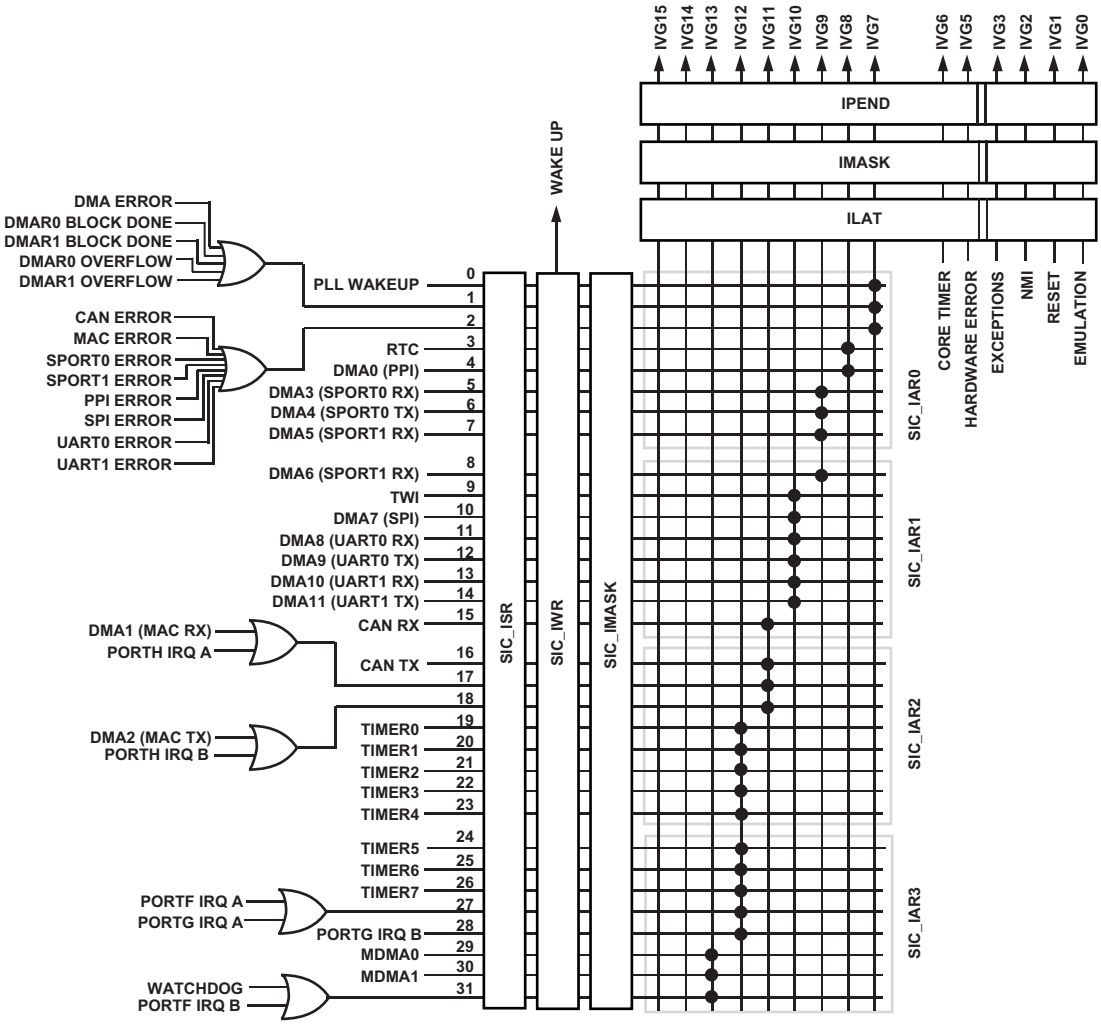


Figure 4-1. Interrupt Routing Overview

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. 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.

Table 4-1. System and Core Event Mapping

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

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

	Event Source	Core Event Name
System interrupts	PLL wakeup interrupt DMA error (generic) DMAR0 block done DMAR1 block done DMAR0 overflow DMAR1 overflow CAN error interrupt MAC error interrupt PPI error interrupt SPORT0 error interrupt SPORT1 error interrupt SPI error interrupt UART0 error interrupt UART1 error interrupt	IVG7
	Real-Time clock interrupts DMA0 interrupt (PPI)	IVG8
	DMA3 interrupt (SPORT0 RX) DMA4 interrupt (SPORT0 TX) DMA5 interrupt (SPORT1 RX) DMA6 interrupt (SPORT1 TX)	IVG9

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

	Event Source	Core Event Name
System interrupts, continued	DMA9 interrupt (UART0 TX) TWI interrupt DMA7 interrupt (SPI) DMA8 interrupt (UART0 RX) DMA10 interrupt (UART1 RX) DMA11 interrupt (UART1 TX)	IVG10
	Port H interrupt A CAN RX interrupt CAN TX interrupt DMA1 interrupt (MAC RX) DMA2 interrupt (MAC TX) Port H interrupt B	IVG11
	Timer 0 interrupt Timer 1 interrupt Timer 2 interrupt Timer 3 interrupt Timer 4 interrupt Timer 5 interrupt Timer 6 interrupt Timer 7 interrupt Port F interrupt A Port G interrupt A Port G interrupt B	IVG12
	MDMA0 interrupt MDMA1 interrupt Software watchdog timer Port F interrupt B	IVG13
	Software interrupt 1	IVG14
	Software interrupt 2 (lowest priority)	IVG15

Note the system interrupt to core event mappings shown 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 32 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 9 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 9 general-purpose interrupt inputs (IVG7–IVG15) of the core event controller have fixed priority. The IVG0 channel has the highest and IVG15 has the lowest priority. Therefore, the interrupt assignment in the SIC_IARx registers not only groups peripheral interrupts 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 4-4 on page 4-18, Figure 4-5 on page 4-18, Figure 4-6 on page 4-19,and Figure 4-7 on page 4-19. If more than one interrupt source is mapped to the same interrupt, they are logically ORed, 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 at all and always have higher priority than requests from all other peripherals.

The system interrupt mask register (SIC_IMASK, shown in Figure 4-8 on page 4-20) allows software to mask any peripheral interrupt source at the system interrupt controller (SIC) level. This functionality is independent

of whether the particular interrupt is enabled at the peripheral itself. At reset, the contents of SIC_IMASK 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 SIC_IMASK.

The SIC includes a read-only system interrupt status register (SIC_ISR) with individual bits which correspond to one of the peripheral interrupt sources. See Figure 4-9 on page 4-21. When the SIC detects the interrupt, the bit is asserted. When the SIC detects that the peripheral interrupt input has been 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 has been deasserted.

Depending on how interrupt sources map to the general-purpose interrupt inputs of the core, the interrupt service routine may have to interrogate multiple interrupt status bits to determine the source of the interrupt. One of the first instructions executed in an interrupt service routine should read SIC_ISR 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_ISR 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_ISR will seldom, if ever, need to be interrogated.

The SIC_ISR register is not affected by the state of the system interrupt mask register (SIC_IMASK) and can be read at any time. Writes to the SIC_ISR 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 and Figure 4-2 show the default DMA assignment. For more information on DMA, see Chapter 5, "Direct Memory Access". Once a peripheral has been assigned to any other 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.

The ADSP-BF534 does not include the MAC requests shown in Figure 4-2. However, for code compatibility, all default assignments are the same as on the ADSP-BF536 and ADSP-BF537.

For dynamic power management, any of the peripherals can be configured to wake up the core from its idled state to process the interrupt, simply by enabling the appropriate bit in the system interrupt wakeup-enable register (SIC_IWR, refer to Figure 4-10 on page 4-22). If a peripheral interrupt source is enabled in SIC_IWR and the core is idled, the interrupt causes the DPMC to initiate the core wakeup sequence in order to process the interrupt. Note this mode of operation may add latency to interrupt processing, depending on the power control state. For further discussion of power modes and the idled state of the core, see Chapter 20, "Dynamic Power Management".

The SIC_IWR register has no effect unless the core is idled. 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 SIC_IWR 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.

The wakeup function is independent of the interrupt mask function. If an interrupt source is enabled in `SIC_IWR` but masked off in `SIC_IMASK`, the core wakes up if it is idled, but it does not generate an interrupt.

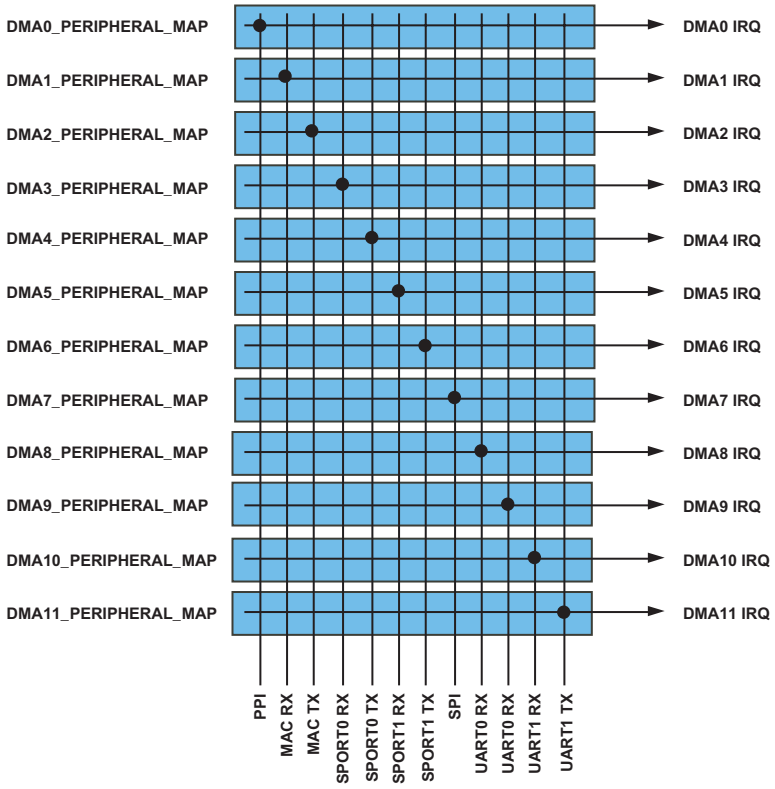


Figure 4-2. Default Peripheral-to-DMA Mapping

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 "SIC_IARx Registers" on page 4-18.

Table 4-2. System Interrupt Controller (SIC)

Peripheral Interrupt Event	Default DMA Source Mapping	Peripheral Interrupt ID	Default Mapping	Default Core Interrupt ID
PLL wakeup		0	IVG7	0
DMA error (generic)		1	IVG7	0
DMAR0 block interrupt		1	IVG7	0
DMAR1 block interrupt		1	IVG7	0
DMAR0 overflow error		1	IVG7	0
DMAR1 overflow error		1	IVG7	0
CAN error		2	IVG7	0
MAC error[1]		2	IVG7	0
SPORT 0 error		2	IVG7	0
SPORT 1 error		2	IVG7	0
PPI error		2	IVG7	0
SPI error		2	IVG7	0
UART0 error		2	IVG7	0
UART1 error		2	IVG7	0
RTC		3	IVG8	1
DMA channel 0	PPI	4	IVG8	1
DMA channel 3	SPORT 0 RX	5	IVG9	2
DMA channel 4	SPORT 0 TX	6	IVG9	2
DMA channel 5	SPORT 1 RX	7	IVG9	2

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

Peripheral Interrupt Event	Default DMA Source Mapping	Peripheral Interrupt ID	Default Mapping	Default Core Interrupt ID
DMA channel 6	SPORT 1 TX	8	IVG9	2
TWI	IVG10	9	IVG10	3
DMA channel 7	SPI	10	IVG10	3
DMA channel 8	UART0 RX	11	IVG10	3
DMA channel 9	UART0 TX	12	IVG10	3
DMA channel 10	UART1 RX	13	IVG10	3
DMA channel 11	UART1 TX	14	IVG10	3
CAN RX		15	IVG11	4
CAN TX		16	IVG11	4
DMA channel 1[1]	MAC RX	17	IVG11	4
Port H interrupt A		17	IVG11	4
DMA channel 2[1]	MAC TX	18	IVG11	4

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

Peripheral Interrupt Event	Default DMA Source Mapping	Peripheral Interrupt ID	Default Mapping	Default Core Interrupt ID
Port H interrupt B		18	IVG11	4
Timer 0		19	IVG12	5
Timer 1		20	IVG12	5
Timer 2		21	IVG12	5
Timer 3		22	IVG12	5
Timer 4		23	IVG12	5
Timer 5		24	IVG12	5
Timer 6		25	IVG12	5
Timer 7		26	IVG12	5
Port F, G interrupt A		27	IVG12	5
Port G interrupt B		28	IVG12	5
Memory DMA stream 0		29	IVG13	6
Memory DMA stream 1		30	IVG13	6
Software watchdog timer		31	IVG13	6
Port F interrupt B		31	IVG13	6

1 MAC error and DMA requests are not available on the ADSP-BF534. However, the DMA channels 1 and 2 can be assigned to other peripherals by reprogramming the DMA1_PERIPHERAL_MAP and DMA2_PERIPHERAL_MAP registers to values different than the default values.

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.

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 4-11 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 SIC_IMASK that the system requires

System Interrupt Processing Summary

Referring to Figure 4-3 on page 4-16, note when an interrupt (interrupt A) is generated by an interrupt-enabled peripheral:

1. SIC_ISR logs the request and keeps track of system interrupts that are asserted but not yet serviced (that is, an interrupt service routine hasn't yet cleared the interrupt).
2. SIC_IWR checks to see if it should wake up the core from an idled state based on this interrupt request.
3. SIC_IMASK 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 registers, which map the peripheral interrupts to a smaller set of general-purpose core interrupts (IVG7 - IVG15), determine the core priority of interrupt A.
5. ILAT adds interrupt A to its log of interrupts latched by the core but not yet actively being serviced.
6. IMASK masks off or enables events of different core priorities. If the IVGx event corresponding to interrupt A is not masked, the process proceeds to Step 7.
7. The event vector table (EVT) is accessed to look up the appropriate vector for interrupt A's interrupt service routine (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, IPEND tracks all pending interrupts, as well as those being presently serviced.
9. When the interrupt service routine (ISR) for interrupt A has been executed, the RTI instruction clears the appropriate IPEND bit. However, the relevant SIC_ISR bit is not cleared unless the interrupt service routine clears the mechanism that generated interrupt A, or if the process of servicing the interrupt clears this bit.

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

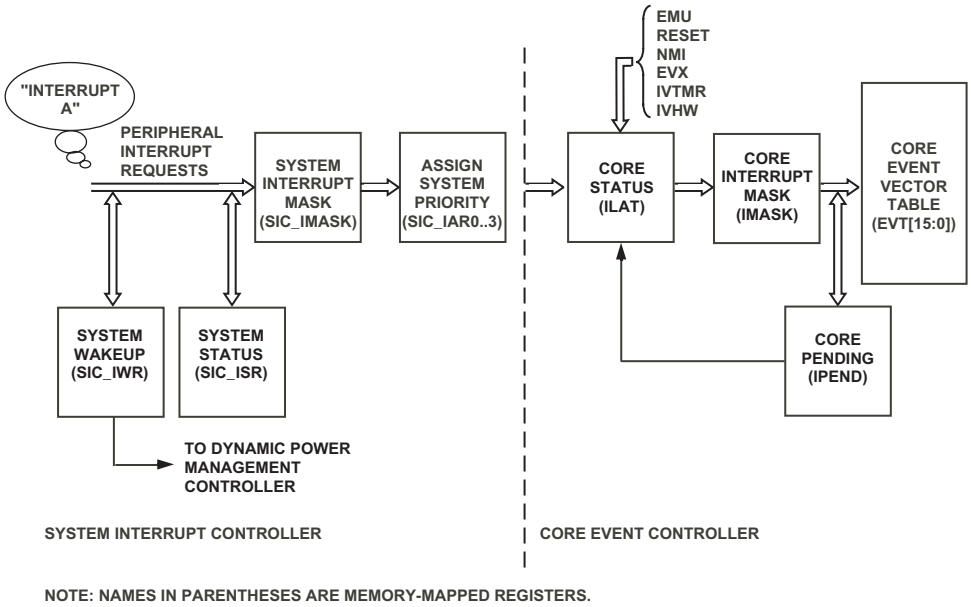


Figure 4-3. Interrupt Processing Block Diagram

System Interrupt Controller Registers

The SIC registers are described in the following sections.

These 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.

Table 4-3 defines the value to write in SIC_IARx to configure a peripheral for a particular IVG priority.

Table 4-3. 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

SIC_IARx Registers

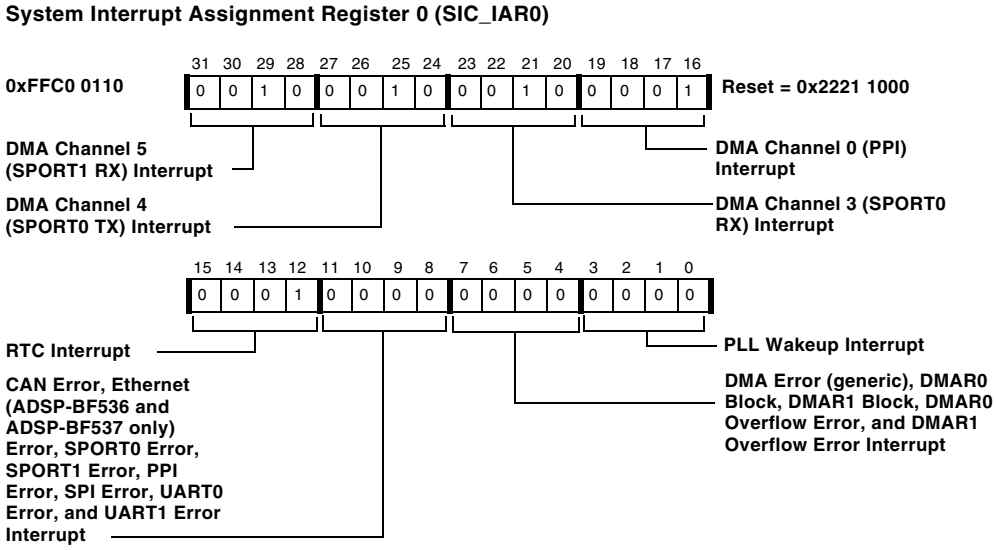


Figure 4-4. System Interrupt Assignment Register 0

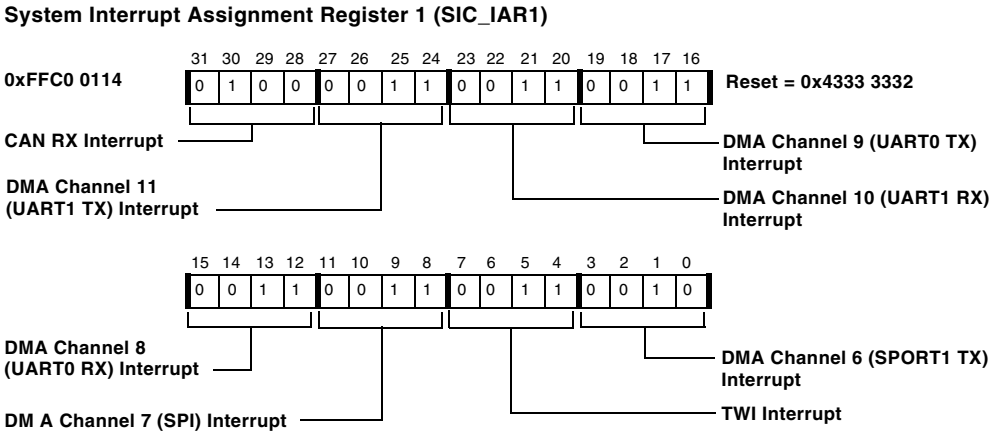


Figure 4-5. System Interrupt Assignment Register 1

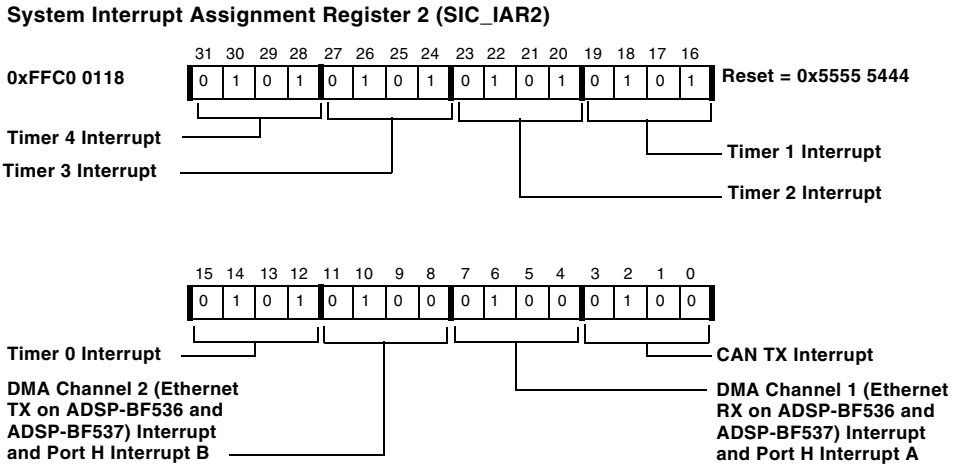


Figure 4-6. System Interrupt Assignment Register 2

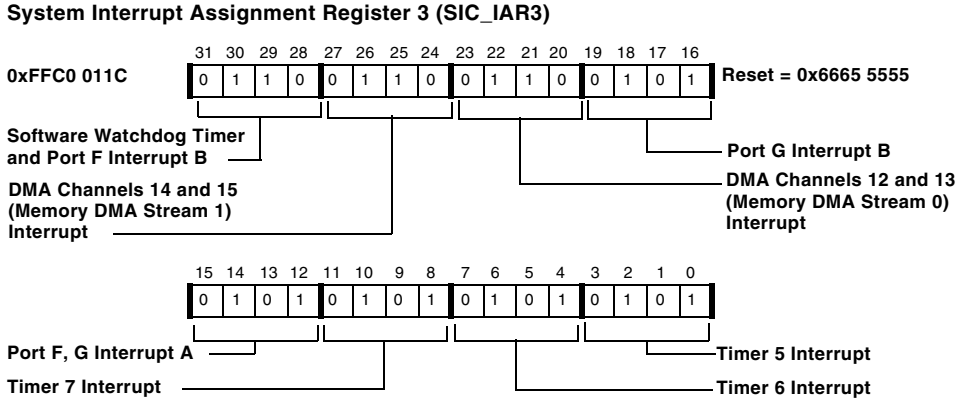


Figure 4-7. System Interrupt Assignment Register 3

SIC_IMASK Register

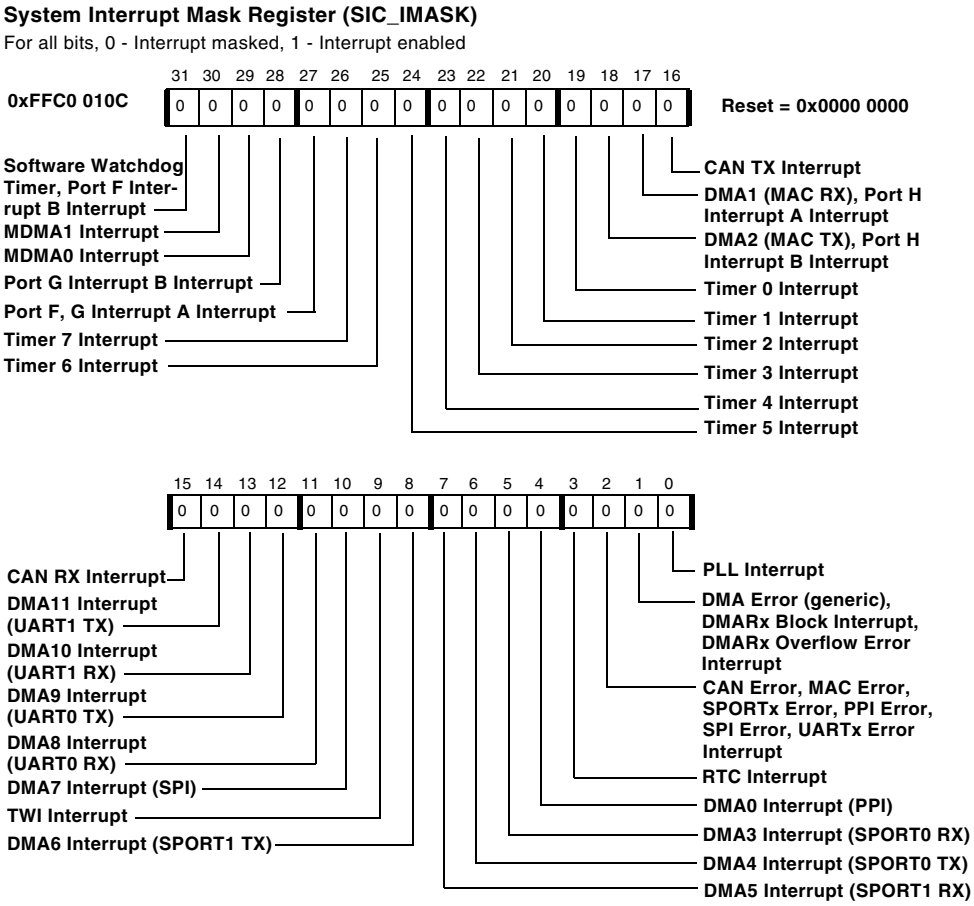


Figure 4-8. System Interrupt Mask Register

SIC_ISR Register

System Interrupt Status Register (SIC_ISR)

For all bits, 0 - Deasserted, 1 - Asserted

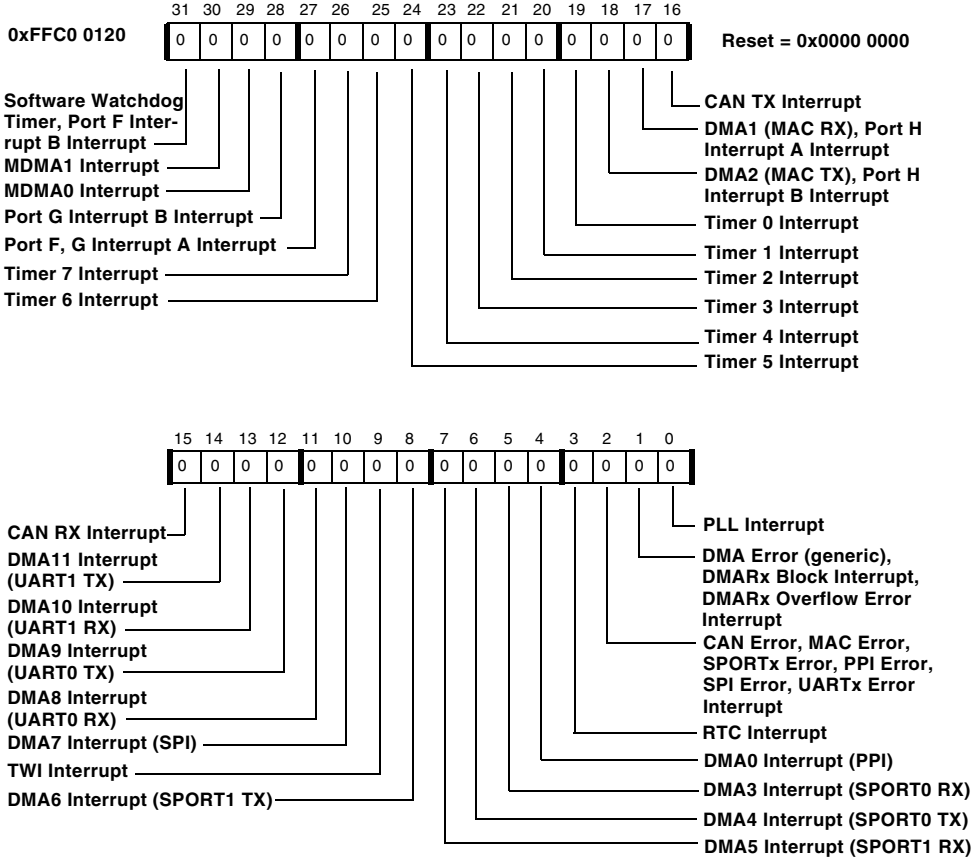


Figure 4-9. System Interrupt Status Register

SIC_IWR Register

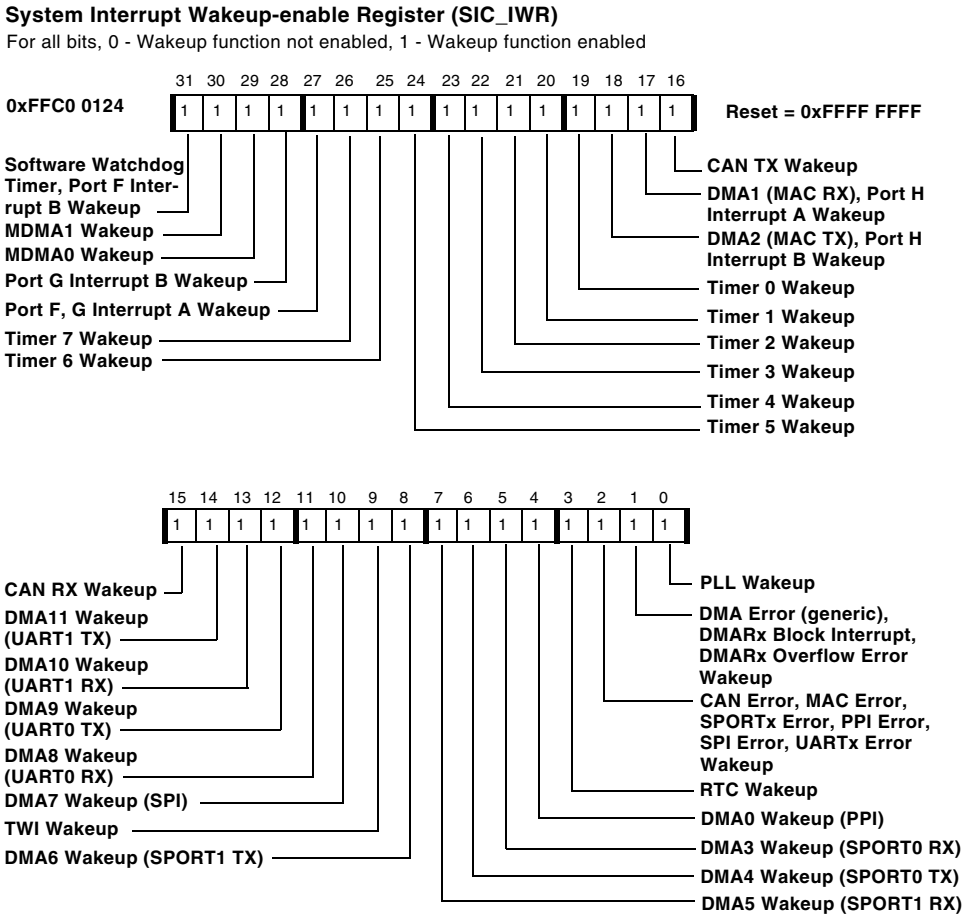


Figure 4-10. System Interrupt Wakeup-enable Register

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 <defBF537.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 W1C 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 W1C 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

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_ISR 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) controller. Following an overview and list of key features is a description of operation and functional modes of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

This chapter describes the features common to all the DMA channels, as well as how DMA operations are set up. For specific peripheral features, see the appropriate peripheral chapter for additional information. Performance and bus arbitration for DMA operations can be found in "DAB, DCB, and DEB Performance" on page 2-10.

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 DMA controller can perform several types of data transfers:

- Peripheral DMA transfers data between memory and on-chip peripherals. The processor has 12 peripheral DMA channels that support 7 peripherals.
    - Ethernet MAC (dedicated DMA channel for transmit and receive. The Ethernet MAC is not available on ADSP-BF534 processors)
    - SPORT0 and SPORT1 (dedicated DMA channel for transmit and receive)
    - UART0 and UART1 (dedicated DMA channel for transmit and receive)
    - PPI (transmit and receive share one DMA channel)
    - SPI (transmit and receive share one DMA channel)
- Memory DMA (MDMA) transfers data between memory and memory. The processor has two 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, boot ROM, and 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)
- 1D DMA, using a set of identical ping-pong buffers defined by a linked ring of 3-word descriptors, each containing { link pointer, 32-bit address }
- 1D DMA, using a linked list of 5-word descriptors containing { link pointer, 32-bit address, length, config } (ADSP-2191 style)
- 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 16 functions can be served by DMA channels:

- PPI receive/transmit
- Ethernet receive (not present on ADSP-BF534)
- Ethernet transmit (not present on ADSP-BF534)

- SPORT0 receive
- SPORT0 transmit
- SPORT1 receive
- SPORT1 transmit
- SPI receive/transmit
- UART0 receive
- UART0 transmit
- UART1 receive
- UART1 transmit
- MDMA0 destination
- MDMA0 source
- MDMA1 destination
- MDMA1 source

DMA Controller Overview

Figure 5-1 provides a block diagram of the DMA controller.

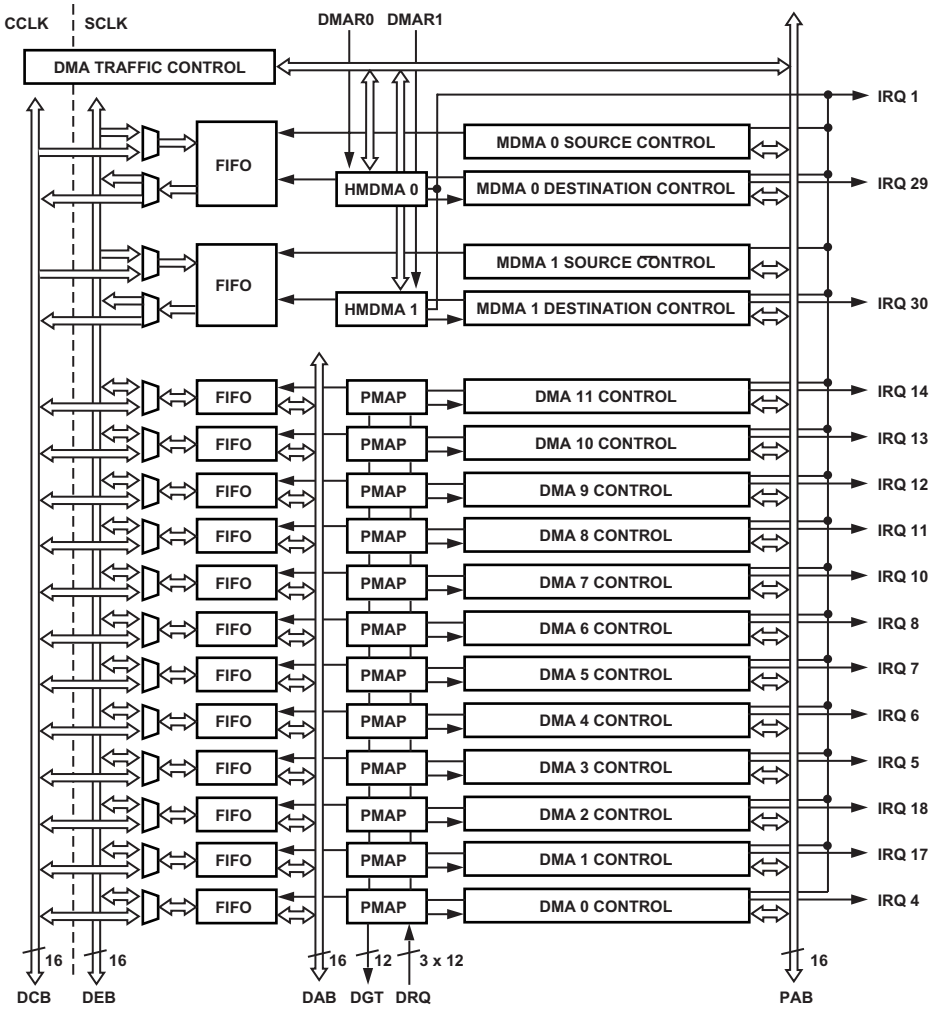


Figure 5-1. DMA Controller Block Diagram

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, 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 MDMA0 destination channel. The DMAR1 pin controls the destination channel of MDMA1. 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 F and compete with UART0 signals. To enable their function, set the PFDE bit in the PORT_MUX register and the PF0 and/or PF1 bits in the PORTF_FER register. 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 controller to interconnect L1 memory, the on-chip peripherals, and the EBIU port.

The 16-bit DMA Core Bus (DCB) connects the DMA controller to a dedicated port of L1 memory. L1 memory has dedicated DMA ports featuring special DMA buffers to decouple DMA operation. See the *ADSP-BF53x/BF56x Blackfin Processor Programming Reference* for a description of the L1 memory architecture. The DCB bus operates at core clock (CCLK) frequency. It is the DMA controller's responsibility to translate DCB transfers to the system clock (SCLK) domain.

The 16-bit DMA Access Bus (DAB) connects the DMA controller to the on-chip peripherals, PPI, SPI, Ethernet MAC, the SPORTs, and the UARTs. It operates at SCLK frequency.

The 16-bit DMA External Bus (DEB) connects the DMA controller to the EBIU port. This path is used for all peripheral and memory DMA transfers to and from external memories and devices. It operates at SCLK frequency.

Transferred data can be 8, 16, or 32 bits wide. The DMA controller, however, connects only to 16-bit buses.

Memory DMA can pass data every SCLK cycle between L1 memory and the EBIU. Transfers from L1 memory to L1 memory requires 2 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 5-44.

Peripheral DMA

As can be seen in Figure 5-1 on page 5-5, the DMA controller features 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	PMAP Default Value	Peripheral Mapped by Default
DMA 0	0x0	PPI receive or transmit
DMA 1	0x1	Ethernet MAC receive

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

DMA Channel	PMAP Default Value	Peripheral Mapped by Default
DMA 2	0x2	Ethernet MAC transmit
DMA 3	0x3	SPORT0 receive
DMA 4	0x4	SPORT0 transmit
DMA 5	0x5	SPORT1 receive
DMA 6	0x6	SPORT1 transmit
DMA 7	0x7	SPI
DMA 8	0x8	UART0 receive
DMA 9	0x9	UART0 transmit
DMA 10	0xA	UART1 receive
DMA 11	0xB	UART1 transmit

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. When competing for any of the system buses, DMA0 has higher priority than DMA1, and so on. DMA 11 has the lowest priority of the peripheral DMA channels.

Although ADSP-BF534 processors do not feature the Ethernet MAC module, DMA 1 and DMA 2 channels are still present and can be used for other purposes. Attention is required as their default PMAP setting is invalid on ADSP-BF534 devices.

Note a 1:1 mapping should exist between DMA channels and peripherals. The user is responsible for ensuring that multiple DMA channels are not mapped to the same peripheral and that multiple peripherals are not mapped to the same DMA port. If multiple channels are mapped to the same peripheral, only one channel is connected (the lowest priority channel). If a nonexistent peripheral (for example, 0xF in the `PMAP` field) is mapped to a

channel, that channel is disabled—DMA requests are ignored, and no DMA grants are issued. The DMA requests are also not forwarded from the peripheral to the interrupt controller.

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

Every DMA channel features its own 4-depth 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 `DMAx_CONFIG` register for details.

Memory DMA

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

Each MDMA controller contains a DMA FIFO, an 8-word by 16-bit FIFO block used to transfer data to and from either L1 or the DCB and DEB buses. Typically, it is used to transfer data between external memory and internal memory. It will also support DMA from boot ROM on the DEB bus. The FIFO can be used to hold DMA data transferred between two L1 memory 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)

A memory-to-memory transfer always requires the source and the destination channel to be enabled. The four channels are hardwired for DMA priorities 12 through 15. Each source/destination channel forms a "stream," and these two streams are hardwired for DMA priorities 8 through 11.

- Priority 12: MDMA0 destination
- Priority 13: MDMA0 source
- Priority 14: MDMA1 destination
- Priority 15: MDMA1 source

MDMA0 takes precedence over MDMA1, unless round robin scheduling is used or priorities become urgent as programmed by the `DRQ` bit field in the `HMDMA_CONTROL` register. Note 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 the MDMA 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. Each pair shares an 8-word-deep 16-bit FIFO. The source DMA engine fills the FIFO, while the destination DMA engine empties it. The FIFO depth allows the burst transfers of the 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.

Handshaked Memory DMA Mode

Handshaked operation applies only to memory DMA channels.

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 MDMA0 and the DMAR1 input with MDMA1. 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 $\overline{\text{MSx}}$ 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 5-40 for a functional description.

Modes of Operation

The following sections describe the DMA operation.

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. This value must always be initialized as there is no default value. Typically, this register is set to `0x0004` for 32-bit DMA transfers, to `0x0002` for 16-bit transfers, and to `0x0001` for byte transfers.

- 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-one-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

should clear 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 64 K x 64 K elements, as well as arbitrary DMAx_X_MODIFY and DMAx_Y_MODIFY values up to ±32 K bytes. Furthermore, DMAx_Y_MODIFY can be negative, allowing implementation of interleaved datastreams. The DMAx_X_COUNT and DMAx_Y_COUNT values specify the row and column sizes, where DMAx_X_COUNT must be 2 or greater.

The start address and modify values are in bytes, and they must be aligned to a multiple of the DMA transfer word size (WDSIZE[1:0] in DMAx_CONFIG). Misalignment causes a DMA error.

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 DMAx_CURR_X_COUNT from 1 to 0, except that it is applied on the final transfer when DMAx_CURR_Y_COUNT 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 DMAx_CURR_Y_COUNT. 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, DMAx_CURR_Y_COUNT = 1, DMAx_CURR_X_COUNT = 0, and DMAx_CURR_ADDR is 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 will be 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 × 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 address offsets 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) × (N × 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 address offsets from the start address:

```
0, (N * M), 2(N * M),
1, (N * M) + 1, 2(N * M) + 1,
2, (N * M) + 2, 2(N * M) + 2,
    ...
(N * M) - 1, 2(N * M) - 1, 3(N * M) - 1,
```

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 or from external memory using the DEB bus.

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 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 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 64 KB 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 must be non-zero. The descriptor size can be any value from 1 entry (the lower 16 bits of DMAx_START_ADDR only) to 9 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 YCNT and YMOD. 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.

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

Note that every descriptor fetch steals bandwidth from 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 has been 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.

DMA Operation Flow

Figure 5-2 and Figure 5-3 describe the DMA flow.

DMA Startup

This section discusses starting DMA "from scratch." This is similar to starting it after it has been paused by FLOW = 0 mode.

Before initiating DMA for the first time on a given channel, be sure to initialize all parameter registers. Be especially careful to initialize the upper 16 bits of 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.

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, DMAx_Y_MODIFY). The contents of NDSIZE and FLOW in DMAx_CONFIG indicate which registers, if any, are fetched from

descriptor elements in memory. After the descriptor fetch, if any, is completed, DMA operation begins, initiated by writing DMAx_CONFIG with DMAEN = 1.

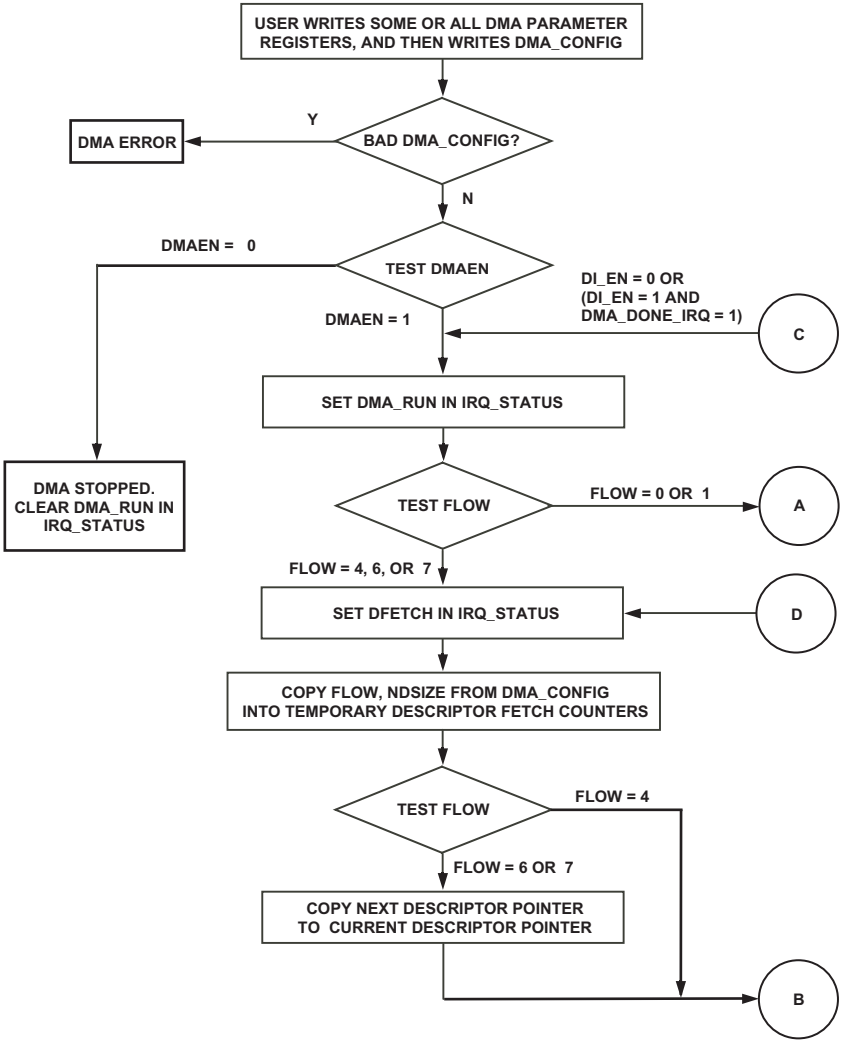


Figure 5-2. DMA Flow, From DMA Controller's Point of View (1 of 2)

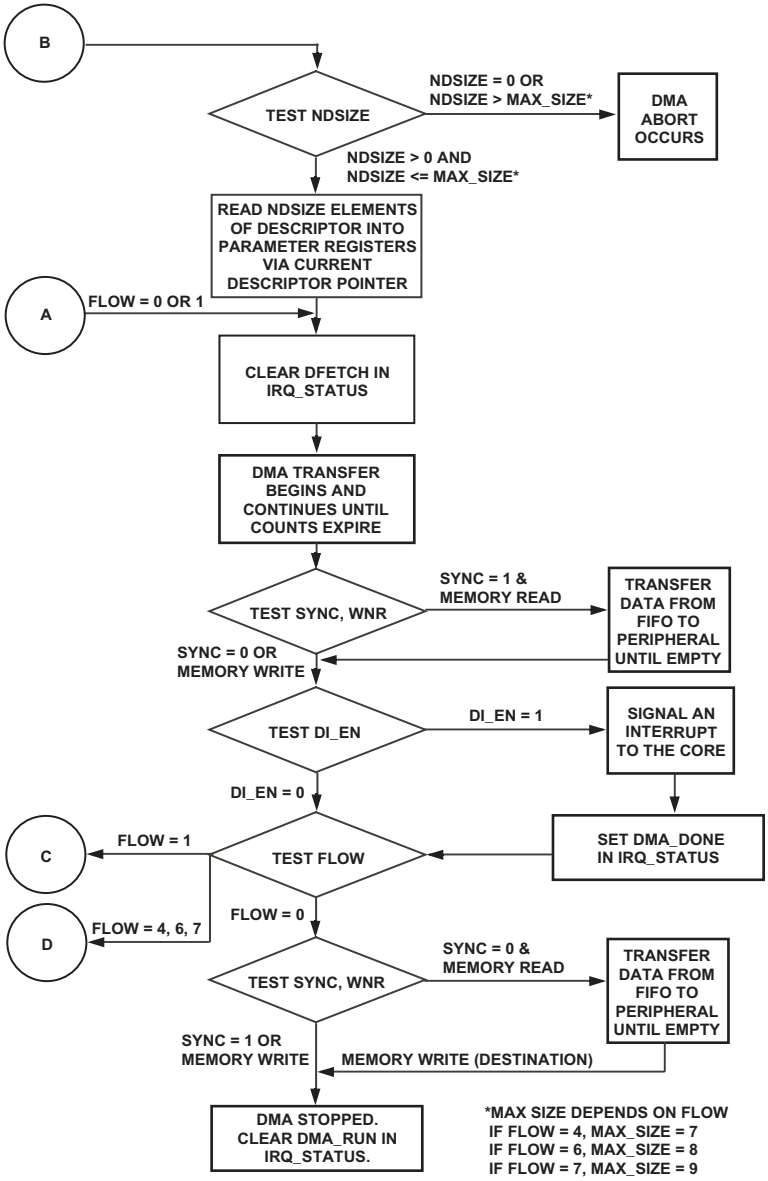


Figure 5-3. DMA Flow, From DMA Controller's Point of View (2 of 2)

When DMAx_CONFIG 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 has been stopped (FLOW = 0).

When the descriptor fetch is complete and DMAEN = 1, the DMACFG descriptor element that was read into DMAx_CONFIG assumes control. Before this point, the direct write to DMAx_CONFIG had control. In other words, the WDSIZE, DI_EN, DI_SEL, SYNC, and DMA2D fields will be 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 5-2 on page 5-21 and Figure 5-3 on page 5-22 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 is copied into DMAx_CURR_DESC_PTR. Then, fetches of new descriptor elements from memory are performed, indexed by DMAx_CURR_DESC_PTR, which is incremented after each fetch. If NDPL and/or NDPH is part of the descriptor, then these values are loaded into DMAx_NEXT_DESC_PTR, but the fetch of the current descriptor continues using DMAx_CURR_DESC_PTR. After completion of the descriptor fetch, DMAx_CURR_DESC_PTR points to the next 16-bit word in memory past the end of the descriptor.

If neither NDPH nor NDPL are part of the descriptor (that is, in descriptor array mode, FLOW = 4), then the transfer from NDPH/NDPL into DMAx_CURR_DESC_PTR does not occur. Instead, descriptor fetch indexing begins with the value in DMAx_CURR_DESC_PTR.

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 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 DMA_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 5-3 on page 5-22.

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 has been 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):

  Loads a new descriptor from memory into DMA registers via 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):

  Copies the 32-bit DMAx_NEXT_DESC_PTR into DMAx_CURR_DESC_PTR.

Next, fetches a descriptor from memory into DMA registers via 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 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):

Copies the 32-bit DMAx_NEXT_DESC_PTR into DMAx_CURR_DESC_PTR. Next, fetches a descriptor from memory into DMA registers via 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 memory 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) the DMA controller clears the DFETCH bit in the DMAx_IRQ_STATUS register.
- If FLOW = any value but 0 (Stop), 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 5-3 on page 5-22.

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

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.

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 vs. 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 has been 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 vs. 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.

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 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 are 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 vs. external) between descriptors or descriptor chains. DMA transmit (memory read) channels may have such restrictions (see "DMA Transmit and MDMA Source" on page 5-28).

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 will always terminate DMA gracefully (without DMA abort).

If a channel has been 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

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 controller flags conditions that cause the DMA process 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 the whole 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 5-34.

- A disallowed register write occurred while the channel was running. Only the DMAx_CONFIG and DMAx_IRQ_STATUS registers can be written when DMA_RUN = 1.
- An address alignment error occurred during any memory access. For example, DMAx_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 00.
- A memory space transition was attempted (internal-to-external or vice versa). For example, the current DMA address (DMAx_CURR_ADDR) crossed the 0xF000 0000 boundary, or the current descriptor pointer (DMAx_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:

- DMAx_CONFIG direction bit (WNR) does not agree with the direction of the mapped peripheral.
- DMAx_CONFIG direction bit does not agree with the direction of the MDMA channel.
- DMAx_CONFIG word size (WDSIZE) is not supported by the mapped peripheral.
- 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 < NDSIZE <= 7	Descriptor array, no descriptor pointer fetched
6	0 < NDSIZE <= 8	Descriptor list, small descriptor pointer fetched
7	0 < NDSIZE <= 9	Descriptor list, large descriptor pointer fetched

DMA Control Commands

Advanced peripherals, such as the ADSP-BF536/ADSP-BF537 processor's Ethernet MAC module, 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 controller 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.

Note that while these DMA control commands are not visible to or controllable 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.

The ADSP-BF536/ADSP-BF537 processors have just one DMA-management-capable peripheral, the Ethernet MAC. Refer to Chapter 8, "Ethernet MAC", for a description of how receive and transmit channels of this peripheral use DMA control commands. ADSP-BF534 processors are not equipped with DMA-management-capable peripherals. 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
000	NOP	No operation
001	Restart	Restarts the current work unit from the beginning
010	Finish	Finishes the current work unit and starts the next
011	-	Reserved
100	Req Data	Typical DMA data request
101	Req Data Urgent	Urgent DMA data request
110	-	Reserved
111	-	Reserved

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 thus use the restart command to re-attempt 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 re-use the memory buffer for a later data transfer.

- **Finish**

  The finish control command causes the current work unit to terminate processing of the current work unit 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 though 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 have been 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 has been fetched, the channel initializes its FIFO, and then resumes granting DMA requests from the peripheral.

- **Request Data**

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

- **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.

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 DMAx_CURR_X_COUNT/ 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 DMAx_CURR_X_COUNT/ DMAx_CURR_Y_COUNT of the current work unit is sufficiently large that it is always at least 5 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 DMAx_CURR_X_COUNT/ DMAx_CURR_Y_COUNT values representing at least 5 data items.

Note in particular that if the DMAx_CURR_X_COUNT/ DMAx_CURR_Y_COUNT registers are programmed to 0 (representing 65,536 transfers, the maximum value) the channel will operate 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 has already performed at least five DMA transfers in the current work unit, or (b) the previous work unit was terminated by a finish control command and the peripheral has performed at least one DMA transfer in the current work unit. If five data transfers have been performed, then at least one data item has been 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 have been 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.

Similarly, if a finish command ended the previous work unit and at least one subsequent DMA data transfer has occurred, then the fact that the DMA channel issued the grant guarantees that the previous work unit has 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_COUNT`s representing more data items than the maximum work unit size that the peripheral will encounter. For example, `DMAx_CURR_X_COUNT`/ `DMAx_CURR_Y_COUNT`s 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 MDMA0 is enabled by the HMDMAEN bit in the HMDMA0_CONTROL register. Similarly, the HMDMAEN bit in the HMDMA1_CONTROL register enables handshake mode for MDMA1.

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 vs. 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.

The HMDMA unit controls only the destination (memory write) channel of the memory DMA. The source channel (memory-read side) fills the 8-depth DMA buffers immediately after the receive being enabled issues 8 read commands.

The HMDMAx_BCINIT registers control how many data transfers are performed upon every DMA request. If set to one, the peripheral can time every individual data transfer. If greater than one, the peripheral must feature sufficient buffer size to provide or consume the number of words programmed. Once the transfer has been 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 $\overline{BR}$ and $\overline{BG}$ 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 has been started, the HMDMAx_BCOUNT registers return the remaining number of transfers to complete the current block. When the complete block has been 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.

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 haven't been served yet and indicates that the MDMA is currently processing. Negative values indicate the number of DMA requests that will be 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 will disregard the programmed number (up to 32768) significant edges on the DMARx inputs before starting normal operation.

Figure 5-4 illustrates how an asynchronous FIFO could be connected. In such a scenario the REP bit was 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 would reduce 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.

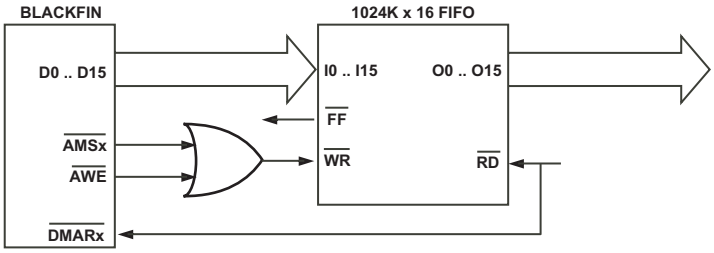


Figure 5-4. Transmit DMA Example Connection

In the receive example shown in Figure 5-5, 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.

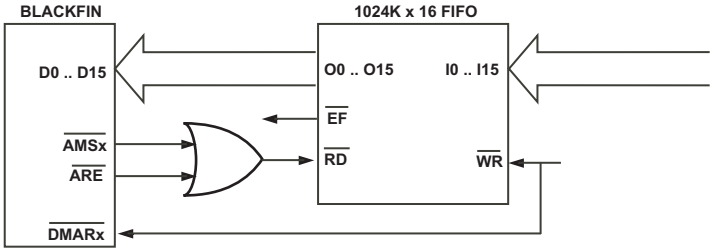


Figure 5-5. Receive DMA Example Connection

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 are routed to the global DMA error interrupt channel. 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 block done interrupt signals that a complete MDMA block as defined by the HMDMAx_BCINIT register has been 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 have been completed.

The overflow interrupt is generated when the HMDMA_ECOUNT register overflows. Since it can count up to 32767, which is much more than most of peripheral devices can support, the Blackfin processor features another threshold register called HMDMA_ECOVERFLOW. It resets to 0xFFFF and should be 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_ECOUNT 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 architecting 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?

- How often do competing DMA channels require the bus systems to alter direction?
- How often do competing DMA or core accesses cause the SDRAM to open different pages?
- Is there a way to distribute DMA requests nicely 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 per two system clocks, per channel, in either direction. As the DAB and DEB buses do, the DMA controller resides in the `SCLK` domain. The controller synchronizes 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 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.
- Transfers between the DMA unit and internal memory (L1) have a maximum rate of one 16-bit transfer per system clock.
- Transfers between the DMA unit and external memory have a maximum rate of one 16-bit transfer per system clock.

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 SDRAM pages, or while filling cache lines.
- Each direction change from `RX` to `TX` 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.

- 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, described in the next section.

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 3 SCLK cycles.

First, if either MDMA channel has been selected to use descriptors, the descriptors are fetched from memory. The destination channel descriptors are fetched first. Then, after a latency of 4 SCLK cycles after the last descriptor word is returned from memory (or typically 8 SCLK cycles after the fetch of the last descriptor word, due to memory pipelining), the source MDMA 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 2 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.

Table 5-5. Priority and Default Mapping of Peripheral to DMA

Priority	DMA Channel	PMAP Default Value	Peripheral Mapped by Default
Highest	DMA 0	0x0	PPI receive or transmit
	DMA 1	0x1	Ethernet MAC receive
	DMA 2	0x2	Ethernet MAC transmit
	DMA 3	0x3	SPORT0 receive
	DMA 4	0x4	SPORT0 transmit
	DMA 5	0x5	SPORT1 receive
	DMA 6	0x6	SPORT1 transmit
	DMA 7	0x7	SPI
	DMA 8	0x8	UART0 receive
	DMA 9	0x9	UART0 transmit
	DMA 10	0xA	UART1 receive
	DMA 11	0xB	UART1 transmit
	MDMA D0	N/A	N/A
	MDMA S0	N/A	N/A
	MDMA D1	N/A	N/A
Lowest	MDMA S1	N/A	N/A

As the Ethernet MAC module is not present on ADSP-BF534 processors, the `PMAP` field should not be set to 0x1 or 0x2 on used DMA channels. Attention is required as DMA 1 and DMA 2 channels default to these invalid values.

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.

Occasionally, instantaneous DMA traffic might exceed the available bandwidth, causing congestion. This may occur if L1 or external memory is temporarily stalled, perhaps for an SDRAM page swap or a cache line fill. Congestion might also occur if one or more DMA channels initiates a flurry of requests, perhaps for descriptor fetches or to fill a FIFO in the DMA or in the peripheral.

If congestion persists, lower priority DMA peripherals may become starved for data. Even though the peripheral's priority is low, if the necessary data transfer does not take place before the end of the peripheral's regular interval, system failure may result. To minimize this possibility, the DMA unit detects peripherals whose need for data has become urgent, and preferentially grants them service at the highest priority.

A DMA channel's request for memory service is defined as "urgent" if both:

- The channel's FIFO is not ready for a DAB bus transfer (that is, a transmit FIFO is empty or a receive FIFO is full), and
- The peripheral is asserting its DMA request line.

Descriptor fetches may be urgent, if they are necessary to initiate or continue a DMA work unit chain for a starving peripheral.

DMA requests from an MDMA channel 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 an urgent request will be granted. The urgent requests compete with each other, if there is more than one, and directional preference among urgent requests is observed.
- The resulting memory transfer is marked for expedited processing in the targeted memory system (L1 or external), and so are all prior incomplete memory transfers ahead of it in that memory system. This may cause a series of external memory core accesses to be delayed for a few cycles so that a peripheral's urgent request may be accommodated.

The preferential handling of urgent DMA transfers is completely automatic. No user controls are required for this function to operate.

Memory DMA Priority and Scheduling

All MDMA operations have lower precedence than any peripheral DMA operations. MDMA thus makes effective use of any memory bandwidth unused by peripheral DMA traffic.

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 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 DMA_TC_PER register (see "Static Channel Prioritization" on page 5-48).

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 <= P <= 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 will be granted (stream 0 in case of conflict), and that stream's selection is then "locked." The MDMA_ROUND_ROBIN_COUNT counter field in the DMA_TC_CNT register is loaded with the period P from MDMA_ROUND_ROBIN_PERIOD, and MDMA transfers begin. The counter is decremented on every data transfer (as each data word is written to memory). After the

transfer corresponding to a count of 1, the MDMA stream selection is passed automatically to the other stream with zero overhead, and the MDMA_ROUND_ROBIN_COUNT counter is reloaded with the period value P from MDMA_ROUND_ROBIN_PERIOD. In this cycle, if the other MDMA stream is ready to perform a transfer, the stream selection is locked on the new MDMA stream. If the other MDMA stream is not ready to perform a transfer, then no transfer is performed, and on the next cycle the stream selection unlocks and becomes free again.

If round robin operation is used when only one MDMA stream is active, one idle cycle will occur for each P MDMA data cycles, slightly lowering bandwidth by a factor of 1/(P+1). If both MDMA streams are used, however, memory DMA can operate continuously with zero additional overhead for alternation of streams (other than overhead cycles normally associated with reversal of read/write direction to memory, for example). By selection of various round robin period values P which limit how often the MDMA streams alternate, maximal transfer efficiency can be maintained.

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 via the DAB bus, and whose data FIFOs are ready to handle the transfer, compete with each other for DAB bus cycles. Similarly but separately, channels whose FIFOs need memory service (prefetch or post-write) compete together for access to the memory buses. MDMA streams compete for memory access as a unit, and source and destination may be granted together if their memory transfers do not conflict. In this way, internal-to-external or external-to-internal memory transfers may occur at the full system clock rate (SCLK). Examples of memory conflict include simultaneous access to the same memory space and simultaneous attempts to fetch descriptors. Special processing may occur if a peripheral

is requesting DMA but its FIFO is not ready (for example, an empty transmit FIFO or full receive FIFO). For more information, see "Temporary DMA Urgency" on page 5-50.

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 DMA_TC_PER and DMA_TC_CNT registers. This mechanism performs the optimization without real-time processor intervention, and without the need to program transfer bursts into the DMA work unit streams. Traffic can be independently controlled for each of the three buses (DAB, DCB, and DEB) with simple counters. In addition, alternation of transfers among MDMA streams can be controlled with the MDMA_ROUND_ROBIN_COUNT field of the DMA_TC_CNT register. See "Memory DMA Priority and Scheduling" on page 5-51.

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 `DMA_TC_PER` 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 5-9). 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 would first be 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 SDRAM to perform a page open operation which will take many system clock cycles. The DMA engine may then move on to another DMA operation by channel B which does not in itself incur latency, but will be stalled behind the slow operation by channel A. Software monitoring channel B could not safely conclude whether the memory location pointed to by 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 for an MDMA FIFO is eight locations (four 32-bit data elements). The DMA will 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 work unit has been completed and will definitely be visible to DSP code. For memory

read DMA, the final memory read data will have been safely received in the DMA's FIFO; for memory write DMA, the DMA unit will have received an acknowledge from L1 memory or the EBIU that the data has been 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 be signaled 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

- 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 being 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 will be taken from the DMAx_CONFIG value in the descriptor read from memory (and the field values initially written to the register are ignored). See "Initializing Descriptors in Memory" on page 5-111 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 will be received or what these requests might contain. The software could manage these transfers using a circular linked list of DMA descriptors, where each descriptor's 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 (I, L, M, and B 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 will be serviced separately (no interrupt overrun).

To maintain synchronization of the descriptor queue, the non-interrupt software maintains a count of descriptors added to the queue, while the interrupt handler maintains a count of completed descriptors removed from the queue. The counts are equal only when the DMA channel is paused after having processed all the descriptors.

When each new work request is received, the DMA manager software initializes a new descriptor, taking care to write a 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 (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 should write 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 would need 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 should queue 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 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 should leave 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 its processing by the DMA unit has been started, unless careful synchronization measures are taken. In the most straightforward implementation of a descriptor queue, the DMA manager software would never modify descriptors on the active queue; instead, the DMA manager waits until the DMA queue completion interrupt indicates the processing of the entire active queue is complete.

When a DMA queue completion interrupt is received, the interrupt handler reads the mailbox from the non-interrupt software and writes the value in it to the DMA channel's DMAx_CONFIG register. This single register write restarts the queue, effectively transforming the waiting queue to an

active queue. The interrupt handler should then pass a message back to the non-interrupt software indicating the location of the last descriptor accepted into the active queue. If, on the other hand, the interrupt handler reads its mailbox and finds a DMAx_CONFIG value of zero, indicating there is no more work to perform, then it should pass an appropriate message (for example, zero) back to the non-interrupt software indicating that the queue has stopped. This simple handler should be able to be coded in a very small number of instructions.

The non-interrupt software which accepts new DMA work requests needs to synchronize the activation of new work with the interrupt handler. If the queue has stopped (that is, if the mailbox from the interrupt software is zero), the non-interrupt software is responsible for starting the queue (writing the first descriptor's DMAx_CONFIG value to the channel's DMAx_CONFIG register). If the queue is not stopped, however, the non-interrupt software must not write the DMAx_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 has been stopped in FLOW = 0 mode, the DMA_RUN bit in the DMAx_IRQ_STATUS register remains set until the content of the internal DMA FIFOs has been completely processed. Once the DMA_RUN bit clears, it is safe to restart the DMA by simply writing again to the DMAx_CONFIG register. The DMA sequence is repeated with the previous settings.

Similarly, a descriptor-based DMA sequence that has been 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.
2. Write to DMAx_CONFIG with

   FLOW = 0x8
   NDSIZE >= 0xA
   DI_EN = 0
   DMAEN = 1.

3. Automatically fetched DMACFG 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 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.

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 5-40). 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 5-114. 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

DMA registers fall into three categories:

- DMA channel registers
- Handshaked MDMA registers
- Global DMA traffic control registers

DMA Channel Registers

The processor features twelve peripheral DMA channels and two channel pairs for memory DMA. All channels have an identical set of registers summarized in Table 5-6.

Table 5-6 lists the generic names of the DMA registers. For each register, the table also shows the MMR offset, a brief description of the register, the register category.

Table 5-6. Generic Names of the DMA Memory-mapped Registers

MMR Offset	Generic MMR Name	MMR Description	Register Category	Name of Corresponding Descriptor Element in Memory
0x00	NEXT_DESC_PTR	Link pointer to next descriptor	Parameter	NDPH (upper 16 bits), NDPL (lower 16 bits)
0x04	START_ADDR	Start address of current buffer	Parameter	SAH (upper 16 bits), SAL (lower 16 bits)
0x08	CONFIG	DMA Configuration register, including enable bit	Parameter	DMACFG
0x0C	Reserved	Reserved		
0x10	X_COUNT	Inner loop count	Parameter	XCNT
0x14	X_MODIFY	Inner loop address increment, in bytes	Parameter	XMOD
0x18	Y_COUNT	Outer loop count (2D only)	Parameter	YCNT
0x1C	Y_MODIFY	Outer loop address increment, in bytes	Parameter	YMOD

Table 5-6. Generic Names of the DMA Memory-mapped Registers (Cont'd)

MMR Offset	Generic MMR Name	MMR Description	Register Category	Name of Corresponding Descriptor Element in Memory
0x20	CURR_DESC_PTR	Current Descriptor Pointer	Current	N/A
0x24	CURR_ADDR	Current DMA Address	Current	N/A
0x28	IRQ_STATUS	Interrupt Status register: Contains Completion and DMA Error Interrupt status and channel state (Run/Fetch/Paused)	Control/ Status	N/A
0x2C	PERIPHERAL_MAP	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	CURR_X_COUNT	Current count (1D) or intra-row X count (2D); counts down from X_ COUNT	Current	N/A
0x34	Reserved	Reserved		
0x38	CURR_Y_COUNT	Current row count (2D only); counts down from Y_ COUNT	Current	N/A
0x3C	Reserved	Reserved		

Channel-specific register names are composed of a prefix and the generic MMR name shown in Table 5-6. For peripheral DMA channels, the prefix "DMAx_" is used where "x" stands for a channel number between 0 and 11. For memory DMA channels, the prefix is "MDMA_yy_", where

"yy" stands for either "D0", "S0", "D1", or "S1" to indicate destination and source channel registers of MDMA0 and MDMA1. For example, the configuration register of peripheral DMA channel 6 is called DMA6_CONFIG. The one for MDMA1 source channel is called MDMA_S1_CONFIG.

The generic MMR names shown in Table 5-6 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 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 5-6.
- 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. However, the following registers 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 5-6 shows the relation.

DMAx_PERIPHERAL_MAP/MDMA_yy_PERIPHERAL_MAP Registers

Each DMA channel's peripheral map register (DMAx_PERIPHERAL_MAP/MDMA_yy_PERIPHERAL_MAP, shown in Figure 5-6) contains 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. Make sure DMA is disabled on channels 6 and 7.
2. Write DMA6_PERIPHERAL_MAP with 0x7000 and DMA7_PERIPHERAL_MAP with 0x6000.
3. Enable DMA on channels 6 and/or 7.

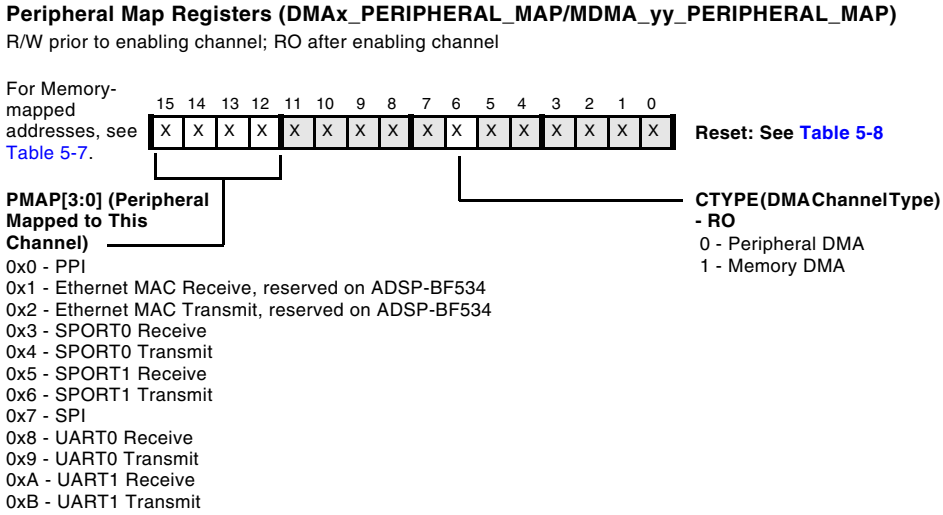


Figure 5-6. Peripheral Map Registers

Table 5-7. Peripheral Map Register Memory-mapped Addresses

Register Name	Memory-mapped Address
DMA0_PERIPHERAL_MAP	0xFFC0 0C2C
DMA1_PERIPHERAL_MAP	0xFFC0 0C6C
DMA2_PERIPHERAL_MAP	0xFFC0 0CAC
DMA3_PERIPHERAL_MAP	0xFFC0 0CEC
DMA4_PERIPHERAL_MAP	0xFFC0 0D2C
DMA5_PERIPHERAL_MAP	0xFFC0 0D6C
DMA6_PERIPHERAL_MAP	0xFFC0 0DAC
DMA7_PERIPHERAL_MAP	0xFFC0 0DEC
DMA8_PERIPHERAL_MAP	0xFFC0 0E2C
DMA9_PERIPHERAL_MAP	0xFFC0 0E6C
DMA10_PERIPHERAL_MAP	0xFFC0 0EAC
DMA11_PERIPHERAL_MAP	0xFFC0 0EEC
MDMA_D0_PERIPHERAL_MAP	0xFFC0 0F2C
MDMA_S0_PERIPHERAL_MAP	0xFFC0 0F6C
MDMA_D1_PERIPHERAL_MAP	0xFFC0 0FAC
MDMA_S1_PERIPHERAL_MAP	0xFFC0 0FEC

Table 5-8 lists the binary peripheral map settings for each DMA-capable peripheral.

Table 5-8. Peripheral Mapping

DMA Channel	Default Peripheral Mapping	Default PERIPHERAL_MAP Setting (Binary)	Comments
DMA0 (highest priority)	PPI receive/transmit	b#0000 0000 0000 0000	
DMA1	Ethernet receive	b#0001 0000 0000 0000	Invalid PMAP default setting on ADSP-BF534
DMA2	Ethernet transmit	b#0010 0000 0000 0000	Invalid PMAP default setting on ADSP-BF534
DMA3	SPORT0 receive	b#0011 0000 0000 0000	
DMA4	SPORT0 transmit	b#0100 0000 0000 0000	
DMA5	SPORT1 receive	b#0101 0000 0000 0000	
DMA6	SPORT1 transmit	b#0110 0000 0000 0000	
DMA7	SPI receive/transmit	b#0111 0000 0000 0000	
DMA8	UART0 receive	b#1000 0000 0000 0000	
DMA9	UART0 transmit	b#1001 0000 0000 0000	
DMA10	UART1 receive	b#1010 0000 0000 0000	
DMA11	UART1 transmit	b#1011 0000 0000 0000	
MDMA_D0	MDMA0 destination	b#0000 0000 0100 0000	Not reassignable
MDMA_S0	MDMA0 source	b#0000 0000 0100 0000	Not reassignable
MDMA_D1	MDMA1 destination	b#0000 0000 0100 0000	Not reassignable
MDMA_S1 (lowest priority)	MDMA1 source	b#0000 0000 0100 0000	Not reassignable

DMAx_CONFIG/MDMA_yy_CONFIG Registers

The DMA configuration register (DMAx_CONFIG/MDMA_yy_CONFIG), shown in Figure 5-7, is used to set up DMA parameters and operating modes. Note that writing the DMAx_CONFIG register while DMA is already running will cause a DMA error unless writing with the DMAEN bit set to 0.

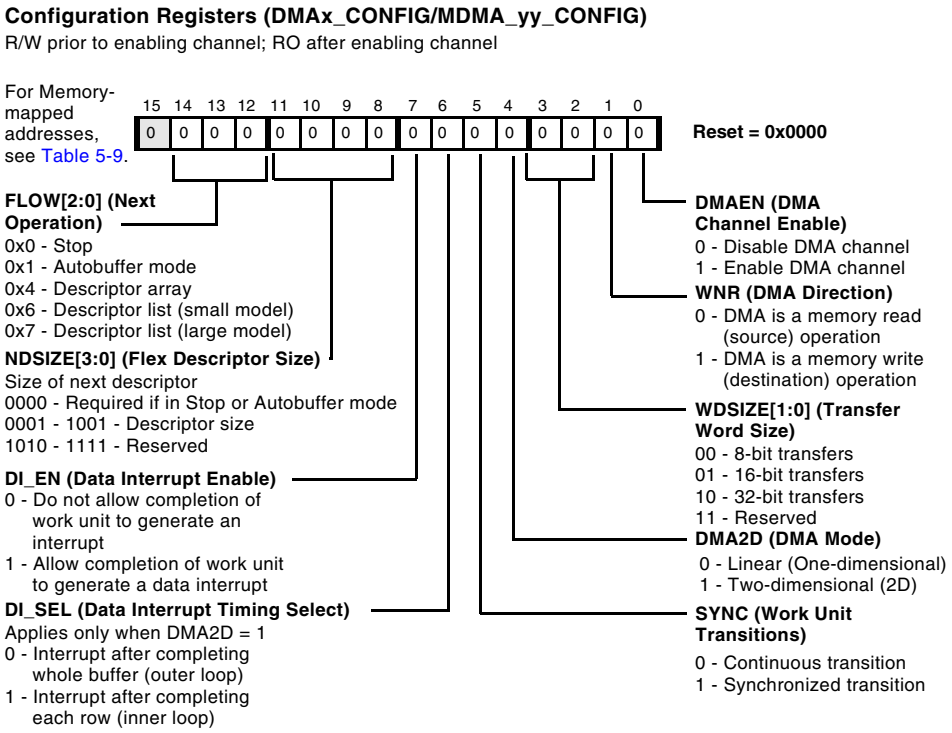


Figure 5-7. Configuration Registers

Table 5-9. 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
MDMA_D0_CONFIG	0xFFC0 0F08
MDMA_S0_CONFIG	0xFFC0 0F48
MDMA_D1_CONFIG	0xFFC0 0F88
MDMA_S1_CONFIG	0xFFC0 0FC8

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 DMAx_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 DMAx_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 DMAx_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 5-27.

  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 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.

- DMA2D (DMA mode). This bit specifies whether DMA mode involves only DMAx_X_COUNT and DMAx_X_MODIFY (one-dimensional DMA) or also involves DMAx_Y_COUNT and DMAx_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.

DMAx_IRQ_STATUS/MDMA_yy_IRQ_STATUS Registers

The interrupt status register (DMAx_IRQ_STATUS/MDMA_yy_IRQ_STATUS), shown in Figure 5-8, contains bits that record whether the DMA channel:

- Is enabled and operating, enabled but stopped, or disabled.
- Is fetching data or a DMA descriptor.
- Has detected that a global DMA interrupt or a channel interrupt is being asserted.
- Has logged occurrence of a DMA error.

Note the DMA_DONE interrupt is asserted when the last memory access (read or write) has completed.

For a 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 has been 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.

Interrupt Status Registers (DMAx_IRQ_STATUS/MDMA_yy_IRQ_STATUS)

For Memory-mapped addresses, see Table 5-10.

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

Reset = 0x0000

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 5-8. Interrupt Status Registers

Table 5-10. 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
MDMA_D0_IRQ_STATUS	0xFFC0 0F28
MDMA_S0_IRQ_STATUS	0xFFC0 0F68
MDMA_D1_IRQ_STATUS	0xFFC0 0FA8
MDMA_S1_IRQ_STATUS	0xFFC0 0FE8

The processor supports a flexible interrupt control structure with three interrupt sources:

- Data driven interrupts (see Table 5-11)
- 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 5-11. 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.
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 ORed 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.

Note the `DMA_DONE` and `DMA_ERR` interrupt indicators are write-one-to-clear (W1C).

When switching a peripheral from DMA to non-DMA mode, the peripheral's interrupts should be disabled during the mode switch (via the appropriate peripheral registers or `SIC_IMASK`) so that no unintended interrupt is generated on the shared DMA/interrupt request line.

DMAx_START_ADDR/MDMA_yy_START_ADDR Registers

The start address register (DMAx_START_ADDR/MDMA_yy_START_ADDR), shown in Figure 5-9, contains the start address of the data buffer currently targeted for DMA.

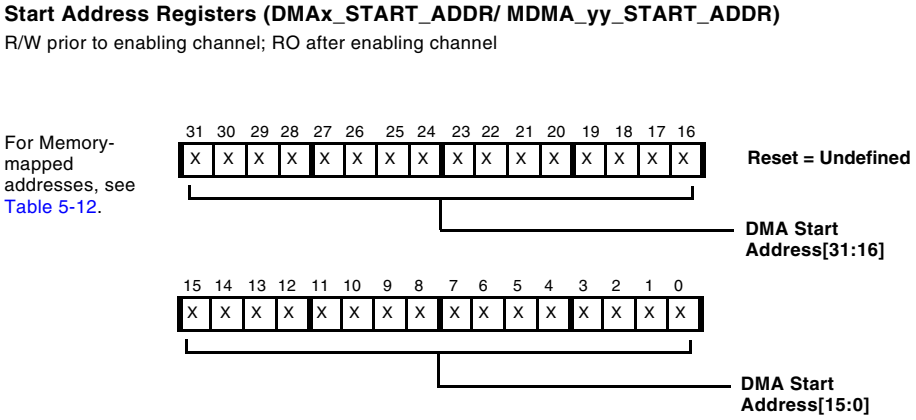


Figure 5-9. Start Address Registers

Table 5-12. 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

Table 5-12. Start Address Register Memory-mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
DMA8_START_ADDR	0xFFC0 0E04
DMA9_START_ADDR	0xFFC0 0E44
DMA10_START_ADDR	0xFFC0 0E84
DMA11_START_ADDR	0xFFC0 0EC4
MDMA_D0_START_ADDR	0xFFC0 0F04
MDMA_S0_START_ADDR	0xFFC0 0F44
MDMA_D1_START_ADDR	0xFFC0 0F84
MDMA_S1_START_ADDR	0xFFC0 0FC4

DMAx_CURR_ADDR/MDMA_yy_CURR_ADDR Registers

The current address register (DMAx_CURR_ADDR/MDMA_yy_CURR_ADDR), shown in Figure 5-10, contains the present DMA transfer address for a given DMA session. On the first memory 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.

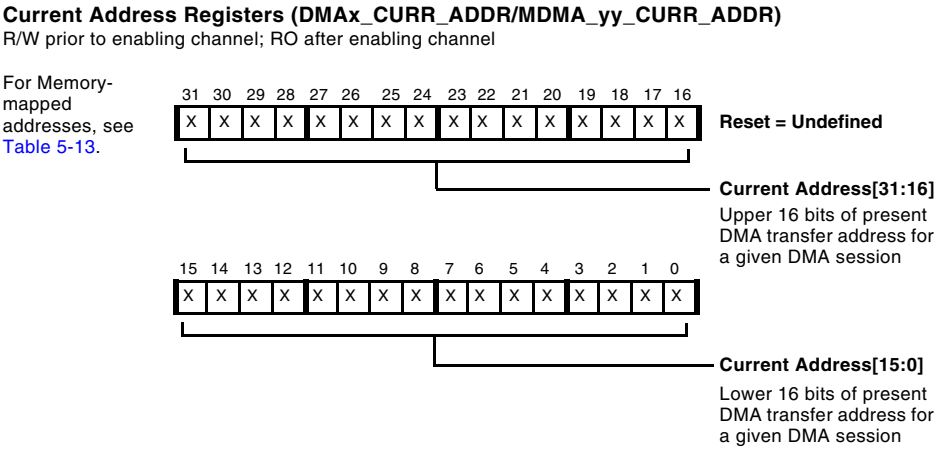


Figure 5-10. Current Address Registers

Table 5-13. 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

Table 5-13. Current Address Register Memory-mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
MDMA_D0_CURR_ADDR	0xFFC0 0F24
MDMA_S0_CURR_ADDR	0xFFC0 0F64
MDMA_D1_CURR_ADDR	0xFFC0 0FA4
MDMA_S1_CURR_ADDR	0xFFC0 0FE4

DMAx_X_COUNT/MDMA_yy_X_COUNT Registers

For 2D DMA, the inner loop count register (DMAx_X_COUNT/MDMA_yy_X_COUNT), shown in Figure 5-11, contains 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 5-14. A value of 0 in DMAx_X_COUNT corresponds to 65,536 elements.

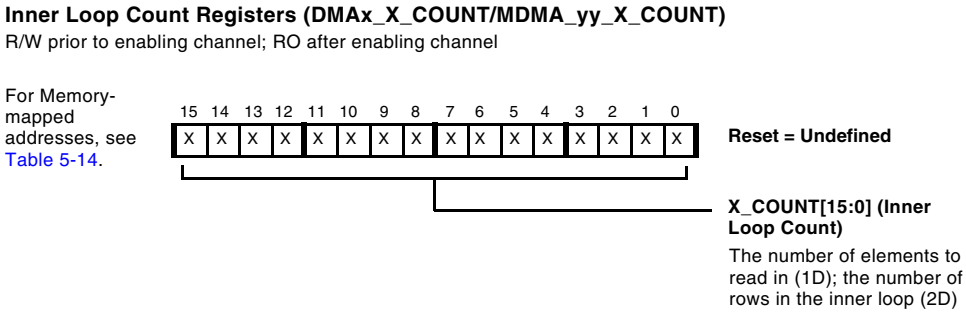


Figure 5-11. Inner Loop Count Registers

Table 5-14. 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
MDMA_D0_X_COUNT	0xFFC0 0F10
MDMA_S0_X_COUNT	0xFFC0 0F50
MDMA_D1_X_COUNT	0xFFC0 0F90
MDMA_S1_X_COUNT	0xFFC0 0FD0

DMAx_CURR_X_COUNT/MDMA_yy_CURR_X_COUNT Registers

The current inner loop count register (`DMAx_CURR_X_COUNT/MDMA_yy_CURR_X_COUNT`), shown in Figure 5-12, holds 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.

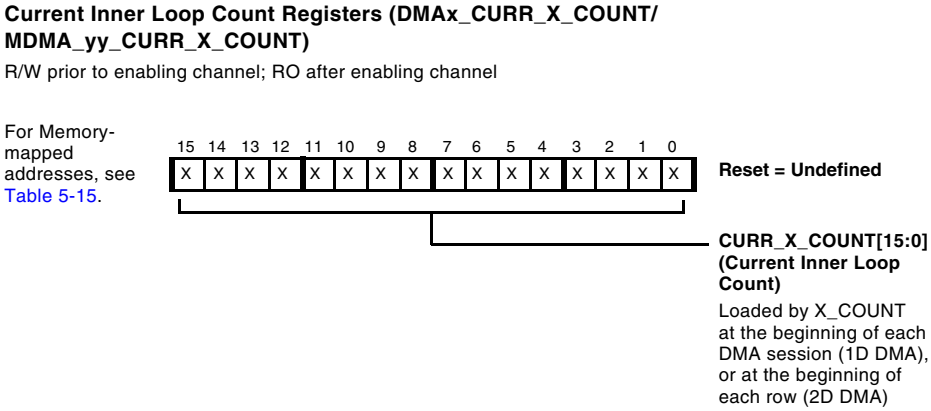


Figure 5-12. Current Inner Loop Count Registers

Table 5-15. 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

Table 5-15. Current Inner Loop Count Register Memory-mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
DMA9_CURR_X_COUNT	0xFFC0 0E70
DMA10_CURR_X_COUNT	0xFFC0 0EB0
DMA11_CURR_X_COUNT	0xFFC0 0EF0
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

DMAx_X_MODIFY/MDMA_yy_X_MODIFY Registers

The inner loop address increment register (`DMAx_X_MODIFY/MDMA_yy_X_MODIFY`), shown in Figure 5-13, contains a signed, two's-complement byte-address increment. In 1D DMA, this increment is the stride that is applied after transferring each element.

Note `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.

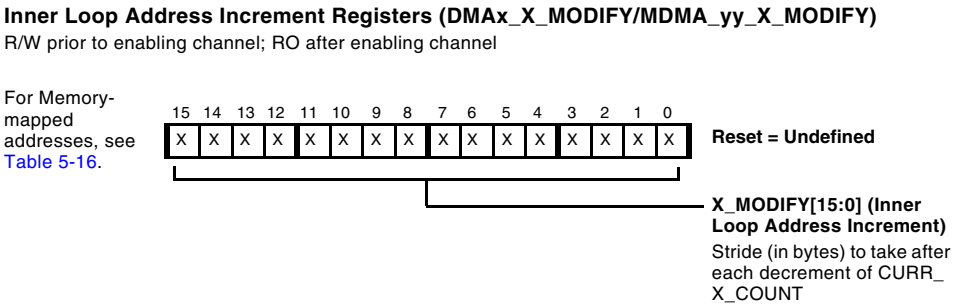


Figure 5-13. Inner Loop Address Increment Registers

Table 5-16. 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

Table 5-16. Inner Loop Address Increment Register Memory-mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
DMA11_X_MODIFY	0xFFC0 0ED4
MDMA_D0_X_MODIFY	0xFFC0 0F14
MDMA_S0_X_MODIFY	0xFFC0 0F54
MDMA_D1_X_MODIFY	0xFFC0 0F94
MDMA_S1_X_MODIFY	0xFFC0 0FD4

DMAx_Y_COUNT/MDMA_yy_Y_COUNT Registers

For 2D DMA, the outer loop count register (`DMAx_Y_COUNT/MDMA_yy_Y_COUNT`), shown in Figure 5-14, contains 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 5-14.

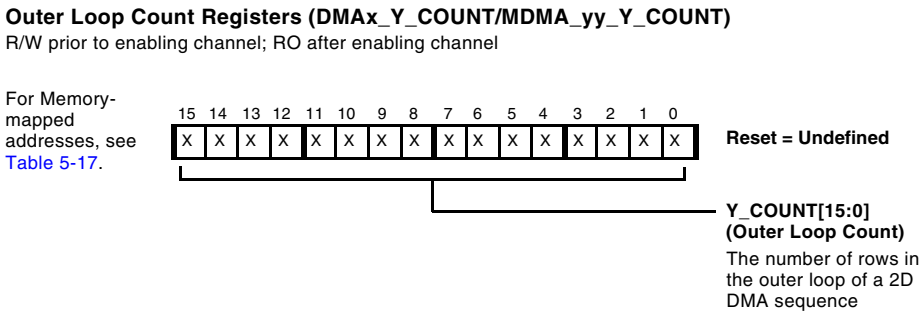


Figure 5-14. Outer Loop Count Registers

Table 5-17. 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
MDMA_D0_Y_COUNT	0xFFC0 0F18
MDMA_S0_Y_COUNT	0xFFC0 0F58
MDMA_D1_Y_COUNT	0xFFC0 0F98
MDMA_S1_Y_COUNT	0xFFC0 0FD8

DMAx_CURR_Y_COUNT/MDMA_yy_CURR_Y_COUNT Registers

The current outer loop count register (`DMAx_CURR_Y_COUNT/MDMA_yy_CURR_Y_COUNT`), used only in 2D mode, holds the number of full or partial rows (outer loops) remaining in the current work unit. See Figure 5-15. 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.

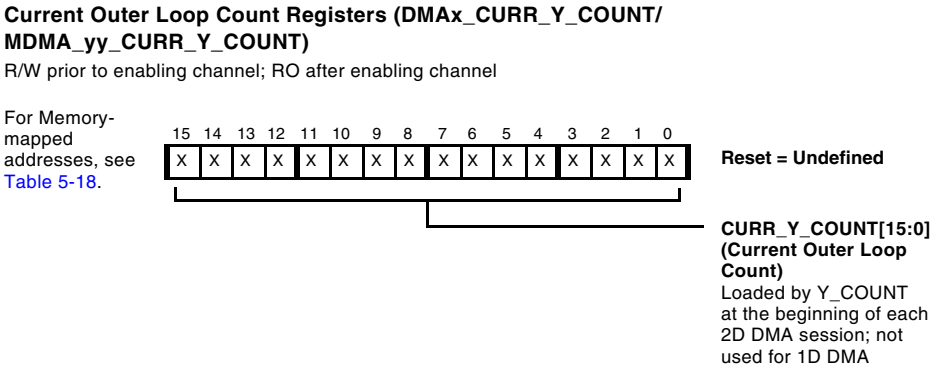


Figure 5-15. Current Outer Loop Count Registers

Table 5-18. 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

Table 5-18. Current Outer Loop Count Register Memory-mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
DMA11_CURR_Y_COUNT	0xFFC0 0EF8
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

DMAx_Y_MODIFY/MDMA_yy_Y_MODIFY Registers

The outer loop address increment register (`DMAx_Y_MODIFY/MDMA_yy_Y_MODIFY`) contains a signed, two's-complement value. See Figure 5-16. 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 5-14.

Note `DMAx_Y_MODIFY` is specified in bytes, regardless of the DMA transfer size.

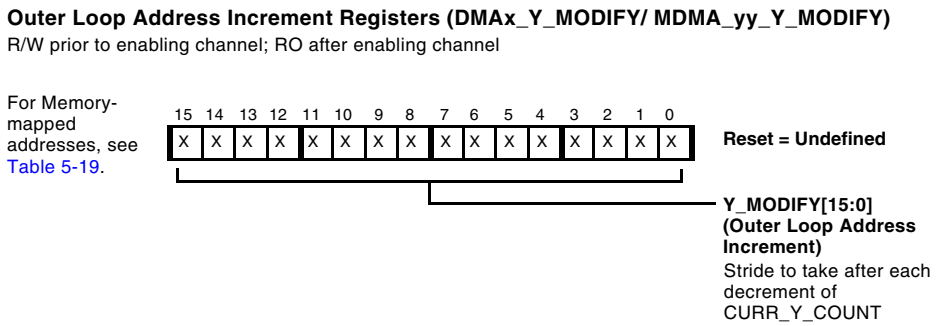


Figure 5-16. Outer Loop Address Increment Registers

Table 5-19. 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
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
MDMA_D0_Y_MODIFY	0xFFC0 0F1C
MDMA_S0_Y_MODIFY	0xFFC0 0F5C
MDMA_D1_Y_MODIFY	0xFFC0 0F9C
MDMA_S1_Y_MODIFY	0xFFC0 0FDC

DMAx_NEXT_DESC_PTR/MDMA_yy_NEXT_DESC_PTR Registers

The next descriptor pointer register (`DMAx_NEXT_DESC_PTR/MDMA_yy_NEXT_DESC_PTR`), shown in Figure 5-17, specifies where to look for the start of the next descriptor block when the DMA activity specified by the current descriptor block finishes. This register is used 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 via MMR access before starting DMA operation.

In descriptor array mode, the next descriptor pointer register is disregarded, and fetching is controlled only by the DMAx_CURR_DESC_PTR register.

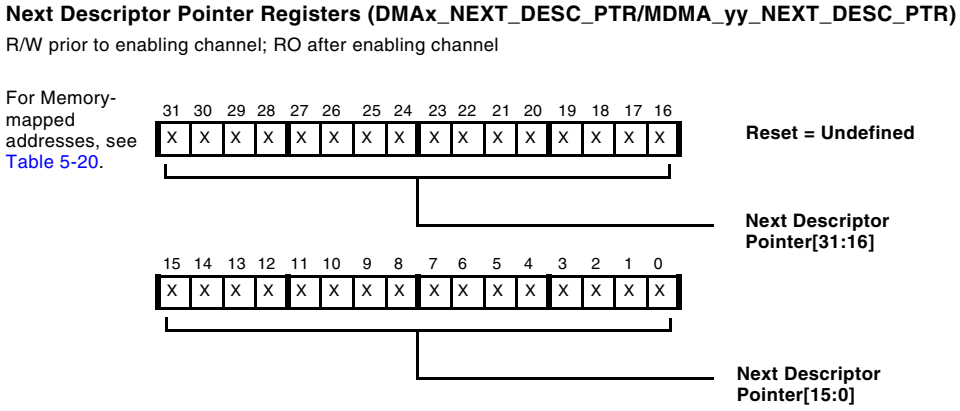


Figure 5-17. Next Descriptor Pointer Registers

Table 5-20. 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

Table 5-20. Next Descriptor Pointer Register Memory-mapped Addresses

Register Name	Memory-mapped Address
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
DMA9_NEXT_DESC_PTR	0xFFC0 0E40
DMA10_NEXT_DESC_PTR	0xFFC0 0E80
DMA11_NEXT_DESC_PTR	0xFFC0 0EC0
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

DMAx_CURR_DESC_PTR/MDMA_yy_CURR_DESC_PTR Registers

The current descriptor pointer register (`DMAx_CURR_DESC_PTR/MDMA_yy_CURR_DESC_PTR`), shown in Figure 5-18, contains 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 has been 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.

Current Descriptor Pointer Registers (DMAx_CURR_DESC_PTR/ MDMA_yy_CURR_DESC_PTR)

R/W prior to enabling channel; RO after enabling channel

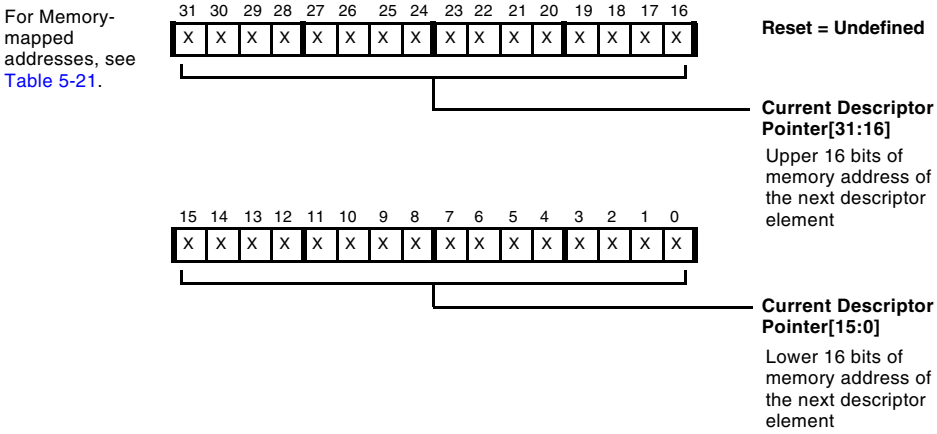


Figure 5-18. Current Descriptor Pointer Registers

Table 5-21. 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

Table 5-21. Current Descriptor Pointer Register Memory-mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
DMA7_CURR_DESC_PTR	0xFFC0 0DE0
DMA8_CURR_DESC_PTR	0xFFC0 0E20
DMA9_CURR_DESC_PTR	0xFFC0 0E60
DMA10_CURR_DESC_PTR	0xFFC0 0EA0
DMA11_CURR_DESC_PTR	0xFFC0 0EE0
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

HMDMA Registers

The processor features two HMDMA blocks. HMDMA0 is associated with MDMA0, and HMDMA1 is associated with MDMA1. Table 5-22 lists the naming conventions for these registers.

Table 5-22. Naming Conventions for Handshake MDMA Registers

Handshake MDMA MMR Name (x = 0 or 1)
HMDMAx_CONTROL
HMDMAx_BCINIT
HMDMAx_BCOUNT
HMDMAx_ECOUNT
HMDMAx_ECINIT

Table 5-22. Naming Conventions for Handshake MDMA Registers

Handshake MDMA MMR Name (x = 0 or 1)
HMDMAx_ECURGENT
HMDMAx_ECOVERFLOW

HMDMAx_CONTROL Registers

The handshake MDMA control register (`HMDMAx_CONTROL`), shown in Figure 5-19, is used to 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 5-23).

Table 5-23. DRQ[1:0] Values

DRQ[1:0]	Priority	Description
00	Disabled	The MDMA request is disabled.
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.
10	Enabled/M	Normal MDMA channel functionality and priority. Request multiple transfers from MDMA channel (default).
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.

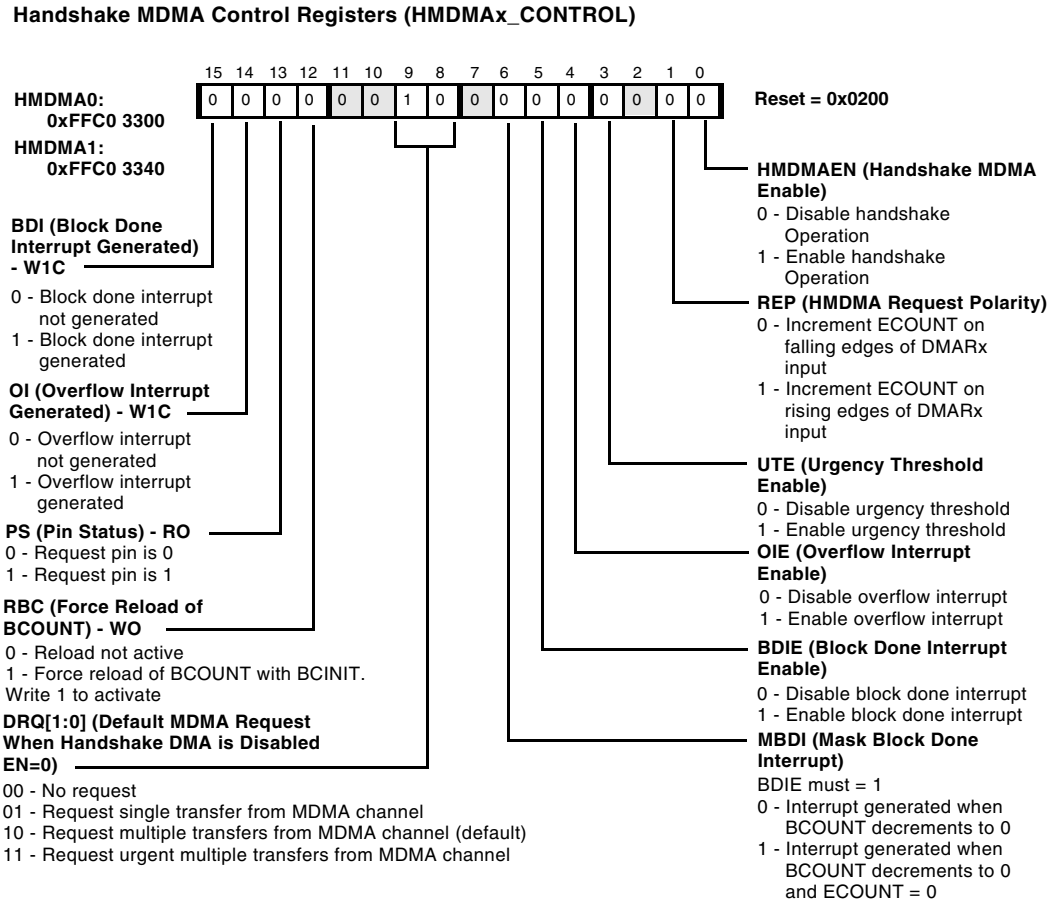


Figure 5-19. Handshake MDMA Control Registers

HMDMAx_BCINIT Registers

The handshake MDMA initial block count register (HMDMAx_BCINIT), shown in Figure 5-20, holds the number of transfers to do per edge of the DMARx control signal.

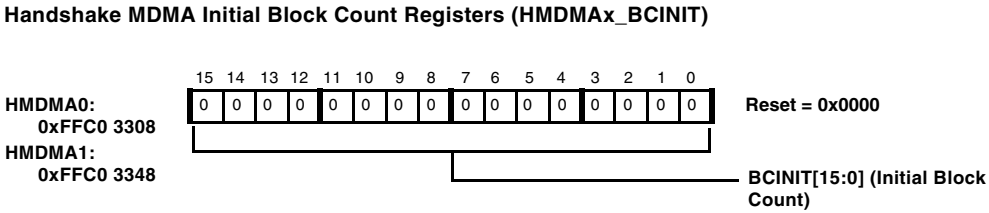


Figure 5-20. Handshake MDMA Initial Block Count Registers

HMDMAx_BCOUNT Registers

The handshake MDMA current block count register (HMDMAx_BCOUNT), shown in Figure 5-21, holds the number of transfers remaining for the current edge. MDMA requests are generated if this count is greater than 0.

Examples:

- 0000 = 0 transfers remaining
- FFFF = 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.

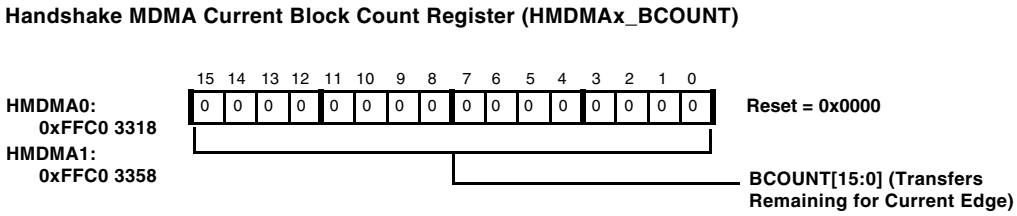


Figure 5-21. Handshake MDMA Current Block Count Registers

HMDMAx_ECOUNT Registers

The handshake MDMA current edge count register (HMDMAx_ECOUNT), shown in Figure 5-22, holds 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 positively or negatively signed number. Examples:

- 7FFF = 32767 edges remaining
- 0000 = 0 edges remaining
- 8000 = –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.

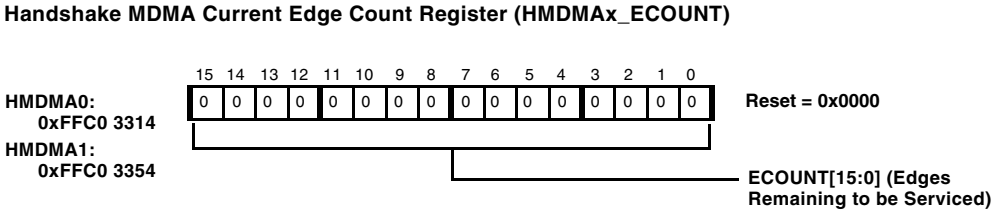


Figure 5-22. Handshake MDMA Current Edge Count Registers

HMDMAx_ECINIT Registers

The handshake MDMA initial edge count register (HMDMAx_ECINIT), shown in Figure 5-23, holds a signed number that is loaded into current edge count (HMDMAx_ECOUNT) when handshake DMA is enabled. This number is in a signed two's complement representation.

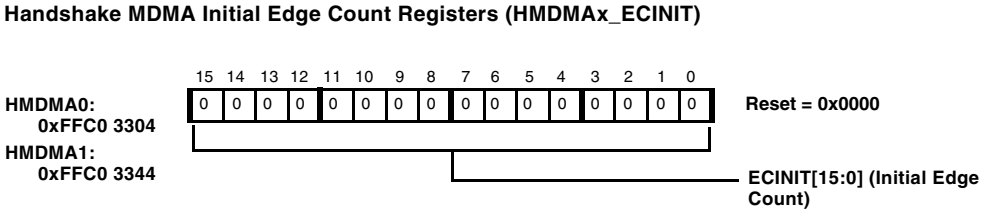


Figure 5-23. Handshake MDMA Initial Edge Count Registers

HMDMAx_ECURGENT Registers

The handshake MDMA edge count urgent register (HMDMAx_ECURGENT), shown in Figure 5-24, holds 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.

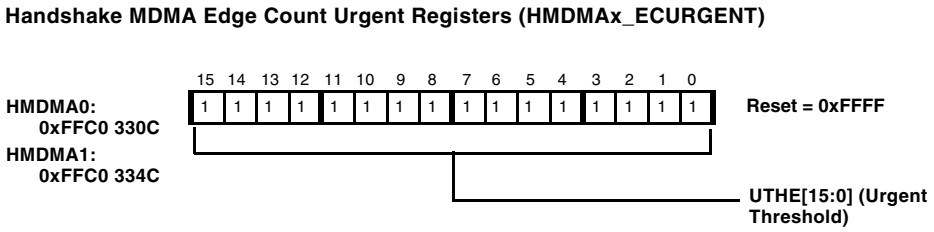


Figure 5-24. Handshake MDMA Edge Count Urgent Registers

HMDMAx_ECOVERFLOW Registers

The handshake MDMA edge count overflow interrupt register (HMDMAx_ECOVERFLOW), shown in Figure 5-25, holds the interrupt threshold. If the ECOUNT field in the handshake MDMA edge count register is greater than this threshold, an overflow interrupt is generated.

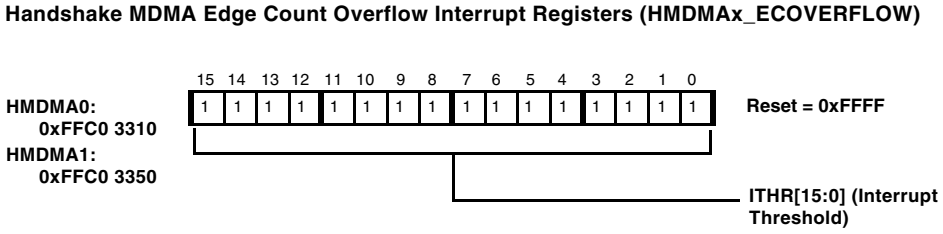


Figure 5-25. Handshake MDMA Edge Count Overflow Interrupt Registers

DMA Traffic Control Registers

The DMA_TC_PER register (see Figure 5-26) and the DMA_TC_CNT register (see Figure 5-27) work with other DMA registers to define traffic control.

DMA_TC_PER Register

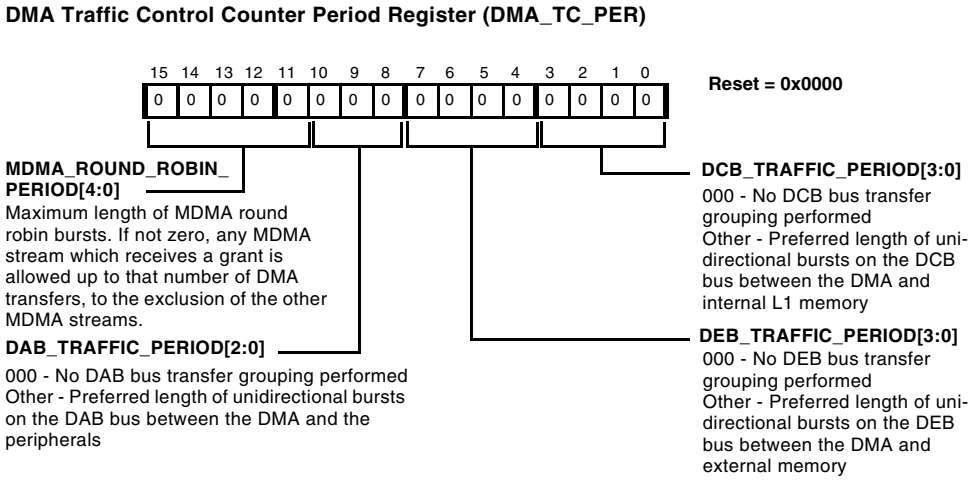


Figure 5-26. DMA Traffic Control Counter Period Register

DMA_TC_CNT Register

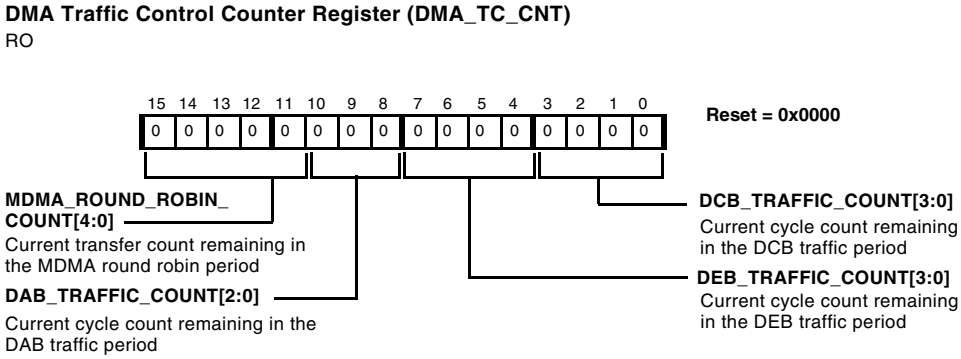


Figure 5-27. DMA Traffic Control Counter Register

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 DMA_TC_PER 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 DMA_TC_PER 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 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 DMA_TC_PER 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 DMA_TC_PER 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.

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.

Register-Based 2D Memory DMA

Listing 5-1 shows a register-based, two-dimensional MDMA. While the source channel processes linearly, the destination channel resorts elements by mirroring the two-dimensional data array. See Figure 5-28.

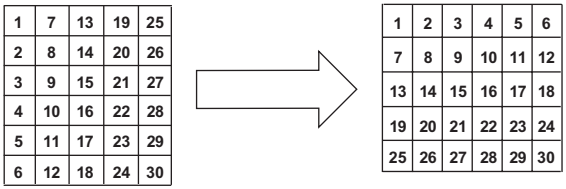


Figure 5-28. DMA Example, 2D Array

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 and SDRAM. For the case where the destination array resided in 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 2D Memory DMA

```
#include <defBF537.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_S0_CONFIG);
    p0.h = hi(MDMA_S0_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_S0_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. Two-Dimensional 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_S0_START_ADDR - MDMA_S0_CONFIG] = r7;
    r7.l = 2;
    w[p0 + MDMA_S0_X_MODIFY - MDMA_S0_CONFIG] = r7;
    r7.l = X * Y;
    w[p0 + MDMA_S0_X_COUNT - MDMA_S0_CONFIG] = r7;
    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 was 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 tool set 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.

Listing 5-4. Two Descriptors in Small List Flow Mode

```
.section sdram;
.byte2 arrBlock1[0x400];
.byte2 arrBlock2[0x800];

.section L1_data_a;
.byte2 descBlock1 = lo(descBlock2);
.var   descBlock1.addr = arrBlock1;
.byte2 descBlock1.cfg = FLOW_SMALL|NDSIZE_5|WDSIZE_16|DMAEN;
.byte2 descBlock1.len = length(arrBlock1);
       descBlock1.end:

.byte2 descBlock2 = lo(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_DESCRIPTORS__
#define __INCLUDE_DESCRIPTORS__
#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_DESCRIPTORS__
```

Note that near pointers are not natively supported by the C language and, thus, 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,
    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, thus, 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;
_main:
/* write descriptor address to next descriptor pointer */
    p0.h = hi(MDMA_S0_CONFIG);
    p0.l = lo(MDMA_S0_CONFIG);
    r0.h = hi(descDest1);
    r0.l = lo(descDest1);
    [p0 + MDMA_D0_NEXT_DESC_PTR - MDMA_S0_CONFIG] = r0;
    r0.h = hi(descSource1);
    r0.l = lo(descSource1);
    [p0 + MDMA_S0_NEXT_DESC_PTR - MDMA_S0_CONFIG] = r0;

/* start first work unit */
    r6.l = FLOW_LARGE|NDSIZE_7|WDSIZE_16|DMAEN;
```

```
    w[p0 + MDMA_S0_CONFIG - MDMA_S0_CONFIG] = r6;
    r7.l = FLOW_LARGE|NDSIZE_7|WDSIZE_16|WNR|DMAEN;
    w[p0 + MDMA_D0_CONFIG - MDMA_S0_CONFIG] = r7;

/* wait until destination channel has finished and W1C latch */
_main.wait:
    r0 = w[p0 + MDMA_D0_IRQ_STATUS - MDMA_S0_CONFIG] (z);
    CC = bittst (r0, bitpos(DMA_DONE));
    if !CC jump _main.wait;
    r0.l = DMA_DONE;
    w[p0 + MDMA_D0_IRQ_STATUS - MDMA_S0_CONFIG] = r0;

/* wait for any software or hardware event here */

/* start next work unit */
    w[p0 + MDMA_S0_CONFIG - MDMA_S0_CONFIG] = r6;
    w[p0 + MDMA_D0_CONFIG - MDMA_S0_CONFIG] = r7;
    jump _main.wait;
_main.end:
```

Handshaked Memory DMA Example

The functional block for the handshaked 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(PORTF_FER);
    r0.l = PF1;
    w[p1] = r0;
    p1.l = lo(PORT_MUX);
    r0.l = PFDE;
    w[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);
    r0.l = 0;
    w[p1] = r0;

/* enable for rising edges */
    p1.l = lo(HMDMA1_CONTROL);
    r2.l = REP | HMDMAEN;
    w[p1] = r2;
```

If the HMDMA is intended to copy from internal memory to external devices the above setup is sufficient. If, however, the data flow is from outside the processor to internal memory, then this small issue must be considered—the HMDMA only controls the destination channel of the

memory DMA. It does not gate requests to the source channel at all. Thus, as soon as the source channel is enabled it starts filling the DMA FIFO immediately. In 16-bit DMA mode this results in eight read strobes on the EBIU even before the first DMAR1 event has been detected. In other words, the transferred data and the DMAR1 strobes are eight positions off. The example in Listing 5-9 delays processing until eight DMAR1 requests have been received. Note that doing so the transmitter is required to add eight trailing dummy writes after all data words have been sent. This is because the transmit channel still has to drain the DMA FIFO.

Listing 5-9. HMDMA With Delayed Processing

```
/* wait for eight requests */
    p1.l = lo(HMDMA1_ECOUNT);
    r0 = 7 (z);
initial_requests:
    r1 = w[p1] (z);
    CC = r1 < r0;
    if CC jump initial_requests;

/* disable and reenable to clear edge count */
    p1.l = lo(HMDMA1_CONTROL);
    r0.l = 0;
    w[p1] = r0;
    w[p1] = r2;
```

If the polling operation as shown in Listing 5-9 is too expensive, an interrupt version of it can be implemented by using the HMDMA overflow feature. Set the `HMDMAx_OVERFLOW` register to eight temporarily.

6 EXTERNAL BUS INTERFACE UNIT

The External Bus Interface Unit (EBIU) provides glueless interfaces to external memories. The processor supports Synchronous DRAM (SDRAM) including mobile SDRAM, and is compliant with the PC100 and PC133 SDRAM standards. The EBIU also supports asynchronous interfaces such as SRAM, ROM, FIFOs, flash memory, and ASIC/FPGA designs.

EBIU Overview

The EBIU services requests for external memory from the core or from a DMA channel. The priority of the requests is determined by the external bus controller. The address of the request determines whether the request is serviced by the EBIU SDRAM controller or the EBIU asynchronous memory controller.

The DMA controller provides high-bandwidth data movement capability. The Memory DMA (MDMA) channels can perform block transfers of code or data between the internal memory and the external memory spaces. The MDMA channels also feature a Handshake Operation mode (HMDMA) via dual external DMA request pins. When used in conjunction with the EBIU, this functionality can be used to interface high-speed external devices, such as FIFOs and USB 2.0 controllers, in an automatic manner. For more information on HMDMA and the external DMA request pins, please refer to Chapter 5, “Direct Memory Access”.

The EBIU is clocked by the system clock (SCLK). All synchronous memories interfaced to the processor operate at the SCLK frequency. The ratio between core frequency and SCLK frequency is programmable using a Phase Locked Loop (PLL) system Memory-Mapped Register (MMR). For more information, see "Core Clock/System Clock Ratio Control" on page 20-5.

The external memory space is shown in Figure 6-1. One memory region is dedicated to SDRAM support. SDRAM interface timing and the size of the SDRAM region are programmable. The SDRAM memory space can range in size from 16M byte to 512M byte.

The start address of the SDRAM memory space is 0x0000 0000. The area from the end of the SDRAM memory space up to address 0x2000 0000 is reserved.

The next four regions are dedicated to supporting asynchronous memories. Each asynchronous memory region can be independently programmed to support different memory device characteristics. Each region has its own memory select output pin from the EBIU.

The next region is reserved memory space. References to this region do not generate external bus transactions. Writes have no effect on external memory values, and reads return undefined values. The EBIU generates an error response on the internal bus, which will generate a hardware exception for a core access or will optionally generate an interrupt from a DMA channel.

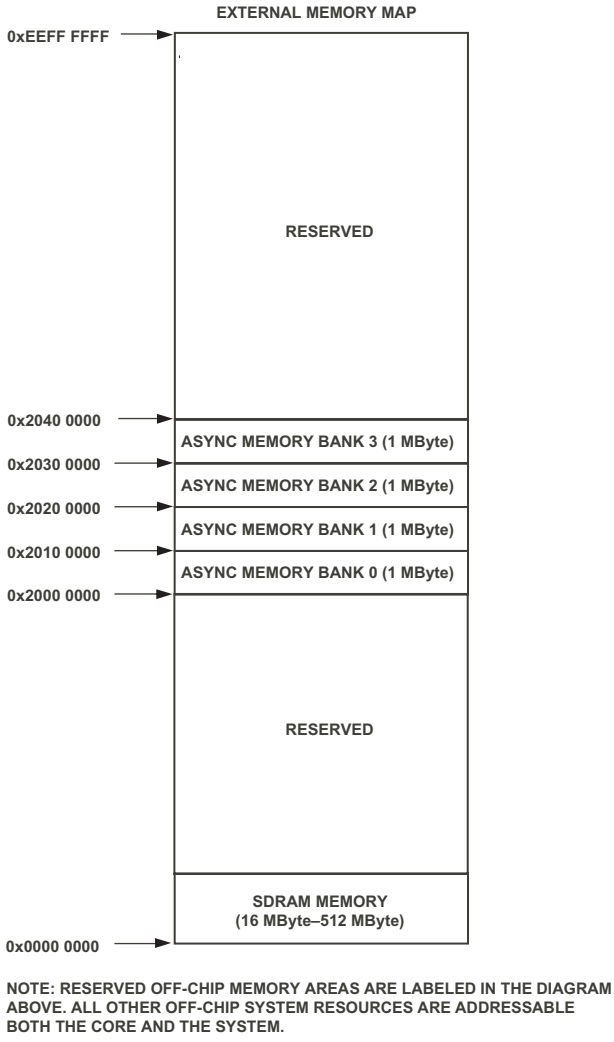


Figure 6-1. External Memory Map

Block Diagram

Figure 6-2 is a conceptual block diagram of the EBIU and its interfaces. Signal names shown with an overbar are active low signals.

Since only one external memory device can be accessed at a time, control, address, and data pins for each memory type are multiplexed together at the pins of the device. The Asynchronous Memory Controller (AMC) and the SDRAM Controller (SDC) effectively arbitrate for the shared pin resources.

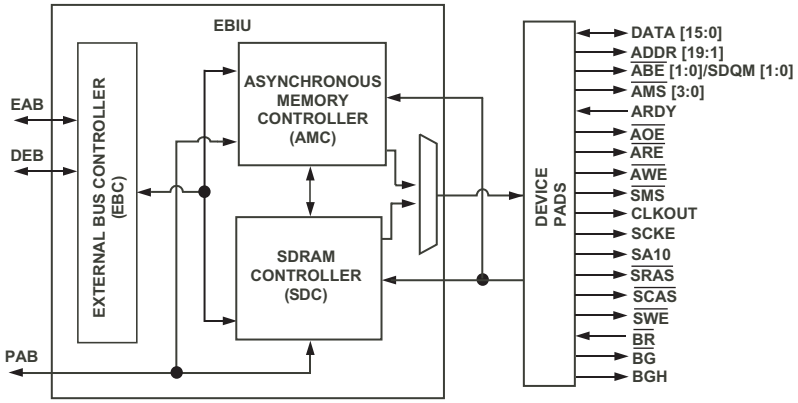


Figure 6-2. External Bus Interface Unit (EBIU)

Internal Memory Interfaces

The EBIU functions as a slave on three buses internal to the processor:

- External Access Bus (EAB), mastered by the core memory management unit on behalf of external bus requests from the core
- DMA External Bus (DEB), mastered by the DMA controller on behalf of external bus requests from any DMA channel
- Peripheral Access Bus (PAB), mastered by the core on behalf of system MMR requests from the core

These are synchronous interfaces, clocked by `SCLK`, as are the EBIU and pads registers. The EAB provides access to both asynchronous external memory and synchronous DRAM external memory. The external access is controlled by either the AMC or the SDC, depending on the internal address used to access the EBIU. Since the AMC and SDC share the same interface to the external pins, access is sequential and must be arbitrated based on requests from the EAB.

The third bus (PAB) is used only to access the memory-mapped control and status registers of the EBIU. The PAB connects separately to the AMC and SDC; it does not need to arbitrate with or take access cycles from the EAB bus.

The External Bus Controller (EBC) logic must arbitrate access requests for external memory coming from the EAB and DEB buses. The EBC logic routes read and write requests to the appropriate memory controller based on the bus selects. The AMC and SDC compete for access to the shared resources. This competition is resolved in a pipelined fashion, in the order dictated by the EBC arbiter. Transactions from the core have priority over DMA accesses in most circumstances. However, if the DMA controller detects an excessive backup of transactions, it can request its priority to be temporarily raised above the core.

Registers

There are six control registers and one status register in the EBIU. They are:

- Asynchronous memory global control register (EBIU_AMGCTL)
- Asynchronous memory bank control 0 register (EBIU_AMBCTL0)
- Asynchronous memory bank control 1 register (EBIU_AMBCTL1)
- SDRAM memory global control register (EBIU_SDGCTL)
- SDRAM memory bank control register (EBIU_SDBCTL)
- SDRAM refresh rate control register (EBIU_SDRRC)
- SDRAM control status register (EBIU_SDSTAT)

Each of these registers is described in detail in the AMC and SDC sections later in this chapter.

Shared Pins

Both the AMC and the SDC share the external interface address and data pins, as well as some of the control signals. These pins are shared:

- ADDR[19:1], address bus
- DATA[15:0], data bus
- $\overline{\text{ABE[1:0]}}$/SDQM[1:0], AMC byte enables/SDC data masks
- $\overline{\text{BR}}$, $\overline{\text{BG}}$, $\overline{\text{BGH}}$, external bus access control signals
- CLKOUT, system clock for SDC and AMC

No other signals are multiplexed between the two controllers.

System Clock

The CLKOUT pin is shared by both the SDC and AMC. Two different registers are used to control this:

- EBIU_SDGCTL register, SCTLE bit for SDC clock
- EBIU_AMGCTL register, AMCKEN bit for AMC clock

If enabling or disabling the system clock, software control for both registers is required.

Error Detection

The EBIU responds to any bus operation which addresses the range of 0x0000 0000 – 0xEEFF FFFF, even if that bus operation addresses reserved or disabled memory or functions. It responds by completing the bus operation (asserting the appropriate number of acknowledges as specified by the bus master) and by asserting the bus error signal for these error conditions:

- Any access to a disabled external memory bank
- Any access to reserved SDRAM memory space
- Any access to unpopulated SDRAM space

If the core requested the faulting bus operation, the bus error response from the EBIU is gated into the hardware error interrupt (IVHW) internal to the core (this interrupt can be masked off in the core). If a DMA master requested the faulting bus operation, then the bus error is captured in that controller and can optionally generate an interrupt to the core.

Bus Request and Grant

The processor can relinquish control of the data and address buses to an external device. The processor three-states its memory interface to allow an external controller to access either external asynchronous or synchronous memory parts.

Operation

When the external device requires access to the bus, it asserts the bus request ($\overline{BR}$) signal. The $\overline{BR}$ signal is arbitrated with EAB requests. If no internal request is pending, the external bus request will be granted. The processor initiates a bus grant by:

- Three-stating the data and address buses and the asynchronous memory control signals. The synchronous memory control signals can optionally be three-stated.
- Asserting the bus grant ($\overline{BG}$) 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 $\overline{BR}$, the processor deasserts $\overline{BG}$ and continues execution from the point at which it stopped.

The processor asserts the $\overline{BGH}$ pin when it is ready to start another external port access, but is held off because the bus was previously granted.

When the bus has been granted, the BGSTAT bit in the SDSTAT register is set. This bit can be used by the processor to check the bus status to avoid initiating a transaction that would be delayed by the external bus grant.

If the system is using SDRAM, be sure to place the SDRAM into self-refresh mode before busmastership is granted. If this is not done, the SDRAM's data is lost.

AMC Overview and Features

The following sections describe the features of the AMC.

Features

The EBIU AMC features include:

- I/O width 16-bit, I/O supply 2.5 or 3.3 V
- Maximum throughput of 133 M bytes/second
- Supports up to 4M byte of SRAM in four external banks
- AMC supports 8-bit data masking writes
- AMC has control of the EBIU while auto-refresh is performed to SDRAM
- AMC supports asynchronous access extension (ARDY pin)
- Supports instruction fetch
- Allows booting from bank 0 ($\overline{\text{AMS0}}$)

Asynchronous Memory Interface

The asynchronous memory interface allows a glueless interface to a variety of memory and peripheral types. These include SRAM, ROM, EPROM, flash memory, and FPGA/ASIC designs. Four asynchronous memory regions are supported. Each has a unique memory select associated with it, shown in Table 6-1.

Table 6-1. Asynchronous Memory Bank Address Range

Memory Bank Select	Address Start	Address End
$\overline{\text{AMS[3]}}$	0x2030 0000	0x203F FFFF
$\overline{\text{AMS[2]}}$	0x2020 0000	0x202F FFFF
$\overline{\text{AMS[1]}}$	0x2010 0000	0x201F FFFF
$\overline{\text{AMS[0]}}$	0x2000 0000	0x200F FFFF

Asynchronous Memory Address Decode

The address range allocated to each asynchronous memory bank is fixed at 1M byte; however, not all of an enabled memory bank need be populated.

Note accesses to unpopulated memory of partially populated AMC banks do not result in a bus error and will alias to valid AMC addresses.

The asynchronous memory signals are defined in Table 6-2. The timing of these pins is programmable to allow a flexible interface to devices of different speeds. For example interfaces, see Chapter 21, “System Design”.

AMC Pin Description

The following table describes the signals associated with each interface.

Table 6-2. Asynchronous Memory Interface Signals

Pad	Pin Type [1]	Description
DATA[15:0]	I/O	External data bus
CLKOUT	O	Switches at system clock frequency. Connect to the peripheral if required.
ADDR[19:1]	O	External address bus
$\overline{\text{AMS[3:0]}}$	O	Asynchronous memory selects

Table 6-2. Asynchronous Memory Interface Signals (Cont'd)

Pad	Pin Type [1]	Description
$\overline{AWE}$	O	Asynchronous memory write enable
$\overline{ARE}$	O	Asynchronous memory read enable
$\overline{AOE}$	O	Asynchronous memory output enable In most cases, the $\overline{AOE}$ pin should be connected to the $\overline{OE}$ pin of an external memory-mapped asynchronous device. Please refer to the product data sheet for specific timing information between the $\overline{AOE}$ and $\overline{ARE}$ signals to determine which interface signal should be used in your system.
ARDY	I	Asynchronous memory ready response
$\overline{ABE[1:0]}$/SDQM[1:0]	O	Byte enables

1 Pin Types: I = Input, O = Output

AMC Description of Operation

The following sections describe the operation of the AMC.

Avoiding Bus Contention

Because the three-stated data bus is shared by multiple devices in a system, be careful to avoid contention. Contention causes excessive power dissipation and can lead to device failure. Contention occurs during the time one device is getting off the bus and another is getting on. If the first device is slow to three-state and the second device is quick to drive, the devices contend.

There are two cases where contention can occur. The first case is a read followed by a write to the same memory space. In this case, the data bus drivers can potentially contend with those of the memory device addressed by the read. The second case is back-to-back reads from two different

memory spaces. In this case, the two memory devices addressed by the two reads could potentially contend at the transition between the two read operations.

To avoid contention, program the turnaround time (bank transition time) appropriately in the asynchronous memory bank control registers. This feature allows software to set the number of clock cycles between these types of accesses on a bank-by-bank basis. Minimally, the EBIU provides one cycle for the transition to occur.

External Access Extension

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 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 External Access Extension” on page 6-16.

AMC Functional Description

The following sections provide a functional description of the AMC.

Programmable Timing Characteristics

This section describes the programmable timing characteristics for the EBIU. Timing relationships depend on the programming of the AMC, whether initiation is from the core or from memory DMA, and the sequence of transactions (read followed by read, read followed by write, and so on).

Asynchronous Reads

Figure 6-3 shows an asynchronous read bus cycle with timing programmed as setup = 2 cycles, read access = 2 cycles, hold = 1 cycle, and transition time = 1 cycle.

Asynchronous read bus cycles proceed as follows.

1. At the start of the setup period, $\overline{\text{AMS[x]}}$ and $\overline{\text{AOE}}$ assert. The address bus becomes valid. The $\overline{\text{ABE[1:0]}}$ signals are low during the read.
2. At the beginning of the read access period and after the 2 setup cycles, $\overline{\text{ARE}}$ asserts.
3. At the beginning of the hold period, read data is sampled on the rising edge of the EBIU clock. The $\overline{\text{ARE}}$ pin deasserts after this rising edge.
4. At the end of the hold period, $\overline{\text{AOE}}$ deasserts unless this bus cycle is followed by another asynchronous read to the same memory space. Also, $\overline{\text{AMS[x]}}$ deasserts unless the next cycle is to the same memory bank.
5. Unless another read of the same memory bank is queued internally, the AMC appends the programmed number of memory transition time cycles.

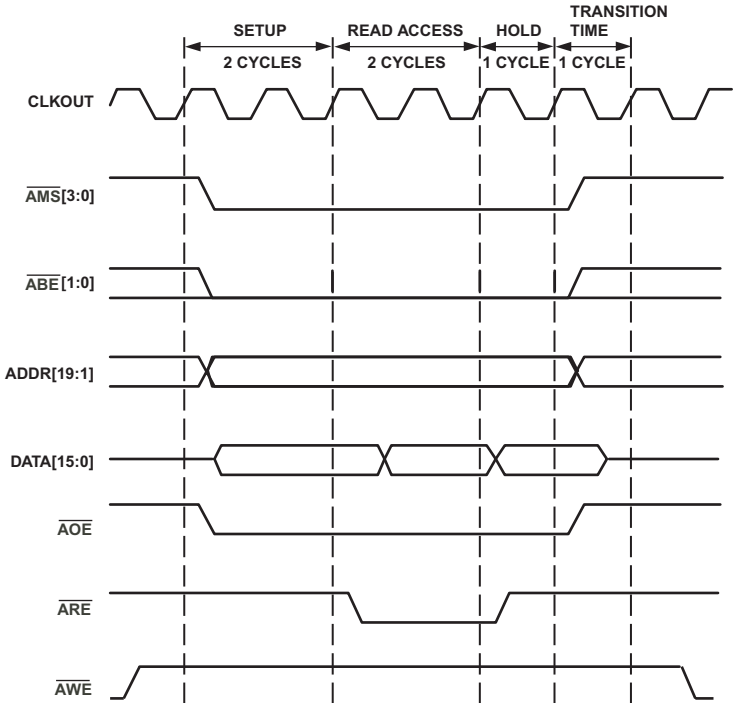


Figure 6-3. Asynchronous Read Bus Cycles

Asynchronous Writes

Figure 6-4 shows an asynchronous write bus cycle followed by an asynchronous read cycle to the same bank, with timing programmed as setup = 2 cycles, write access = 2 cycles, read access = 3 cycles, hold = 1 cycle, and transition time = 1 cycle.

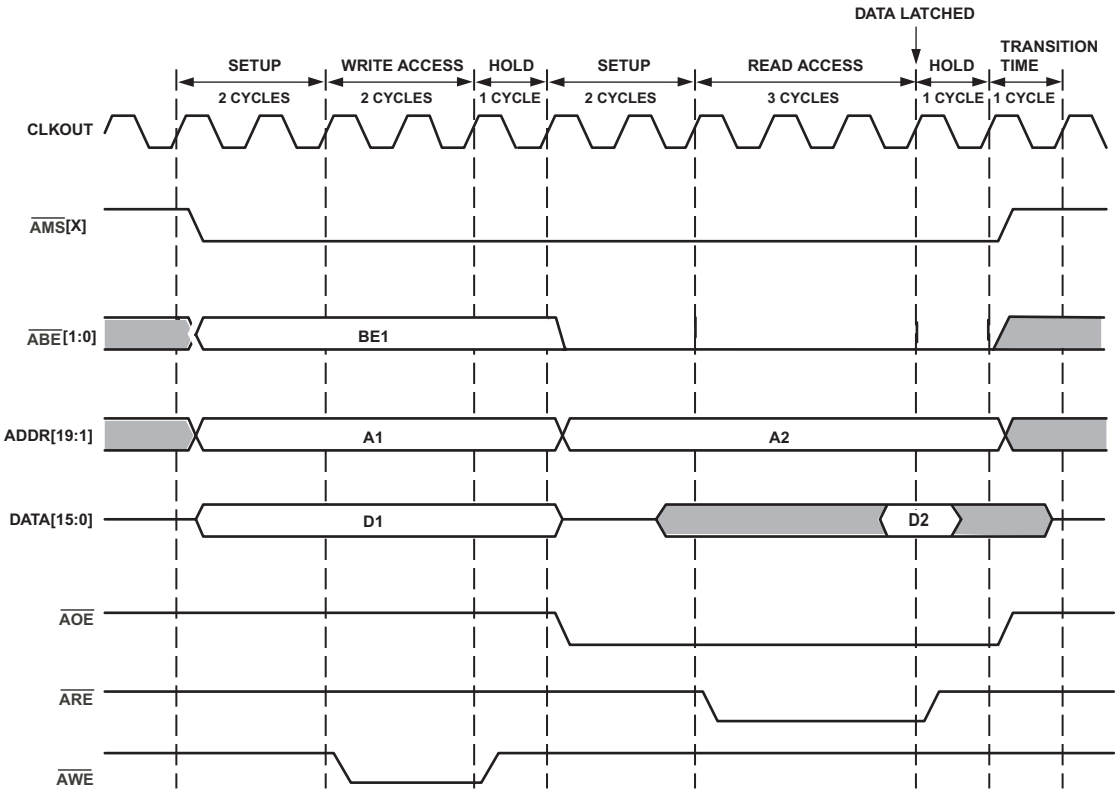


Figure 6-4. Asynchronous Write and Read Bus Cycles

Asynchronous write bus cycles proceed as follows.

1. At the start of the setup period, AMS[x], the address bus, data buses, and ABE[1:0] become valid. See "Byte Enables" on page 6-18 for more information.
2. At the beginning of the write access period, AWE asserts.
3. At the beginning of the hold period, AWE deasserts.

Asynchronous read bus cycles proceed as follows.

1. At the start of the setup period, $\overline{\text{AMS[x]}}$ and $\overline{\text{AOE}}$ assert. The address bus becomes valid. The $\overline{\text{ABE[1:0]}}$ signals are low during the read.
2. At the beginning of the read access period, $\overline{\text{ARE}}$ asserts.
3. At the beginning of the hold period, read data is sampled on the rising edge of the EBIU clock. The $\overline{\text{ARE}}$ signal deasserts after this rising edge.
4. At the end of the hold period, $\overline{\text{AOE}}$ deasserts unless this bus cycle is followed by another asynchronous read to the same memory space. Also, $\overline{\text{AMS[x]}}$ deasserts unless the next cycle is to the same memory bank.
5. Unless another read of the same memory bank is queued internally, the AMC appends the programmed number of memory transition time cycles.

Adding External Access Extension

The ARDY pin is used to insert extra wait states. The EBIU starts sampling ARDY on the clock cycle before the end of the programmed strobe period. If ARDY is sampled as deasserted, the access period is extended. The ARDY pin is then sampled on each subsequent clock edge. Read data is latched on the clock edge after ARDY is sampled as asserted. The read- or write-enable remains asserted for one clock cycle after ARDY is sampled as asserted. An example of this behavior is shown in Figure 6-5, 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.

The read access period must be programmed to a minimum of two cycles to make use of the ARDY input.

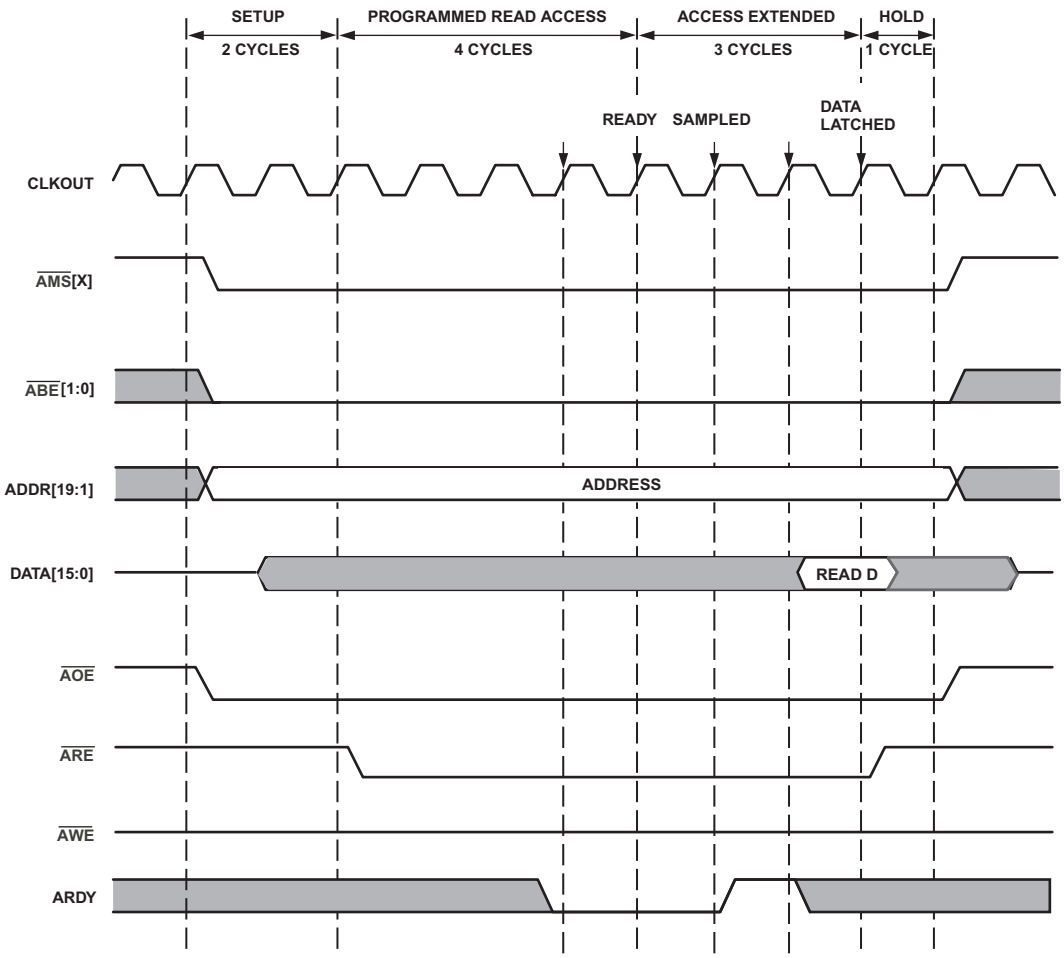


Figure 6-5. Inserting Wait States Using ARDY

Byte Enables

The AMC provides byte enable pins to allow the processor to perform efficient byte-wide arithmetic and byte-wide processing in external memory.

In general, there are two different ways to modify a single byte within the 16-bit interface. First, it can be done by a read/modify/write sequence. However, this is not very efficient because multiple accesses are required (that is, it takes many cycles for reads and writes to external memory). Another option is available where just a specific byte can be modified for a 16-bit devices using the $\overline{\texttt{ABE[1:0]}}$ pins. See Table 6-3.

The $\overline{\texttt{ABE[1:0]}}$ pins are both low during all asynchronous reads and 16-bit asynchronous writes. When an asynchronous write is made to the upper byte of a 16-bit memory, $\overline{\texttt{ABE1}}$ = 0 and $\overline{\texttt{ABE0}}$ = 1. When an asynchronous write is made to the lower byte of a 16-bit memory, $\overline{\texttt{ABE1}}$ = 1 and $\overline{\texttt{ABE0}}$ = 0.

Table 6-3. Byte Enables 8-Bit Write Accesses

Internal Address IA[0]	Internal Transfer Size	
	byte	2 bytes
0	$\overline{\text{ABE[1]}}$ = 1 $\overline{\text{ABE[0]}}$ = 0	$\overline{\text{ABE[1]}}$ = 0 $\overline{\text{ABE[1]}}$ = 0
1	$\overline{\text{ABE[0]}}$ = 0 $\overline{\text{ABE[0]}}$ = 1	$\overline{\text{ABE[1]}}$ = 0 $\overline{\text{ABE[1]}}$ = 0

AMC Programming Model

The asynchronous memory global control register (EBIU_AMGCTL) configures global aspects of the controller. It contains bank enables and other information as described in this section. This register should not be

programmed while the AMC is in use. The EBIU_AMGCTL register should be the last control register written to when configuring the processor to access external memory-mapped asynchronous devices.

Additional information for the EBIU_AMGCTL register bits includes:

- **Asynchronous memory clock enable** (AMCKEN)

  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 enable** (AMBEN).

  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. 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.

- **Core/DMA priority** (CDPRIO).

  This bit configures the AMC to control the priority over requests that occur simultaneously to the EBIU from either processor core or the DMA controller. When this bit is set to 0, a request from the core has priority over a request from the DMA controller to the AMC, unless the DMA is urgent. When the CDPRIO bit is set, all requests from the DMA controller, including the memory DMAs, have priority over core accesses. For the purposes of this discussion, core accesses include both data fetches and instruction fetches.

The CDPRIO bit also applies to the SDC.

The EBIU asynchronous memory controller has two asynchronous 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. These registers should not be programmed while the AMC is in use.

The timing characteristics of the AMC can be programmed using these four parameters:

- Setup: the time between the beginning of a memory cycle ($\overline{\text{AMS[x]}}$ low) and the read-enable assertion ($\overline{\text{ARE}}$ low) or write-enable assertion ($\overline{\text{AWE}}$ low).
- Read access: the time between read-enable assertion ($\overline{\text{ARE}}$ low) and deassertion ($\overline{\text{ARE}}$ high).
- Write access: the time between write-enable assertion ($\overline{\text{AWE}}$ low) and deassertion ($\overline{\text{AWE}}$ high).
- Hold: the time between read-enable deassertion ($\overline{\text{ARE}}$ high) or write-enable deassertion ($\overline{\text{AWE}}$ high) and the end of the memory cycle ($\overline{\text{AMS[x]}}$ high).

Each of these parameters can be programmed in terms of EBIU clock cycles. In addition, there are minimum values for these parameters:

- Setup ≥ 1 cycle
- Read access ≥ 1 cycle
- Write access ≥ 1 cycle
- Hold ≥ 0 cycles

AMC Registers

The following sections describe the AMC registers.

EBIU_AMGCTL Register

Figure 6-6 shows the asynchronous memory global control register (EBIU_AMGCTL).

Asynchronous Memory Global Control Register (EBIU_AMGCTL)

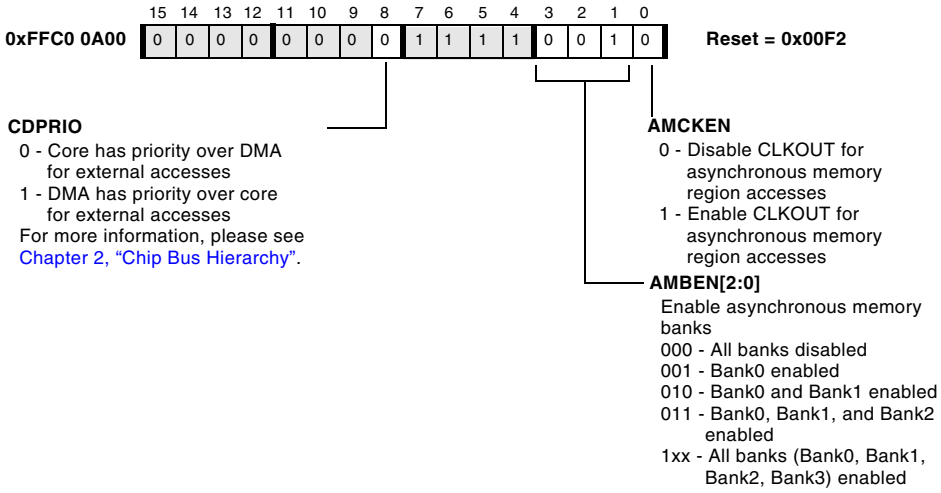


Figure 6-6. Asynchronous Memory Global Control Register

EBIU_AMBCTL0 and EBIU_AMBCTL1 Registers

Figure 6-7 and Figure 6-8 show the asynchronous memory bank control registers (EBIU_AMBCTL0 and EBIU_AMBCTL1).

Asynchronous Memory Bank Control 0 Register (EBIU_AMBCTL0)

0xFFC0 0A04 **Reset = 0xFFC2 FFC2**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
1	1	1	1	1	1	1	1	1	1	0	0	0	0	1	0

B1WAT[3:0]
Bank 1 write access time (number of cycles $\overline{AWE}$ is held asserted)
0000 - Not supported
0001 to 1111 - 1 to 15 cycles

B1RAT[3:0]
Bank 1 read access time (number of cycles $\overline{ARE}$ is held asserted)
0000 - Not supported
0001 to 1111 - 1 to 15 cycles

B1HT[1:0]
Bank 1 hold time (number of cycles between $\overline{AWE}$ or $\overline{ARE}$ deasserted, and $\overline{AOE}$ deasserted)
00 - 0 cycles
01 - 1 cycle
10 - 2 cycles
11 - 3 cycles

B1ST[1:0]
Bank 1 setup time (number of cycles after $\overline{AOE}$ asserted, before $\overline{AWE}$ or $\overline{ARE}$ asserted)
00 - 4 cycles
01 - 1 cycle
10 - 2 cycles
11 - 3 cycles

B1RDYEN
Bank 1 ARDY enable
0 - Ignore ARDY for accesses to this memory bank
1 - After access time countdown, use state of ARDY to determine completion of access

B1RDYPOL
Bank 1 ARDY polarity
0 - Transaction completes if ARDY sampled low
1 - Transition completes if ARDY sampled high

B1TT[1:0]
Bank 1 memory transition time (number of cycles inserted after a read access to this bank, and before a write access to this bank or a read access to another bank)
00 - 4 cycles for bank transition
01 - 1 cycle for bank transition
10 - 2 cycles for bank transition
11 - 3 cycles for bank transition

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

B0WAT[3:0]
Bank 0 write access time (number of cycles $\overline{AWE}$ is held asserted)
0000 - Not supported
0001 to 1111 - 1 to 15 cycles

B0RAT[3:0]
Bank 0 read access time (number of cycles $\overline{ARE}$ is held asserted)
0000 - Not supported
0001 to 1111 - 1 to 15 cycles

B0HT[1:0]
Bank 0 hold time (number of cycles between $\overline{AWE}$ or $\overline{ARE}$ deasserted, and $\overline{AOE}$ deasserted)
00 - 0 cycles
01 - 1 cycle
10 - 2 cycles
11 - 3 cycles

B0ST[1:0]
Bank 0 setup time (number of cycles after $\overline{AOE}$ asserted, before $\overline{AWE}$ or $\overline{ARE}$ asserted)
00 - 4 cycles
01 - 1 cycle
10 - 2 cycles
11 - 3 cycles

B0RDYEN
Bank 0 ARDY enable
0 - Ignore ARDY for accesses to this memory bank
1 - After access time countdown, use state of ARDY to determine completion of access

B0RDYPOL
Bank 0 ARDY polarity
0 - Transaction completes if ARDY sampled low
1 - Transition completes if ARDY sampled high

B0TT[1:0]
Bank 0 memory transition time (number of cycles inserted after a read access to this bank, and before a write access to this bank or a read access to another bank)
00 - 4 cycles for bank transition
01 - 1 cycle for bank transition
10 - 2 cycles for bank transition
11 - 3 cycles for bank transition

Figure 6-7. Asynchronous Memory Bank Control 0 Register

Asynchronous Memory Bank Control 1 Register (EBIU_AMBCTL1)

0xFFC0 0A08

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
1	1	1	1	1	1	1	1	1	1	0	0	0	0	1	0

Reset = 0xFFC2 FFC2

B3WAT[3:0]
Bank 3 write access time (number of cycles $\overline{AWE}$ is held asserted)
0000 - Not supported
0001 to 1111 - 1 to 15 cycles

B3RAT[3:0]
Bank 3 read access time (number of cycles $\overline{ARE}$ is held asserted)
0000 - Not supported
0001 to 1111 - 1 to 15 cycles

B3HT[1:0]
Bank 3 hold time (number of cycles between $\overline{AWE}$ or $\overline{ARE}$ deasserted, and $\overline{AOE}$ deasserted)
00 - 0 cycles
01 - 1 cycle
10 - 2 cycles
11 - 3 cycles

B3ST[1:0]
Bank 3 setup time (number of cycles after $\overline{AOE}$ asserted, before $\overline{AWE}$ or $\overline{ARE}$ asserted)
00 - 4 cycles
01 - 1 cycle
10 - 2 cycles
11 - 3 cycles

B3RDYEN
Bank 3 ARDY enable
0 - Ignore ARDY for accesses to this memory bank
1 - After access time countdown, use state of ARDY to determine completion of access

B3RDYPOL
Bank 3 ARDY polarity
0 - Transaction completes if ARDY sampled low
1 - Transition completes if ARDY sampled high

B3TT[1:0]
Bank 3 memory transition time (number of cycles inserted after a read access to this bank, and before a write access to this bank or a read access to another bank)
00 - 4 cycles for bank transition
01 - 1 cycle for bank transition
10 - 2 cycles for bank transition
11 - 3 cycles for bank transition

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

B2WAT[3:0]
Bank 2 write access time (number of cycles $\overline{AWE}$ is held asserted)
0000 - Not supported
0001 to 1111 - 1 to 15 cycles

B2RAT[3:0]
Bank 2 read access time (number of cycles $\overline{ARE}$ is held asserted)
0000 - Not supported
0001 to 1111 - 1 to 15 cycles

B2HT[1:0]
Bank 2 hold time (number of cycles between $\overline{AWE}$ or $\overline{ARE}$ deasserted, and $\overline{AOE}$ deasserted)
00 - 0 cycles
01 - 1 cycle
10 - 2 cycles
11 - 3 cycles

B2ST[1:0]
Bank 2 setup time (number of cycles after $\overline{AOE}$ asserted, before $\overline{AWE}$ or $\overline{ARE}$ asserted)
00 - 4 cycles
01 - 1 cycle
10 - 2 cycles
11 - 3 cycles

B2RDYEN
Bank 2 ARDY enable
0 - Ignore ARDY for accesses to this memory bank
1 - After access time countdown, use state of ARDY to determine completion of access

B2RDYPOL
Bank 2 ARDY polarity
0 - Transaction completes if ARDY sampled low
1 - Transition completes if ARDY sampled high

B2TT[1:0]
Bank 2 memory transition time (number of cycles inserted after a read access to this bank, and before a write access to this bank or a read access to another bank)
00 - 4 cycles for bank transition
01 - 1 cycle for bank transition
10 - 2 cycles for bank transition
11 - 3 cycles for bank transition

Figure 6-8. Asynchronous Memory Bank Control 1 Register

AMC Programming Examples

Listing 6-1 and Listing 6-2 provide examples for working with the AMC.

Listing 6-1. 16-Bit Core Transfers to SRAM

```
.section L1_data_b;
.byte2 source[N] = 0x1122, 0x3344, 0x5566, 0x7788;
.section SRAM_bank_0;
.byte2 dest[N];
.section L1_code;
I0.L = lo(source);
I0.H = hi(source);
I1.L = lo(dest);
I1.H = hi(dest);
     R0.L = w[I0++];
     P5=N-1;
     lsetup(lp, lp) LC0=P5;
lp:  R0.L = w[I0++] || w[I1++] = R0.L;
                       w[I1++] = R0.L;
```

Listing 6-2. 8-Bit Core Transfers to SRAM Using Byte Mask ABE[1:0] Pins

```
.section L1_data_b;
.byte source[N] = 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88;
.section SRAM_bank_0;
.byte dest[N];
p0.L = lo(source);
p0.H = hi(source);
p1.L = lo(dest);
p1.H = hi(dest);
        p5=N;
        lsetup(start, end) LC0=P5;
```

```
start:  R0 = b[p0++](z);
end:    b[p1++] = R0;   /* byte data masking */
```

SDC Overview and Features

The SDRAM Controller (SDC) enables the processor to transfer data to and from Synchronous DRAM (SDRAM) with a maximum frequency specified in the product data sheet. The processor supports a glueless interface with one external bank of standard SDRAMs of 64 Mbit to 512 Mbit, with configurations x4, x8, and x16, up to a maximum total capacity of 512M bytes of SDRAM.

Features

The EBIU SDC provides a glueless interface with standard SDRAMs. Features include:

- I/O width 16-bit, I/O supply 2.5 or 3.3 V
- Maximum throughput of 266 M bytes/second
- Supports up to 512M byte of SDRAM in external bank
- Types of 64, 128, 256, and 512M bit with I/O of x4, x8, and x16
- Supports SDRAM page sizes of 512 byte, 1K , 2K , and 4K byte
- Supports multibank operation within the SDRAM
- Supports mobile SDRAMs
- SDC uses no-burst mode (`BL = 1`) with sequential burst type
- SDC supports 8-bit data masking writes
- SDC uses open page policy—any open page is closed only if a new access in another page of the same bank occurs

- Uses a programmable refresh counter to coordinate between varying clock frequencies and the SDRAM's required refresh rate
- Provides multiple timing options to support additional buffers between the processor and SDRAM
- Allows independent auto-refresh while the asynchronous memory controller has control of the EBIU port
- Supports self-refresh mode for power savings
- During hibernate state, self-refresh mode is supported
- Supports instruction fetch

SDRAM Configurations Supported

Table 6-4 shows all possible bank sizes, and SDRAM discrete component configurations that can be gluelessly interfaced to the SDC. The bank width for all cases is 16 bits.

Table 6-4. SDRAM Discrete Component Configurations Supported

System Size (M byte)	System Size (M bit)	SDRAM Configuration	Number of Chips
16	8M x 16	8M x 8	2
16	8M x 16	8M x 16	1
32	16M x 16	16M x 4	4
32	16M x 16	16M x 8	2
32	16M x 16	16M x 16	1
64	32M x 16	32M x 4	4
64	32M x 16	32M x 8	2
64	32M x 16	32M x 16	1

Table 6-4. SDRAM Discrete Component Configurations Supported (Cont'd)

System Size (M byte)	System Size (M bit)	SDRAM Configuration	Number of Chips
128	64M x 16	64M x 4	4
128	64M x 16	64M x 8	2
128	64M x 16	64M x 16	1
256	128M x 16	128M x 4	4
256	128M x 16	128M x 8	2
256	128M x 16	128M x 16	1
512	256M x 16	256M x 4	4
512	256M x 16	256M x 8	2
512	256M x 16	256M x 16	1

SDRAM External Bank Size

The total amount of external SDRAM memory addressed by the processor is controlled by the EBSZ bits of the EBIU_SDBCTL register (see Table 6-5). Accesses above the range shown for a specialized EBSZ value results in an internal bus error and the access does not occur. For more information, see "Error Detection" on page 6-7.

SDC Address Mapping

The address mapping scheme describes how the SDC maps the address into SDRAM. To access SDRAM, the SDC uses the bank interleaving map scheme, which fills each internal SDRAM bank before switching to the next internal bank. Since the SDRAMs have four internal banks, the entire SDRAM address space is therefore divided into four sub-address regions containing the addresses of each internal bank. (See Figure 6-10

on page 6-40.) It starts with address 0x0 for internal bank A and ends with the last valid address (specified with EBSZ and EBCAW parameters) containing the internal bank D.

The internal 29-bit non-multiplexed address (See Figure 6-9) is multiplexed into:

- Byte data mask (IA[0])
- SDRAM column address
- SDRAM row address
- Internal SDRAM bank address

A good understanding of the SDC's address map scheme in conjunction with the multibank operation is required to obtain optimized system performance.

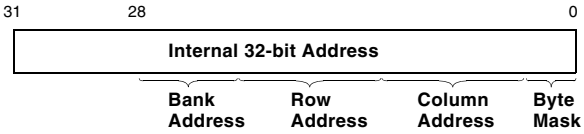


Figure 6-9. Multiplexed SDRAM Addressing Scheme

Table 6-5. External Bank Size Encodings

EBSZ	Bank Size (Mbyte)	Valid SDRAM Addresses
000	16	0x0000 0000 – 0x00FF FFFF
001	32	0x0000 0000 – 0x01FF FFFF
010	64	0x0000 0000 – 0x03FF FFFF
011	128	0x0000 0000 – 0x07FF FFFF
100	256	0x0000 0000 – 0x0FFF FFFF
101	512	0x0000 0000 – 0x1FFF FFFF

Internal SDRAM Bank Select

The internal SDRAM banks are driven by the ADSP-BF537's `ADDR[19:18]` which are part of the row and column address and connected to the SDRAM's `BA[1:0]`.

Do not flip up both internal bank select connections, if using the mobile SDRAM's PASR feature. If this is done, the system will not work properly because the selected internal banks are not refreshed during partial array self-refresh.

Parallel Connection of SDRAMs

To specify an SDRAM system, multiple possibilities are given based on the different architectures. (See Table 6-13 on page 6-64.) For the ADSP-BF537, I/0 capabilities of 1 x 16-bit, 2 x 8-bit or 4 x 4-bit are given. The reason to use a system of 4 x 4-bit vs. 2 x 8-bit or 1 x 16-bit is determined by the SDRAM's page size. All 3 systems have the same external bank size, but different page sizes. On one hand, the higher the page size, the higher the performance. On the other hand, the higher the page size, the higher the hardware layout requirements.

Even if connecting SDRAMs in parallel, the SDC always considers the entire system as one external SDRAM bank (`SMS` pin) because all address and control lines feed the parallel parts.

However, access to a single cluster part is achieved using the mask feature (`SDQM[1:0]` pins). This allows masked 8-bit I/O writes to dedicated chips whereby the other 8-bit I/O is masked at its input buffer of the other chips. See Listing 6-4 on page 6-80.

SDC Interface Overview

The following sections describe the SDC interface.

SDC Pin Description

The SDRAM interface signals are shown in Table 6-6.

Table 6-6. SDRAM Interface Signals

Pad	Pin Type [1]	Description
DATA[15:0]	I/O	External data bus
ADDR[19:18], ADDR[16:12], ADDR[10:1],	O	External address bus Connect to SDRAM address pins. Bank address is output on ADDR[19:18] and should be connected to SDRAM BA[1:0] pins.
$\overline{\text{SRAS}}$	O	SDRAM row address strobe pin Connect to SDRAM's $\overline{\text{RAS}}$ pin.
$\overline{\text{SCAS}}$	O	SDRAM column address strobe pin Connect to SDRAM's $\overline{\text{CAS}}$ pin.
$\overline{\text{SWE}}$	O	SDRAM write enable pin Connect to SDRAM's $\overline{\text{WE}}$ pin.
$\overline{\text{ABE[1:0]}}$/ SDQM[1:0]	O	SDRAM data mask pins Connect to SDRAM's DQM pins.
$\overline{\text{SMS}}$	O	Memory select pin of external memory bank configured for SDRAM Connect to SDRAM's $\overline{\text{CS}}$ (Chip Select) pin. Active low.
SA10	O	SDRAM A10 pin SDRAM interface uses this pin to be able to do refreshes while the AMC is using the bus. Connect to SDRAM's A[10] pin.

Table 6-6. SDRAM Interface Signals (Cont'd)

Pad	Pin Type [1]	Description
SCKE	O	SDRAM clock enable pin Connect to SDRAM's CKE pin.
CLKOUT	O	SDRAM clock output pin Switches at system clock frequency. Connect to the SDRAM's CLK pin.

1 Pin Types: I = Input, O = Output

SDRAM Performance

On-page sequential or non-sequential accesses are from internal data memory to SDRAM. Table 6-7 summarizes SDRAM performance for these on-page accesses.

Table 6-7. SDRAM Performance Between Internal Data Memory and SDRAM

Type of access	Performance
DAG access, write	1 SCLK cycle per 16-bit word
DAG access, read	8 SCLK cycles per 16-bit word
MemDMA access, write	1 SCLK cycle per 16-bit word
MemDMA access, read	≈ 1.1 SCLK cycles per 16-bit word

On-page sequential instruction fetches from SDRAM are summarized in Table 6-8.

Table 6-8. SDRAM Performance For On-Page Instruction Fetches

Type of access	Performance
Ifetch from SDRAM	≈ 1.1 SCLK cycles per 16-bit word
I/Dcache line fill from SDRAM	≈ 1.1 SCLK cycles per 16-bit word

Off-page accesses are summarized in Table 6-9.

Table 6-9. SDRAM Stall Cycles For Off-Page Accesses

Type of access	Stall Cycles
Write	$t_{WR} + t_{RP} + t_{RCD}$
Read	$t_{RP} + t_{RCD} + CL$

SDC Description of Operation

The following sections describe the operation of the SDC.

Definition of SDRAM Architecture Terms

The following are definitions of SDRAM architecture terms used in the remainder of this chapter.

Refresh

Because the information is stored in a small capacitance suffering on leakage effects, the SDRAM cell needs to be refreshed periodically with the refresh command.

Row Activation

SDRAM accesses are multiplexed, which means any first access will open a row/page before the column access is performed. It stores the row in a "row cache" called row activation.

Column Read/Write

The row's columns represent a page, which can be accessed with successive read or write commands without needing to activate another row. This is called column access and performs transfers from the "row cache."

Row Precharge

If the next access is in a different row, the current row is closed before another is opened. The current "row cache" is written back to the row. This is called row precharge.

Internal 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.

External Bank

This is the address region where the SDC address the SDRAM.

Do not confuse the internal banks, which are internal to the SDRAM and are selected with the `BA[1:0]` pins with the external bank that is enabled by the `CS` pin.

Memory Size

Since the 2D memory is based on rows and columns, the size is:

```
mem size =
         (# rows) x (# columns) x (# internal banks) x I/O (Mbit)
```

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 = 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 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.

CAS Latency

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.

Data I/O Mask Function

SDRAMs allow a data byte-masking capability on writes. The mask pins DQM[1:0] are used to block the data input buffer of the SDRAM during write operations.

SDRAM Commands

SDRAM commands are not based on typical read or write strobes. The pulsed $\overline{CS}$, $\overline{RAS}$, $\overline{CAS}$, and $\overline{WE}$ lines determine the command on the rising clock edge by a truth table.

Mode Register Set (MRS) command

SDRAM devices contain an internal extended configuration register which allows specification of the mobile SDRAM device's functionality.

Extended Mode Register Set (EMRS) command

Mobile SDRAM devices contain an internal extended configuration register which allows specification of the mobile SDRAM device's functionality.

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.

Read/Write command

For the read command, the SDRAM latches the column address. The start address is set according to the column address. For the write command, SDRAM latches the column address. Data is also asserted in the same cycle. The start address is set according to the column address.

Precharge/Precharge All 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.

Auto-refresh command

When the SDC refresh counter times out, the SDC precharges all four banks of SDRAM and then issues an auto-refresh command to them. This causes the SDRAM to generate an internal auto-refresh cycle. When the internal refresh completes, all four internal SDRAM banks are precharged.

Enter Self-Refresh Mode

When the SDRAM enters self-refresh mode, the SDRAM's internal timer initiates refresh cycles periodically, without external control input.

Exit Self-Refresh Mode

When the SDRAM exits self-refresh mode, the SDRAM's internal timer stops refresh cycles and relinquishes control to external SDC.

SDC Timing Specs

The following SDRAM timing specs are discussed because they are used by the SDC and SDRAM. To program the SDRAM interface, you need the SDRAM's specific datasheet information

Any absolute timing parameter must be normalized to the system clock which allows the SDC to adapt to the timing parameter of the device.

t_{MRD}

This is the required delay between issuing a mode register set and an activate command during powerup.

Dependency: system clock frequency
SDC setting: 3 system clock cycles
SDC usage: MRS command

t_{RAS}

This is the required delay between issuing a bank A activate command and issuing a bank A precharge command.

Dependency: system clock frequency
SDC setting: 1–15 normalized system clock cycles
SDC usage: single column read/write, auto-refresh, self-refresh command

CL

The CAS latency or read latency is the delay between when the SDRAM detects the read command and when it provides the data off-chip. This spec does not apply to writes.

Dependency: system clock frequency and speed grade
SDC setting: 2–3 normalized system clock cycles
SDC usage: first read command

t_{RCD}

This is the required delay between a bank A activate command and the first bank A read or write command.

Dependency: system clock frequency
SDC setting: 1–7 normalized system clock cycles
SDC usage: first read/write command

t_{RRD}

This is the required delay between a bank A activate command and a bank B activate command. This spec is used for multibank operation.

Dependency: system clock frequency
SDC setting: t_{RCD} + 1 normalized system clock cycles
SDC usage: multiple bank activation

t_{WR}

This is the required delay between a bank A write command and a bank A precharge command. This spec does not apply to reads.

Dependency: system clock frequency
SDC setting: 1–3 normalized system clock cycles
SDC usage: during off-page write command

t_{RP}

This is the required delay between a bank A precharge command and a bank A activation command.

Dependency: system clock frequency
SDC setting: 1–7 normalized system clock cycles
SDC usage: off-page read/write, auto-refresh, self-refresh command

t_{RC}

This is the required delay between issuing successive bank activate commands.

Dependency: system clock frequency
SDC setting: t_{RAS} + t_{RP} normalized system clock cycles
SDC usage: single column read/write command

t_{RFC}

This is the required delay between issuing successive auto-refresh commands (all banks).

Dependency: system clock frequency
SDC setting: t_{RAS} + t_{RP} normalized system clock cycles
SDC usage: auto-refresh, exit self-refresh command

t_{XSR}

This is the required delay between exiting self-refresh mode and the auto-refresh command.

Dependency: system clock frequency
SDC setting: t_{RAS} + t_{RP} normalized system clock cycles
SDC usage: exit self-refresh command

t_{REF}

This is the row refresh period, and typically takes 64 ms.

Dependency: system clock frequency
SDC setting: none
SDC usage: auto-refresh command

t_{REFI}

This is the row refresh interval and typically takes 15.6 ms for < 8k rows and 7.8 ms for >= 8k rows. This spec is available by dividing the number of rows by t_{REF}. This number is used by the SDC refresh counter.

Dependency: system clock frequency
SDC setting: t_{REFI} normalized system clock cycles (RDIV register)
SDC usage: auto-refresh command

SDC Functional Description

The functional description of the SDC is provided in the following sections.

SDC Operation

The AMC normally generates an external memory address, which then asserts the corresponding CS select, along with RD and WR strobes. However these control signals are not used by the SDC. The internal strobes are used to generate pulsed commands ($\overline{\text{SMS}}$, SCKE, $\overline{\text{SRAS}}$, $\overline{\text{SCAS}}$, $\overline{\text{SWE}}$) within a truth table (see Table 6-11 on page 6-48). The memory access to SDRAM is based by mapping ADDR[28:0] causing an internal memory select to SDRAM space (see Figure 6-10).

The configuration is programmed in the SDBCTL register. The SDRAM controller can hold off the processor core or DMA controller with an internally connected acknowledge signal, as controlled by refresh, or page miss latency overhead.

A programmable refresh counter is provided which generates background auto-refresh cycles at the required refresh rate based on the clock frequency used. The refresh counter period is specified with the RDIV field in the SDRAM refresh rate control register.

To allow auto-refresh commands to execute in parallel with any AMC access, a separate A10 pin (SA10) is provided.

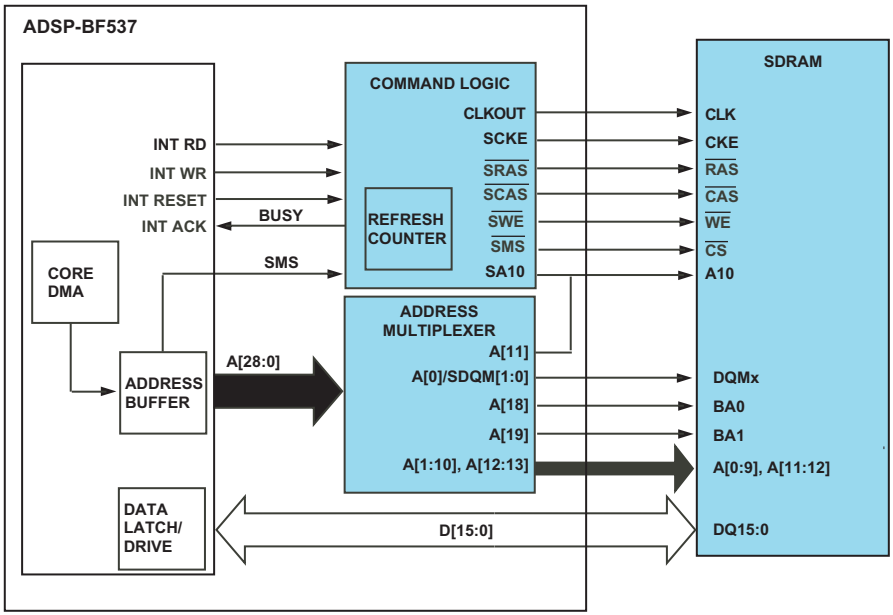


Figure 6-10. Simplified SDC Architecture

The internal 32-bit non-multiplexed address is multiplexed into:

- Data mask for bytes
- SDRAM column address
- SDRAM row address
- Internal SDRAM bank address

Bit `A[0]` is used for 8-bit wide SDRAMs to generate the data masks. The next lowest bits are mapped into the column address, next bits are mapped into the row address, and the final two bits are mapped into the internal bank address. This mapping is based on the `EBCAW` and `EBSZ` values programmed into the SDRAM memory bank control register.

The SDC uses no burst mode (`BL = 1`) for read and write operations. This requires the SDC to post every read or write address on the bus as for non-sequential reads or writes, but does not cause any performance degradation. For read commands, there is a latency from the start of the read command to the availability of data from the SDRAM, equal to the CAS latency. This latency is always present for any single read transfer. Subsequent reads do not have latency.

Whenever a page miss to the same bank occurs, the SDC executes a precharge command followed by a bank activate command before executing the read or write command. If there is a page hit, the read or write command can be given immediately without requiring the precharge command.

SDC Address Muxing

Table 6-10 shows the connection of the address pins with the SDRAM device pins.

Table 6-10. SDRAM Address Connections for 16-bit Banks

External Address Pin	SDRAM Address Pin
ADDR[19]	BA[1]
ADDR[18]	BA[0]
ADDR[16]	A[15]
ADDR[15]	A[14]
ADDR[14]	A[13]
ADDR[13]	A[12]
ADDR[12]	A[11]
ADDR[11]	Not used
SA[10]	A[10]
ADDR[10]	A[9]
ADDR[9]	A[8]
ADDR[8]	A[7]
ADDR[7]	A[6]
ADDR[6]	A[5]
ADDR[5]	A[4]
ADDR[4]	A[3]
ADDR[3]	A[2]
ADDR[2]	A[1]
ADDR[1]	A[0]

Multibank Operation

Since an SDRAM contains 4 independent internal banks (A-D), the SDC is capable of supporting multibank operation thus taking advantage of the architecture.

Any first access to SDRAM bank (A) will force an activate command before a read or write command. However, if any new access falls into the address space of the other banks (B, C, D) the SDC leaves bank (A) open and activates any of the other banks (B, C, D). Bank (A) to bank (B) active time is controlled by $t_{RRD} = t_{RCD} + 1$. This scenario is repeated until all 4 banks (A-D) are opened and results in an effective page size up to 4 pages because no latency causes switching between these open pages (compared to 1 page in only one bank at the time). Any access to any closed page in any opened bank (A-D) forces a precharge command only to that bank. If, for example, 2 MemDMA channels are pointing to the same internal SDRAM bank, this always forces precharge and activation cycles to switch between the different pages. However, if the 2 MemDMA channels are pointing to different internal SDRAM banks, it does not cause additional overhead. See Figure 6-11.

The benefit of multibank operation reduces precharge and activation cycles by mapping opcode/data among different internal SDRAM banks driven by the `A[19:18]` pins.

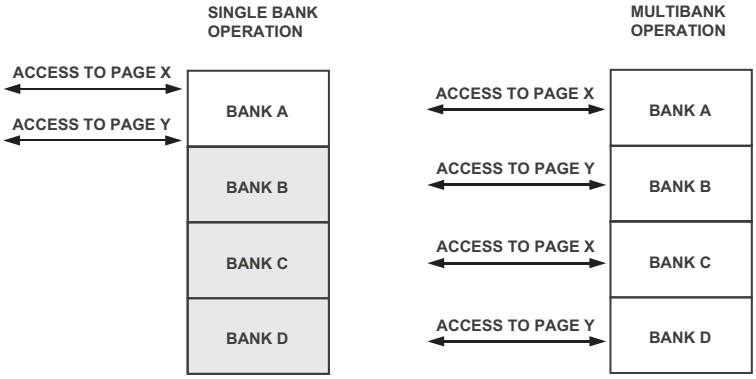


Figure 6-11. SDRAM Bank Operation Types

Core and DMA Arbitration

The CDPRIO bit configures the SDC to control the priority over requests that occur simultaneously to the EBIU from either the processor core or the DMA controller. When this bit is set to 0, a request from the core has priority over a request from the DMA controller to the SDC, unless the DMA is urgent. When it is set to 1, all requests from the DMA controller, including the memory DMAs, have priority over core accesses. For the purposes of this discussion, core accesses include both data fetches and instruction fetches.

Changing System Clock During Runtime

All timing specs are normalized to the system clock. Since most of them are minimum specs, except t_{REF}, which is a maximum spec, a variation of system clock will on one hand violate a specific spec and on the other hand cause a performance degradation for the other specs.

The reduction of system clock will violate the minimum specs, while increasing system clock will violate the maximum t_{REF} spec. Therefore, careful software control is required to adapt these changes.

For most applications, the SDRAM powerup sequence and writing of the mode register needs to be done only once. Once the powerup sequence has completed, the PSSE bit should not be set again unless a change to the mode register is desired.

The recommended procedure for changing the PLL VCO frequency is:

1. Issue an SSYNC instruction to ensure all pending memory operations have completed.
2. Set the SDRAM to self-refresh mode by writing a 1 to the SRFS bit of EBIU_SDGCTL.
3. Execute the desired PLL programming sequence. (For details, refer to Chapter 20, "Dynamic Power Management".)
4. After the wakeup occurs that signifies the PLL has settled to the new VCO frequency, reprogram the SDRAM registers (EBIU_SDRRC, EBIU_SDGCTL) with values appropriate to the new SCLK frequency, and assure that the PSSE bit is set.
5. Bring the SDRAM out of self-refresh mode by clearing the SRFS bit of EBIU_SDGCTL.

Changing the SCLK frequency using the SSEL bits in PLL_DIV, as opposed to actually changing the VCO frequency, should be done using these steps:

1. Issue an SSYNC instruction to ensure all pending memory operations have completed.
2. Set the SDRAM to self-refresh mode by writing a 1 to the SRFS bit of EBIU_SDGCTL.
3. Execute the desired write to the SSEL bits.

4. Reprogram the SDRAM registers with values appropriate to the new SCLK frequency, and assure that the PSSE bit is set.
5. Bring the SDRAM out of self-refresh mode by clearing the SRFS bit of EBIU_SDGCTL.

Changing Power Management During Runtime

Deep Sleep Mode

During deep sleep mode, the core and system clock will halt. Therefore, a careful software control is required to enter SDRAM in self-refresh before the device enters deep sleep mode.

Hibernate State

In hibernate state, only the I/O voltage is applied, the core voltage is 0 (core reset). In order to save the SDRAM's volatile data, the ADSP-BF537 supports low level of SCKE pin even during core reset. Setting the SCKELOW bit of VR_CTL keeps the SCKE signal low thus ensuring self-refresh mode. For details, refer to Chapter 20, "Dynamic Power Management".

Shared SDRAM

Busmastership can be requested using $\overline{BR}$ and $\overline{BG}$ pins. To grant busmastership to external SDC, use the CDDBG bit of EBIU_SDGCTL. This occurs asynchronously during self-refresh mode, because both auto-refresh time bases are fully independent and will be synchronized during the t_{XSR} timing spec by each busmaster, leaving the self-refresh mode to get access to shared SDRAM. The system requires additional logic for a common control of the SDRAM's CKE pin. The following steps illustrate the recommended procedure.

1. Boot stream of ADSP-BF537 processor must set the CDDBG bit of EBIU_SDGCTL to allow busmastership to a host.
2. SDRAM dummy access will perform the powerup sequence.

3. Host enters self-refresh mode.
4. Programmable flag PFx driven from host will trigger an ISR which sets the SFRS bit to cause the ADSP-BF537 to enter self-refresh mode.
5. Host deasserts $\overline{BR}$ pin which is granted with $\overline{BG}$ pin.
6. Host SDC asserts CKE pin to exit self-refresh mode indicating SDRAM access.
7. Host SDC deasserts CKE pin to finish SDRAM access and re-enters self-refresh mode.
8. Host deasserts $\overline{BR}$ pin, answered with deassertion of $\overline{BG}$ pin.
9. Programmable flag PFx driven from host will trigger an ISR which clears the SRFS bit of EBIU_SDGCTL and performs a dummy access to exit self-refresh mode.

SDC Commands

This section provides a description of each of the commands that the SDC uses to manage the SDRAM interface. These commands are initiated automatically upon a memory read or memory write. A summary of the various commands used by the on-chip controller for the SDRAM interface is as follows.

- Mode register set
- Extended mode register set
- Bank activation
- Read and write
- Single precharge
- Precharge all

- Auto-refresh
- Self-refresh
- NOP

Table 6-11 shows the SDRAM pin state during SDC commands.

Table 6-11. Pin State During SDC Commands

Command	SCKE (n - 1)	SCKE (n)	$\overline{SMS}$	$\overline{SRAS}$	$\overline{SCAS}$	$\overline{SWE}$	SA10	Addresses
(E)/Mode register set	High	High	Low	Low	Low	Low	Op-code	Op-code
Activate	High	High	Low	Low	High	Low	Valid address bit	Valid
Read	High	High	Low	High	Low	High	Low (CMD)	Valid
Single precharge	High	High	Low	Low	High	Low	Don't care	Valid
Precharge all	High	High	Low	Low	High	Low	Low	Don't care
Write	High	High	Low	High	Low	Low	Low (CMD)	Valid
Auto-refresh	High	High	Low	Low	Low	High	Don't care	Don't care
Self-refresh entry	High	Low	Low	Low	Low	High	Don't care	Don't care
Self-refresh	Low	Low	Don't care	Don't care	Don't care	Don't care	Don't care	Don't care
Self-refresh exit	Low	High	High	Don't care	Don't care	Don't care	Don't care	Don't care

Table 6-11. Pin State During SDC Commands (Cont'd)

Command	SCKE (n - 1)	SCKE (n)	$\overline{SMS}$	$\overline{SRAS}$	$\overline{SCAS}$	$\overline{SWE}$	SA10	Addresses
NOP	High	High	Low	High	High	High	Don't care	Don't care
Inhibit	High	High	High	Don't care	Don't Care	Don't care	Don't care	Don't care

Mode Register Set Command

The Mode Register Set (MRS) command initializes SDRAM operation parameters. This command is a part of the SDRAM powerup sequence. The MRS command uses the address bus of the SDRAM as data input. The powerup sequence is initiated by writing 1 to the `PSSE` bit in the SDRAM memory global control register (`EBIU_SDGCTL`) and then writing or reading from any enabled address within the SDRAM address space to trigger the powerup sequence. The exact order of the powerup sequence is determined by the `PSM` bit of the `EBIU_SDGCTL` register.

The MRS command initializes these parameters:

- Burst length = 1, bits `A[2-0]`, always 0
- Burst type = sequential, bit `A[3]`, always 0
- CAS latency, bits `A[6-4]`, programmable in the `EBIU_SDGCTL` register
- Bits `A[12-7]`, always 0

After powerup and before executing a read or write to the SDRAM memory space, the application must trigger the SDC to write the SDRAM's mode register. The write of the SDRAM's mode register is triggered by writing a 1 to the `PSSE` bit in the SDRAM memory global control register (`EBIU_SDGCTL`) and then issuing a read or write transfer to the SDRAM address space. The initial read or write triggers the SDRAM powerup

sequence to be run, which programs the SDRAM's mode register with burst length, burst type, and CAS latency from the EBIU_SDGCTL register and optionally the content to the extended mode register. This initial read or write to SDRAM takes many cycles to complete.

While executing an MRS command, the unused address pins are set to 0. During the two clock cycles following the MRS command (t_{MRD}), the SDC issues only NOP commands.

Extended Mode Register Set Command (Mobile SDRAM)

The extended mode register is a subset of the mode register. The EBIU enables programming of the extended mode register during powerup via the EMREN bit in the EBIU_SDGCTL register.

The extended mode register is initialized with these parameters:

- Partial array self-refresh, bits A[2-0], bit A[2] always 0, bits A[1-0] programmable in EBIU_SDGCTL
- Temperature compensated self-refresh, bits A[4-3], bit A[3] always 1, bit A[4] programmable in EBIU_SDGCTL
- Drive strength control, bits A[6-5], always 0
- Bits A[12-7], always 0, and bit A[13] always 1

Not programming the extended mode register upon initialization results in default settings for the low-power features. The extended mode defaults with the temperature sensor enabled, full drive strength, and full array refresh.

Bank Activation Command

The bank activation command is required for first access to any internal bank in SDRAM. Any subsequent access to the same internal bank but different row will be preceded by a precharge and activation command to that bank.

However, if an access to another bank occurs, the SDC leaves the current page open and issues a bank activate command before executing the read or write command to that bank. With this method, called multibank operation, one page per bank can be open at a time, which results in a maximum of 4 pages.

Read/Write Command

A read/write command is executed if the next read/write access is in the present active page. During the read command, the SDRAM latches the column address. The delay between activate and read commands is determined by the t_{RCD} parameter. Data is available from the SDRAM after the CAS latency has been met.

In the write command, the SDRAM latches the column address. The write data is also valid in the same cycle. The delay between activate and write commands is determined by the t_{RCD} parameter.

The SDC does not use the auto-precharge function of SDRAMs, which is enabled by asserting `SA10` high during a read or write command.

Write Command With Data Mask

During partial writes to SDRAM, the `SDQM[1:0]` pins are used to mask writes to bytes that are not accessed. Table 6-12 shows the `SDQM[1:0]` encodings based on the internal transfer address bit `IA[0]` and the transfer size.

During read transfers to SDRAM banks, reads are always done of all bytes in the bank regardless of the transfer size. This means for 16-bit SDRAM banks, `SDQM[1:0]` are all 0s.

Table 6-12. SDQM[1:0] Encodings During Writes

Internal Address IA[0]	Internal Transfer Size	
	byte	2 bytes
0	SDQM[1] = 1 SDQM[0] = 0	SDQM[1] = 0 SDQM[1] = 0
1	SDQM[0] = 0 SDQM[0] = 1	SDQM[1] = 0 SDQM[1] = 0

Single Precharge Command

For a page miss during reads or writes in a specific internal SDRAM bank, the SDC uses the single precharge command to that bank.

The SDC does not use the auto-precharge read or write command of SDRAMs, which is enabled by asserting `SA10` high during a read or write command.

Precharge All Command

The precharge all command is given to precharge all internal banks at the same time before executing an auto-refresh. All open banks will be automatically closed. This is possible since the SDC uses a separate `SA10` pin which is asserted high during this command. This command is preceding the auto-refresh command.

Auto-Refresh Command

The SDRAM internally increments the refresh address counter and causes an auto-refresh to occur internally for that address when the auto-refresh command is given. The SDC generates an auto-refresh command after the

SDC refresh counter times out. The RDIV value in the SDRAM refresh rate control register must be set so that all addresses are refreshed within the t_{REF} period specified in the SDRAM timing specifications. This command is issued to the external bank whether or not it is enabled (EBE in the SDRAM memory global control register). Before executing the auto-refresh command, the SDC executes a precharge all command to the external bank. The next activate command is not given until the t_{RFC} specification ($t_{RFC} = t_{RAS} + t_{RP}$) is met.

Auto-refresh commands are also issued by the SDC as part of the powerup sequence and also after exiting self-refresh mode.

Self-Refresh Mode

The self-refresh mode is controlled by the self-refresh entry and self-refresh exit commands. The SDC must issue a series of commands including the self-refresh entry command to put the SDRAM into this low power operation, and it must issue another series of commands including the self-refresh exit command to re-access the SDRAM.

Self-Refresh Entry Command

The self-refresh entry command causes refresh operations to be performed internally by the SDRAM, without any external control. This means that the SDC does not generate any auto-refresh commands while the SDRAM is in self-refresh mode. Before executing the self-refresh entry command, all internal banks are precharged. The self-refresh entry command is started by writing a 1 to the SRFS bit of the SDRAM memory global control register (EBIU_SDGCTL). The SDC now drives SCKE low.

Only the SCKE pin keeps control during self-refresh, all other SDRAM pins are allowed to be disabled. However the SDC still drives the SCLK during self-refresh mode. However, software may disable the clock by clearing the SCTLE bit in EBIU_SDGCTL.

Self-Refresh Exit Command

Leaving self-refresh mode is performed with the self-refresh exit command, whereby the SDC asserts SCKE. Any internal core/DMA access causes the SDC to perform an exit self-refresh command. The SDC waits to meet the t_{XSR} specification ($t_{XSR} = t_{RAS} + t_{RP}$) and then issues an auto-refresh command. After the auto-refresh command, the SDC waits for the t_{RFC} specification ($t_{RFC} = t_{RAS} + t_{RP}$) to be met before executing the activate command for the transfer that caused the SDRAM to exit self-refresh mode. Therefore, the latency from when a transfer is received by the SDC while in self-refresh mode, until the activate command occurs for that transfer, is:

Time to exit self-refresh: $2 \times (t_{RAS} + t_{RP})$

The minimum time between a subsequent self-refresh entry and the self-refresh exit command is at least t_{RAS} cycles. If a self-refresh entry command is issued during any MemDMA transfer, the SDC satisfies this core request with the minimum self-refresh period (t_{RAS}).

The application software should ensure that all applicable clock timing specifications are met before the transfer to SDRAM address space which causes the controller to exit self-refresh mode. If a transfer occurs to SDRAM address space when the SCTLE bit is cleared, an internal bus error is generated, and the access does not occur externally, leaving the SDRAM in self-refresh mode. For more information, see “Error Detection” on page 6-7.

No Operation Command

The No Operation (NOP) command to the SDRAM has no effect on operations currently in progress. The command inhibit command is the same as a NOP command; however, the SDRAM is not chip-selected.

When the SDC is actively accessing the SDRAM to insert additional wait states, the NOP command is given. When the SDC is not accessing the SDRAM, the command inhibit command is given (`SMS = 1`).

SDC SA10 Pin

The SDRAM's `A[10]` pin follows the truth table below:

- During the precharge command, it is used to indicate a precharge all
- During a bank activate command, it outputs the row address bit
- During read and write commands, it is used to disable auto-precharge

Therefore, the SDC uses a separate `SA10` pin with these rules.

Connect the `SA10` pin with the SDRAM's `A[10]` pin. Because the ADSP-BF537 uses byte addressing, it starts with `A[1]`. The `A[11]` pin is left unconnected for SDRAM accesses and it is replaced by the `SA10` pin.

SDC Programming Model

The following sections provide programming model information for the SDC.

SDC Configuration

After a processor's hardware or software reset, the SDC clocks are enabled; however, the SDC must be configured and initialized. Before programming the SDC and executing the powerup sequence, these steps are required:

1. Ensure the clock to the SDRAM is stable after the power has stabilized for the proper amount of time (typically 100 µs).
2. Write to the SDRAM refresh rate control register (EBIU_SDRRC).
3. Write to the SDRAM memory bank control register (EBIU_SDBCTL).
4. Write to and SDRAM memory global control register (EBIU_SDGCTL).
5. Perform SDRAM access.

The SDRS bit of the SDRAM control status register can be checked to determine the current state of the SDC. If this bit is set, the SDRAM powerup sequence has not been initiated.

The RDIV field of the EBIU_SDRRC register should be written to set the SDRAM refresh rate.

The EBIU_SDBCTL register should be written to describe the sizes and SDRAM memory configuration used (EBSZ and EBCAW) and to enable the external bank (EBE). Note until the SDRAM powerup sequence has been started, any access to SDRAM address space, regardless of the state of the EBE bit, generates an internal bus error, and the access does not occur externally. For more information, see "Error Detection" on page 6-7.

The powerup latency can be estimated as:

$$t_{RP} + (8 \times t_{RFC}) + t_{MRD} + t_{RCD}$$

After the SDRAM powerup sequence has completed, if the external bank is disabled, any transfer to it results in a hardware error interrupt, and the SDRAM transfer does not occur.

The EBIU_SDGCTL register is written:

- To set the SDRAM cycle timing options (CL, TRAS, TRP, TRCD, TWR, EBUFE)
- To enable the SDRAM clock (SCTLE)
- To select and enable the start of the SDRAM powerup sequence (PSM, PSSE)

Note if SCTLE is disabled, any access to SDRAM address space generates an internal bus error and the access does not occur externally. For more information, see "Error Detection" on page 6-7.

Once the PSSE bit in the EBIU_SDGCTL register is set to 1, and a transfer occurs to enabled SDRAM address space, the SDC initiates the SDRAM powerup sequence. The exact sequence is determined by the PSM bit in the EBIU_SDGCTL register. The transfer used to trigger the SDRAM powerup sequence can be either a read or a write. This transfer occurs when the SDRAM powerup sequence has completed. This initial transfer takes many cycles to complete since the SDRAM powerup sequence must take place.

Example SDRAM System Block Diagrams

Figure 6-12 shows a block diagram of an SDRAM interface. In this example, the SDC connected to 2 x (8M x 8) = 8M x 16 to form one external bank of 128Mbit / 16Mbyte of memory. The system's page size is 1024 bytes. The same address and control bus feeds both SDRAM devices.

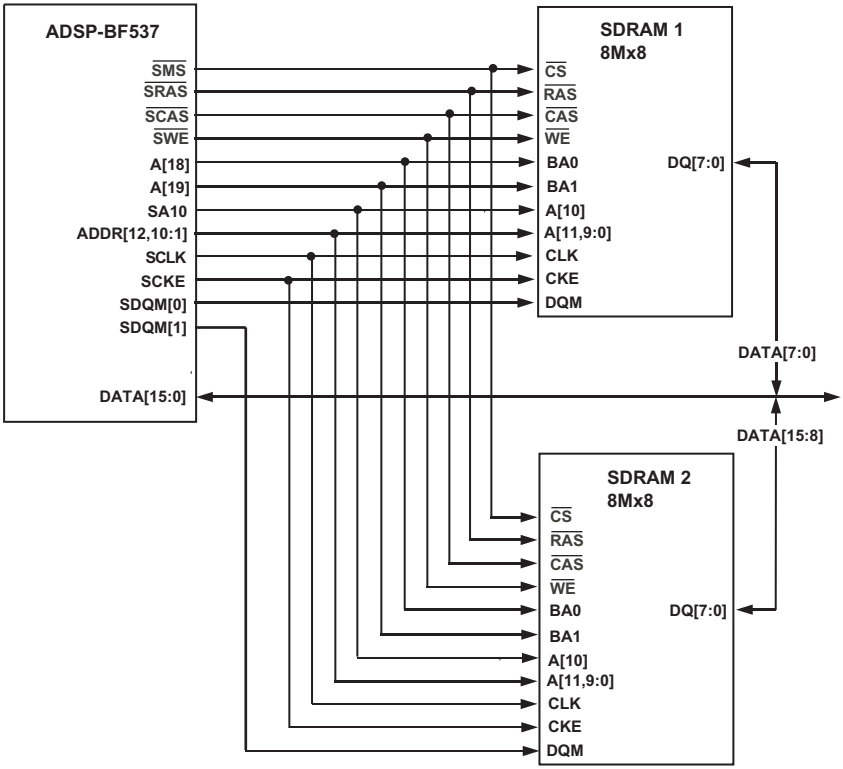


Figure 6-12. SDRAM System Block Diagram, Example 1

Figure 6-13 shows a block diagram of an SDRAM interface. In this example, the SDC connected to 4 x (16M x 4) = 16M x 16 to form one external bank of 256Mbit / 32Mbyte of memory. The system's page size is 2048 bytes. The same address and control bus pass a registered buffer before they feed all 4 SDRAM devices.

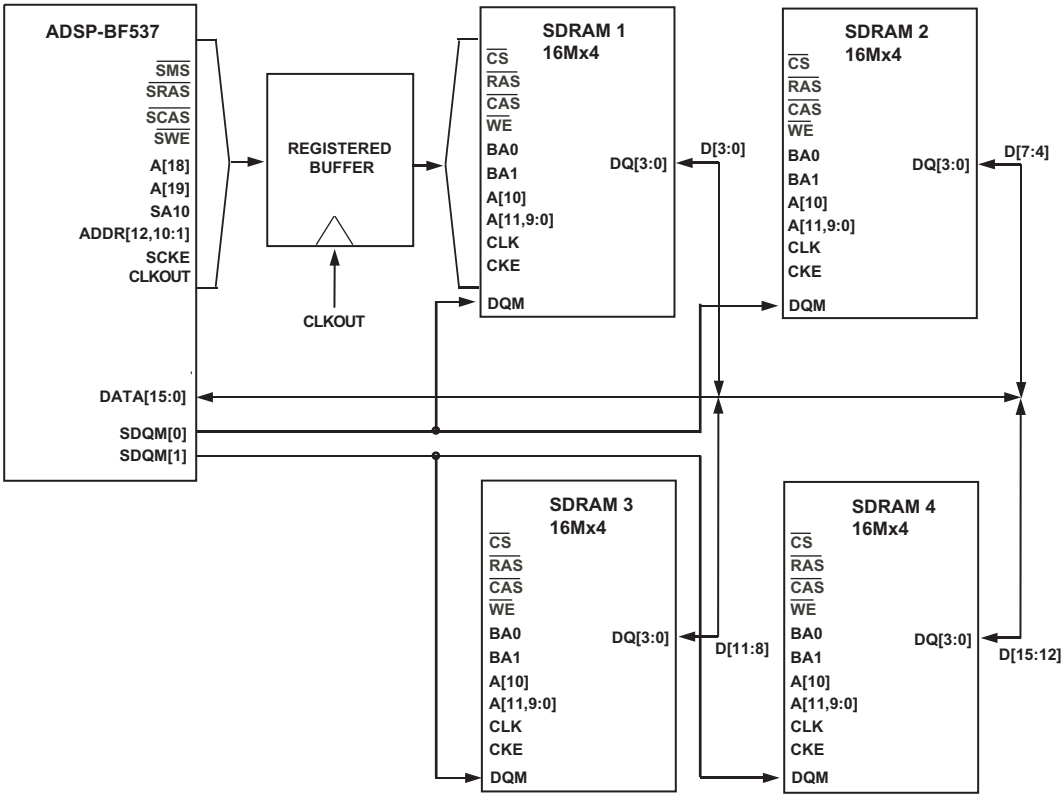


Figure 6-13. SDRAM System Block Diagram, Example 2

Furthermore, the EBUFE bit should be used to enable or disable external buffer timing. When buffered SDRAM modules or discrete register-buffers are used to drive the SDRAM control inputs, EBUFE should be set to 1. Using this setting adds a cycle of data buffering to read and write accesses.

SDC Register Definitions

The following sections describe the SDC registers.

EBIU_SDRRC Register

The SDRAM refresh rate control register (EBIU_SDRRC, shown in Figure 6-14) provides a flexible mechanism for specifying the auto-refresh timing. Since the clock supplied to the SDRAM can vary, the SDC provides a programmable refresh counter, which has a period based on the value programmed into the RDIV field of this register. This counter coordinates the supplied clock rate with the SDRAM device's required refresh rate.

The desired delay (in number of SDRAM clock cycles) between consecutive refresh counter time-outs must be written to the RDIV field. A refresh counter time-out triggers an auto-refresh command to all external SDRAM devices. Write the RDIV value to the EBIU_SDRRC register before the SDRAM powerup sequence is triggered. Change this value only when the SDC is idle.

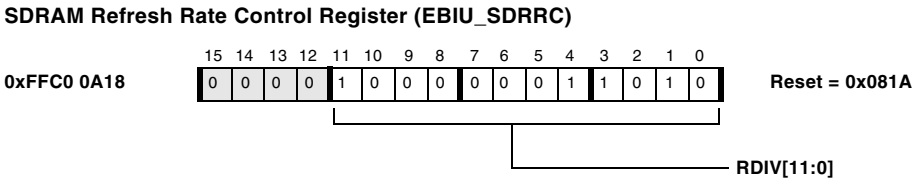


Figure 6-14. SDRAM Refresh Rate Control Register

To calculate the value that should be written to the EBIU_SDRRC register, use the following equation:

$$\text{RDIV} = ((f_{SCLK} \times t_{REF}) / \text{NRA}) - (t_{RAS} + t_{RP})$$

$$= (f_{SCLK} \times t_{REFI}) - (t_{RAS} + t_{RP})$$

Where:

- f_{SCLK} = SDRAM clock frequency (system clock frequency)
- t_{REF} = SDRAM row refresh period
- t_{REFI} = SDRAM row refresh interval
- NRA = Number of row addresses in SDRAM (refresh cycles to refresh whole SDRAM)
- t_{RAS} = Active to precharge time (TRAS in the SDRAM memory global control register) in number of clock cycles
- t_{RP} = RAS to precharge time (TRP in the SDRAM memory global control register) in number of clock cycles

This equation calculates the number of clock cycles between required refreshes and subtracts the required delay between bank activate commands to the same internal bank ($t_{RC} = t_{RAS} + t_{RP}$). The t_{RC} value is subtracted, so that in the case where a refresh time-out occurs while an SDRAM cycle is active, the SDRAM refresh rate specification is guaranteed to be met. The result from the equation should always be rounded down to an integer.

Below is an example of the calculation of RDIV for a typical SDRAM in a system with a 133 MHz clock:

f_{SCLK} = 133 MHz

t_{REF} = 64 ms

NRA = 8192 row addresses

t_{RAS} = 6

t_{RP} = 3

The equation for RDIV yields:

$$RDIV = ((133 \times 10^{6} \times 64 \times 10^{-3}) / 8192) - (6 + 3) = 1030 \text{ clock cycles}$$

This means RDIV is 0x406 (hex) and the SDRAM refresh rate control register should be written with 0x406.

Note RDIV must be programmed to a nonzero value if the SDRAM controller is enabled. When RDIV = 0, operation of the SDRAM controller is not supported and can produce undesirable behavior. Values for RDIV can range from 0x001 to 0xFFF.

EBIU_SDBCTL Register

The SDRAM memory bank control register (EBIU_SDBCTL), shown in Figure 6-15, includes external bank-specific programmable parameters. It allows software to control some parameters of the SDRAM. The external bank can be configured for a different size of SDRAM. It uses the access timing parameters defined in the SDRAM memory global control register (EBIU_SDGCTL). The EBIU_SDBCTL register should be programmed before powerup and should be changed only when the SDC is idle.

- **External bank enable** (EBE)

  The EBE bit is used to enable or disable the external SDRAM bank. If the SDRAM is disabled, any access to the SDRAM address space generates an internal bus error, and the access does not occur externally. For more information, see “Error Detection” on page 6-7.

- **External bank size** (EBSZ)

  The EBSZ encoding stores the configuration information for the SDRAM bank interface. The EBIU supports 64 Mbit, 128 Mbit, 256 Mbit, and 512 Mbit SDRAM devices with x4, x8, and x16

configurations. Table 6-13 maps SDRAM density and I/O width. See "SDRAM External Bank Size" on page 6-27 for more information on bank starting address decodes.

- **External bank column address width** (`EBCAW`)

  The SDC determines the internal SDRAM page size from the `EBCAW` parameters. Page sizes of 512 B, 1K byte, 2K byte, and 4K byte are supported. Table 6-13 shows the page size and breakdown of the internal address (`IA[31:0]`, as seen from the core or DMA) into the row, bank, column, and byte address. The bank width in all cases is 16 bits. The column address and the byte address together make up the address inside the page.

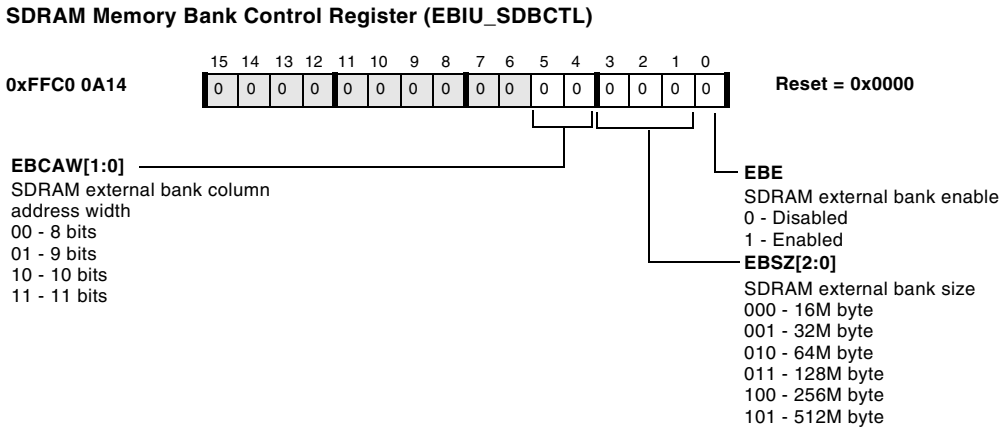


Figure 6-15. SDRAM Memory Bank Control Register

The page size can be calculated for 16-bit SDRAM banks with this formula:

$$\text{page size} = 2^{(CAW + 1)}$$

where CAW is the column address width of the SDRAM, plus 1 because the SDRAM bank is 16 bits wide (1 address bit = 2 bytes).

Table 6-13. Internal Address Mapping

Bank Size (Mbyte) EBSZ bits	Col. Addr. Width (CAW) EBCAW bits	Page Size (K Byte)	Bank Address	Row Address	Page	
					Column Address	Byte Address
512	11	4	IA[28:27]	IA[26:12]	IA[11:1]	IA[0]
512	10	2	IA[28:27]	IA[26:11]	IA[10:1]	IA[0]
256	11	4	IA[27:26]	IA[25:12]	IA[11:1]	IA[0]
256	10	2	IA[27:26]	IA[25:11]	IA[10:1]	IA[0]
256	9	1	IA[27:26]	IA[25:10]	IA[9:1]	IA[0]
128	11	4	IA[26:25]	IA[24:12]	IA[11:1]	IA[0]
128	10	2	IA[26:25]	IA[24:11]	IA[10:1]	IA[0]
128	9	1	1A[26:25]	IA[24:10]	IA[9:1]	IA[0]
128	8	.5	IA[26:25]	IA[24:9]	IA[8:1]	IA[0]
64	11	4	IA[25:24]	IA[23:12]	IA[11:1]	IA[0]
64	10	2	IA[25:24]	IA[23:11]	IA[10:1]	IA[0]
64	9	1	IA[25:24]	IA[23:10]	IA[9:1]	IA[0]
64	8	.5	IA[25:24]	IA[23:9]	IA[8:1]	IA[0]
32	11	4	IA[24:23]	IA[22:12]	IA[11:1]	IA[0]
32	10	2	IA[24:23]	IA[22:11]	IA[10:1]	IA[0]
32	9	1	IA[24:23]	IA[22:10]	IA[9:1]	IA[0]

Table 6-13. Internal Address Mapping (Cont'd)

Bank Size (Mbyte) EBSZ bits	Col. Addr. Width (CAW) EBCAW bits	Page Size (K Byte)	Bank Address	Row Address	Page	
					Column Address	Byte Address
32	8	.5	IA[24:23]	IA[22:9]	IA[8:1]	IA[0]
16	11	4	IA[23:22]	IA[21:12]	IA[11:1]	IA[0]
16	10	2	IA[23:22]	IA[21:11]	IA[10:1]	IA[0]
16	9	1	IA[23:22]	IA[21:10]	IA[9:1]	IA[0]
16	8	.5	IA[23:22]	IA[21:9]	IA[8:1]	IA[0]

The minimum column width for a 256M byte SDRAM configuration is 9 and the minimum column width for a 512M byte SDRAM configuration is 10.

Using SDRAMs With Systems Smaller than 16M byte

It is possible to use SDRAMs smaller than 16M byte on the ADSP-BF534, ADSP-BF536, and ADSP-BF537, as long as it is understood how the resulting memory map is altered. Figure 6-16 shows an example where a 2M byte SDRAM (512K x 16 bits x 2 banks) is mapped to the external memory interface. In this example, there are 11 row addresses and 8 column addresses per bank. Referring to Table 6-4 on page 6-26, the lowest available bank size (16M byte) for a device with 8 column addresses has 2 bank address lines (IA[23:22]) and 13 row address lines (IA[21:9]). Therefore, 1 processor bank address line and 2 row address lines are unused when hooking up to the SDRAM in the example. This causes aliasing in the processor's external memory map, which results in the SDRAM being mapped into noncontiguous regions of the processor's memory space.

Referring to the table in Figure 6-16, note that each line in the table corresponds to 2^{19} bytes, or 512K byte. Thus, the mapping of the 2M byte SDRAM is noncontiguous in Blackfin memory, as shown by the memory mapping in the left side of the figure.

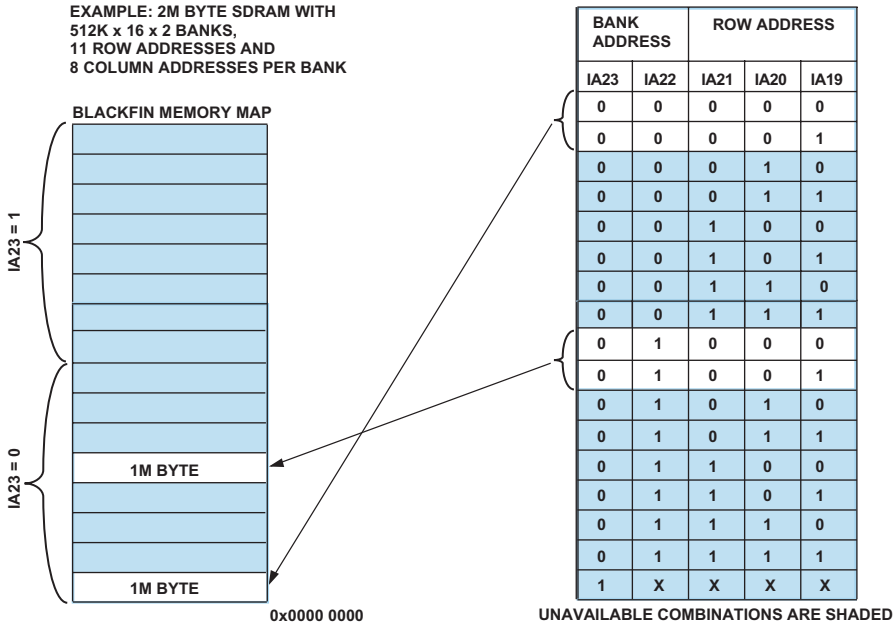


Figure 6-16. Using Small SDRAMs

EBIU_SDGCTL Register

The SDRAM memory global control register (EBIU_SDGCTL) includes all programmable parameters associated with the SDRAM access timing and configuration. Figure 6-17 shows the EBIU_SDGCTL register bit definitions.

- **SDRAM clock enable** (SCTLE)

  The SCTLE bit is used to enable or disable the SDC. If SCTLE is disabled, any access to SDRAM address space generates an internal bus error, and the access does not occur externally. For more information, see "Error Detection" on page 6-7. When SCTLE is disabled, all SDC control pins are in their inactive states and the SDRAM clock is not running. The SCTLE bit must be enabled for SDC operation and is enabled by default at reset. The CAS latency (CL), SDRAM t_{RAS} timing (TRAS), SDRAM t_{RP} timing (TRP), SDRAM t_{RCD} timing (TRCD), and SDRAM t_{WR} timing (TWR) bits should be programmed based on the system clock frequency and the timing specifications of the SDRAM used.

  The SCTLE bit allows software to disable all SDRAM control pins. These pins are SDQM[3:0], $\overline{\text{SCAS}}$, $\overline{\text{SRAS}}$, $\overline{\text{SWE}}$, SCKE, and CLKOUT.

  - SCTLE = 0
    Disable all SDRAM control pins (control pins negated, CLKOUT low).

  - SCTLE = 1
    Enable all SDRAM control pins (CLKOUT toggles).

  Note the CLKOUT function is also shared with the AMC. Even if SCTLE is disabled, CLKOUT can be enabled independently by the CLKOUT enable in the AMC (AMCKEN in the EBIU_AMGCTL register).

  If the system does not use SDRAM, SCTLE should be set to 0.

SDRAM Memory Global Control Register (EBIU_SDGCTL)

0xFFC0 0A10

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
1	1	1	0	0	0	0	0	0	0	0	0	1	0	0	0

Reset = 0xE008 8849

CDDBG
Control disable during bus grant
0 - Continue driving SDRAM controls during bus grant
1 - Three-state SDRAM controls during bus grant

TCSR
Temperature compensated self-refresh value in extended mode register
0 - 45 degrees C
1 - 85 degrees C

EMREN
Extended mode register enable
0 - Disabled
1 - Enabled

FBBRW
Fast back-to-back read to write
0 - Disabled
1 - Enabled

EBUFE
SDRAM timing for external buffering of address and control
0 - External buffering timing disabled
1 - External buffering timing enabled

SRFS
SDRAM self-refresh enable
0 - Disable self-refresh
1 - Enable self-refresh during inactivity

TRCD[2:1]
SDRAM t_{RCD} in SCLK cycles
000 - Reserved
001-111 - 1 to 7 cycles

TWR[1:0]
SDRAM t_{WR} in SCLK cycles
00 - Reserved
01-11 - 1 to 3 cycles

PUPSD
Powerup start delay
0 - No extra delay added before first Precharge command
1 - Fifteen SCLK cycles of delay before first Precharge command

PSM
SDRAM powerup sequence
0 - Precharge, 8 CBR refresh cycles, mode register set
1 - Precharge, mode register set, 8 CBR refresh cycles

PSSE
SDRAM powerup sequence start enable. Always reads 0
0 - No effect
1 - Enables SDRAM powerup sequence on next SDRAM access

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

TRCD[0]
SDRAM t_{RCD} in SCLK cycles
000 - Reserved
001-111 - 1 to 7 cycles

TRP[2:0]
SDRAM t_{RP} in SCLK cycles
000 - No effect
001-111 - 1 to 7 cycles

TRAS[3:0]
SDRAM t_{RAS} in SCLK cycles
0000 - No effect
0001-1111 - 1 to 15 cycles

SCTLE
Enable CLKOUT, $\overline{\text{SRAS}}$, $\overline{\text{SCAS}}$, $\overline{\text{SWE}}$, SDQM[1:0]
0 - Disabled
1 - Enabled

CL[1:0]
SDRAM CAS latency
00–01 - Reserved
10 - 2 cycles
11 - 3 cycles

PASR[1:0]
Partial array self-refresh in extended mode register
00 - All 4 banks refreshed
01 - Int banks 0, 1 refreshed
10 - Int bank 0 only refreshed
11 - Reserved

Figure 6-17. SDRAM Memory Global Control Register

If an access occurs to the SDRAM address space while SCTLE is 0, the access generates an internal bus error and the access does not occur externally. For more information, see “Error Detection” on page 6-7.

With careful software control, the SCTLE bit can be used in conjunction with the SRFS bit to further lower power consumption by freezing the CLKOUT pin. However, SCTLE must remain enabled at all times when the SDC is needed to generate auto-refresh commands to SDRAM.

- **CAS latency** (CL)

  The CL bits in the SDRAM memory global control register (EBIU_SDGCTL) select the CAS latency value:

  - CL = 00
    Reserved
  - CL = 01
    Reserved
  - CL = 10
    2 clock cycles
  - CL = 11
    3 clock cycles

- **Partial array self refresh** (PASR)

  The PASR bits determine how many internal SDRAM banks are refreshed during self-refresh.

  - PASR = 00
    All 4 banks
  - PASR = 01
    Internal banks 0 and 1 refreshed
  - PASR = 10
    Only internal bank 0 refreshed
  - PASR = 11
    reserved

  Internal banks are decoded with the A[19:18] pins.

  The PASR feature requires careful software control with regard to the internal bank used.

- **Bank activate command delay** (TRAS)

  The TRAS bits in the SDRAM memory global control register (EBIU_SDGCTL) select the t_{RAS} value. Any value between 1 and 15 clock cycles can be selected. For example:

  - TRAS = 0000
    No effect
  - TRAS = 0001
    1 clock cycle
  - TRAS = 0010
    2 clock cycles
  - TRAS = 1111
    15 clock cycles

- **Bank precharge delay** (TRP)

  The TRP bits in the SDRAM memory global control register (EBIU_SDGCTL) select the t_{RP} value. Any value between 1 and 7 clock cycles may be selected. For example:

  - TRP = 000
    No effect
  - TRP = 001
    1 clock cycle
  - TRP = 010
    2 clock cycles
  - TRP = 111
    7 clock cycles

- **RAS to CAS delay** (TRCD)

  The TRCD bits in the SDRAM memory global control register (EBIU_SDGCTL) select the t_{RCD} value Any value between 1 and 7 clock cycles may be selected. For example:

  - TRCD = 000
    Reserved, no effect
  - TRCD = 001
    1 clock cycle
  - TRCD = 010
    2 clock cycles
  - TRCD = 111
    7 clock cycles

- **Write to precharge delay** (TWR)

  The TWR bits in the SDRAM memory global control register (EBIU_SDGCTL) select the t_{WR} value. Any value between 1 and 3 clock cycles may be selected. For example:

  - TWR = 00
    Reserved
  - TWR = 01
    1 clock cycle
  - TWR = 10
    2 clock cycles
  - TWR = 11
    3 clock cycles

- **Powerup start delay** (PUPSD)

  The powerup start delay bit (PUPSD) optionally delays the powerup start sequence for 15 SCLK cycles. This is useful for multiprocessor systems sharing an external SDRAM. If the bus has been previously granted to the other processor before powerup and self-refresh mode is used when switching bus ownership, then the PUPSD bit can be used to guarantee a sufficient period of inactivity from self-refresh to the first Precharge command in the powerup sequence in order to meet the exit self-refresh time (t_{XSR}) of the SDRAM.

- **Powerup sequence mode** (PSM)

  If the PSM bit is set to 1, the SDC command sequence is:

  1. Precharge all
  2. Mode register set
  3. 8 auto-refresh cycles

If the PSM bit is set to 0, the SDC command sequence is:

1. Precharge all
2. 8 auto-refresh cycles
3. Mode register set

- **Powerup sequence start enable** (PSSE)

  The PSM and PSSE bits work together to specify and trigger an SDRAM powerup (initialization) sequence. Two events must occur before the SDC does the SDRAM powerup sequence:

  - The PSSE bit must be set to 1 to enable the SDRAM powerup sequence.
  - A read or write access must be done to enabled SDRAM address space in order to have the external bus granted to the SDC so that the SDRAM powerup sequence may occur.

  The SDRAM powerup sequence occurs and is followed immediately by the read or write transfer to SDRAM that was used to trigger the SDRAM powerup sequence. Note there is a latency for this first access to SDRAM because the SDRAM powerup sequence takes many cycles to complete.

Before executing the SDC powerup sequence, ensure that the SDRAM receives stable power and is clocked for the proper amount of time, as specified by the SDRAM specification.

- **Self-refresh setting** (SRFS)

  The SRFS and SCTLE bits work together in EBIU_SDGCTL for self-refresh control:

  - SRFS = 0
    Disable self-refresh mode
  - SRFS = 1
    Enter self-refresh mode

  When SRFS is set to 1, self-refresh mode is triggered. Once the SDC completes any active transfers, the SDC executes a sequence of commands to put the SDRAM into self-refresh mode.

  When the device comes out of reset, the SCKE pin is driven high. If it is necessary to enter self-refresh mode after reset, program SRFS = 1.

Enter Self-Refresh Mode

When SRFS is set to 1, once the SDC enters an idle state it issues a precharge all command and then issues a self-refresh entry command. If an internal access is pending, the SDC delays issuing the self-refresh entry command until it completes the pending SDRAM access and any subsequent pending access requests.

Once the SDRAM device enters into self-refresh mode, the SDRAM controller asserts the SDSRA bit in the SDRAM control status register (EBIU_SDSTAT).

Note once the SRFS bit is set to 1, the SDC enters self-refresh mode when it finishes pending accesses. There is no way to cancel the entry into self-refresh mode.

Before disabling the CLKOUT pin with the SCTLE bit, be sure to place the SDC in self-refresh mode (SRFS bit). If this is not done, the SDRAM is unclocked and will not work properly.

Exit Self-Refresh Mode

The SDRAM device exits self-refresh mode only when the SDC receives core or DMA requests. In conjunction with the SRFS bit, 2 possibilities are given to exit self-refresh mode:

1. If the SRFS bit keeps set before the core/DMA request, the SDC exits self-refresh mode temporarily for a single request and returns back to self-refresh mode until a new request is latched.

2. If the SRFS bit is cleared before the core/DMA request, the SDC exits self-refresh mode and returns to auto-refresh mode.

Before exiting self-refresh mode with the SRFS bit, be sure to enable the CLKOUT pin (SCTLE bit). If this is not done, the SDRAM is unclocked and will not work properly.

- **External buffering enabled** (EBUFE)

  With the total I/O width of 16 bits, a maximum of 4x4 bits can be connected in parallel in order to increase the system's overall page size.

  To meet overall system timing requirements, systems that employ several SDRAM devices connected in parallel may require buffering between the processor and multiple SDRAM devices. This buffering generally consists of a register and driver.

  To meet such timing requirements and to allow intermediary registration, the SDC supports pipelining of SDRAM address and control signals.

The EBUFE bit in the EBIU_SDGCTL register enables this mode:

- EBUFE = 0
  Disable external buffering timing
- EBUFE = 1
  Enable external buffering timing

When EBUFE = 1, the SDRAM controller delays the data in write accesses by one cycle, enabling external buffer registers to latch the address and controls. In read accesses, the SDRAM controller samples data one cycle later to account for the one-cycle delay added by the external buffer registers. When external buffering timing is enabled, the latency of all accesses is increased by one cycle.

Connection of 4 x 4 bits rather than 1 x 16 bits increases the page size by a factor of 4, thus resulting in fewer off page penalties.

- **Fast back to back read to write** (FBBRW)

  The FBBRW bit enables an SDRAM read followed by write to occur on consecutive cycles. In many systems, this is not possible because the turn-off time of the SDRAM data pins is too long, leading to bus contention with the succeeding write from the processor. When this bit is 0, a clock cycle is inserted between read accesses followed immediately by write accesses.

- **Extended mode register enabled** (EMREN)

  The EMREN bit enables programming of the extended mode register during startup. The extended mode register is used to control SDRAM power consumption in certain mobile low power SDRAMs. If the EMREN bit is enabled, then the TCSR and PASR[1:0] bits control the value written to the extended mode register.

- **Temperature compensated self-refresh** (TCSR)

  The TCSR bit signals to the SDRAM the worst case temperature range for the system, and thus how often the SDRAM internal banks need to be refreshed during self-refresh.

- **Control disable during bus grant** (CDDBG)

  The CDDBG bit is used to enable or disable the SDRAM control signals when the external memory interface is granted to an external controller. If this bit is set to a 1, then the control signals are three-stated when bus grant is active. Otherwise, these signals continue to be driven during grant. If the bit is set and the external bus is granted, all SDRAM internal banks are assumed to have been changed by the external controller. This means a precharge is required on each bank prior to use after control of the external bus is re-established. The control signals affected by this pin are $\overline{\text{SRAS}}$, $\overline{\text{SCAS}}$, $\overline{\text{SWE}}$, $\overline{\text{SMS}}$, SA10, SCKE, and CLKOUT.

Note all reserved bits in this register must always be written with 0s.

EBIU_SDSTAT Register

The SDRAM control status register (EBIU_SDSTAT), shown in Figure 6-18, provides information on the state of the SDC. This information can be used to determine when it is safe to alter SDC control parameters or it can be used as a debug aid.

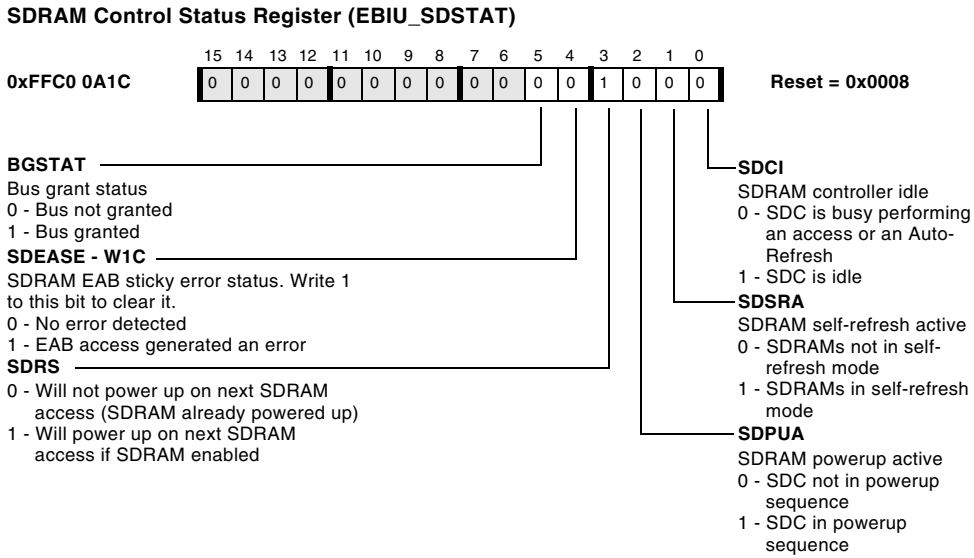


Figure 6-18. SDRAM Control Status Register

- **SDC idle** (SDCI)

  If the SDCI bit is 0, the SDC is performing a user access or auto-refresh. If the SDCI bit is 1, no commands are issued and the SDC is in idle state.

- **SDC self-refresh active** (SDSRA)

  If the SDSRA bit is 0, the SDC is performing auto-refresh (SCKE pin = 0). If the SDSRA bit is 1, the SDC performs self-refresh mode (SCKE pin = 1).

- **SDC powerup active** (SDPUA)

  If the SDPUA bit is 0, the SDC is not in powerup sequence. If the SDPUA bit is 1, the SDC performs the powerup sequence.

- **SDC powerup delay** (SDRS)

  If the SDRS bit is 0, the SDC has already powered up. If the SDRS bit is 1, the SDC will still perform the powerup sequence.

- **SDC EAB sticky error status** (SDEASE)

  If the SDEASE bit is 0, there were no errors detected on the EAB core bus. If the SDEASE bit is 1, there were errors detected on the EAB core bus. The SDEASE bit is sticky. Once it has been set, software must explicitly write a 1 to the bit to clear it. Writes have no effect on the other status bits, which are updated by the SDC only.

- **Bus grant status** (BGSTAT).

  If the BGSTAT bit is 0, the bus is not granted. If the BGSTAT bit is 1, the bus is granted.

SDC Programming Examples

Listing 6-3 through Listing 6-6 provide examples for working with the SDC.

Listing 6-3. 16-Bit Core Transfers to SDRAM

```
.section L1_data_b;
.byte2 source[N] = 0x1122, 0x3344, 0x5566, 0x7788;
.section SDRAM;
.byte2 dest[N];
.section L1_code;
  I0.L = lo(source);
```

```
  I0.H = hi(source);
  I1.L = lo(dest);
  I1.H = hi(dest);
      R0.L = w[I0++];
      p5=N-1;
      lsetup(lp, lp) lc0=p5;
lp:R0.L = w[I0++] || w[I1++] = R0.L;
                            w[I1++] = R0.L;
```

Listing 6-4. 8-Bit Core Transfers to SDRAM Using Byte Mask SDQM[1:0] Pins

```
.section L1_data_b;
.byte source[N] = 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88;

.section SDRAM;
.byte dest[N];

  p0.L = lo(source);
  p0.H = hi(source);
  p1.L = lo(dest);
  p1.H = hi(dest);
    p5=N;
    lsetup(start, end) lc0=p5;
start:  R0 = b[p0++](z);
end:    b[p1++] = R0;  /* byte data masking */
```

Listing 6-5. Self-Refresh Mode Power Savings With Disabled CLKOUT

```
    r0.l = w[I1++];    /* SDRAM access */
  ssync;    /* force last SDRAM access to finish */
  P0.L = lo(EBIU_SDGCTL);
  P0.H = hi(EBIU_SDGCTL);
  R1 = [P0];
  bitset(R1, bitpos(SRFS));    /* enter self-refresh */
  [P0] = R1;
  ssync;

  P0.L = lo(EBIU_SDSTAT);
  P0.H = hi(EBIU_SDSTAT);
  R0 = [P0];
  ssync;

self_refresh_status:
  cc = bittst(R0, bitpos(SDSRA));  /* poll self-refresh status */
  if !cc jump self_refresh_status;

  P0.L = lo(EBIU_SDGCTL);
  P0.H = hi(EBIU_SDGCTL);
  R1 = [P0];
  bitclr(R1, bitpos(SCTLE));  /* disable CLKOUT after approx 20
cycles */
  [P0] = R1;
  ssync;

          P5 = 30000;
          LSETUP(lp,lp) LC0 = P5;
    lp:   nop;  /* dummy loop */

  R1 = [P0];
  bitset(R1, bitpos(SCTLE));  /* enable CLKOUT after approx 20
cycles */
```

```
  [P0] = R1;
  ssync;

  R1 = [P0];
  bitclr(R1, bitpos(SRFS));  /* exit self-refresh */
  [P0] = R1;
  ssync;

    w[I1++] = r0.l;  /* SDRAM access */
```

Listing 6-6. Init

```
/*********************************************************/
/* SDRAM part# Micron MT48LC32M8A2-75 (32Mx8/256Mbit) */
/* 8k rows, 1k columns         -> EBCAW = 10 */
/* 2xSDRAM: 32Mx16 = 64Mbytes  -> EBSZ = 010 */

/* populated SDRAM addresses   -> 0x00000000 - 0x01FFFFFF */
/* internal SDRAM bank A  0x00000000 - 0x007FFFFF */
/* internal SDRAM bank B  0x00800000 - 0x00FFFFFF */
/* internal SDRAM bank C  0x01000000 - 0x017FFFFF */
/* internal SDRAM bank D  0x01800000 - 0x01FFFFFF */

/* powerup: PRE-REF-MRS   -> PSM = 0 */
/* SCLK = 133 MHz */
/* tCK = 7.5ns min@CL=3    -> CL = 3 */
/* tRAS = 44ns min         -> TRAS = 6 */
/* tRP = 20ns min          -> TRP = 3 */
/* tRCD = 20ns min         -> TRCD = 3 */
/* tWR = 15ns min          -> TWR = 2 */
/* tREF = 64ms max  ->RDIV = (133MHz*64ms)/8192-(6+3)=0x406
cycles */
/*********************************************************/
#ifdef INIT_SDRAM
```

```
  /* Check if already enabled */
  p0.l = lo(EBIU_SDSTAT);
  p0.h = hi(EBIU_SDSTAT);
  r0 = [p0];
  cc = bittst(r0, bitpos(SDRS));
  if !cc jump skip init_sdram;

  /* SDRAM Refresh Rate Control Register */
  P0.L = lo(EBIU_SDRRC);
  P0.H = hi(EBIU_SDRRC);
  R0.L = 0x0406;
  W[P0] = R0.L;

  /* SDRAM Memory Bank Control Register */
  P0.L = lo(EBIU_SDBCTL);
  P0.H = hi(EBIU_SDBCTL);
  R0.L = 0x0025;
  W[P0] = R0.L;

/* SDRAM Memory Global Control Register */
  P0.L = lo(EBIU_SDGCTL);
  P0.H = hi(EBIU_SDGCTL);
  R0.L = 0x998d;
  R0.H = 0x8491;
  [P0] = R0;
  ssync;  /* wait until executed */
```

7 PARALLEL PERIPHERAL INTERFACE

This chapter describes the Parallel Peripheral Interface (PPI). Following an overview and a list of key features are a description of operation and functional modes of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

Overview

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. The highest system throughput is achieved with 8-bit data, since two 8-bit data samples can be packed as a single 16-bit word. In such a case, the earlier sample is placed in the 8 least significant bits (LSBs).

Features

The PPI includes these features:

- Half duplex, bidirectional parallel port
- Supports up to 16 bits of data
- Programmable clock and frame sync polarities
- ITU-R 656 support
- Interrupt generation on overflow and underrun

Typical peripheral devices that can be interfaced to the PPI port:

- A/D converters
- D/A converters
- LCD panels
- CMOS sensors
- Video encoders
- Video decoders

Interface Overview

Figure 7-1 shows a block diagram of the PPI.

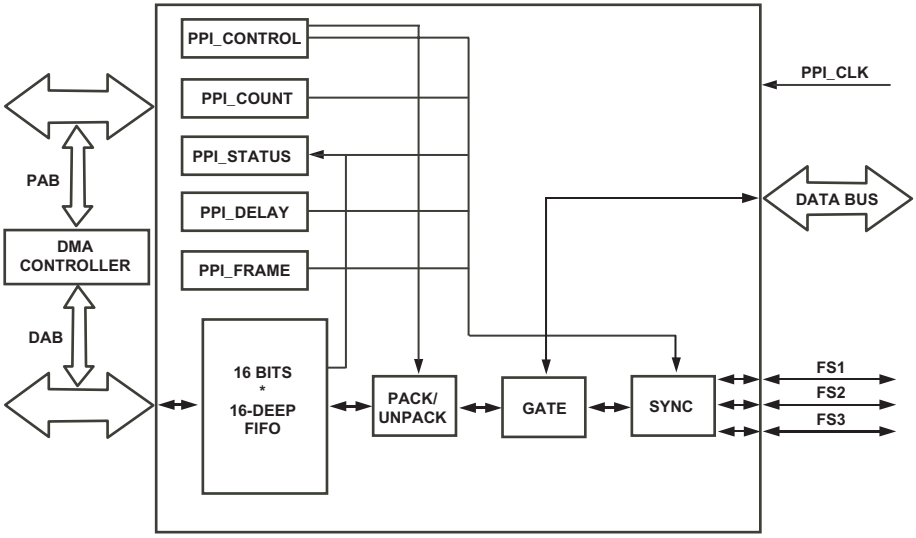


Figure 7-1. PPI Block Diagram

The PPI_CLK pin accepts an external clock input. It cannot source a clock internally.

When the PPI_CLK is not free-running, there may be additional latency cycles before data gets received or transmitted. In RX and TX modes, there may be at least 2 cycles latency before valid data is received or transmitted.

Table 7-1 shows the pin interface for the PPI. Enabling a particular pin involves writing to the appropriate PORTx_FER register and, if applicable, the PORT_MUX register. To configure for particular PPI pin usage, program the PORT_MUX, PORTF_FER, and PORTG_FER MMRs as shown in Table 7-1.

The 16 PPI data pins are found on port G. The upper data lines are multiplexed with SPORT0 signals. While 8-bit PPI operation still enables full SPORT0 functionality, 10-bit PPI configuration disables the secondary data signals of SPORT0. If 13 or fewer data lines are required for PPI operation, the transmit channel of SPORT0 remains fully functional. The three control bits PGSE, PGRE, and PGTE in the PORT_MUX register control this granularity of signal multiplexing.

The PPI clock and the three PPI frame sync signals are found on port F. The PPI_CLK not only supplies the PPI module itself, it also can clock all of the eight timers to work synchronously with the PPI. Depending on PPI operation mode, the PPI_CLK can either equal or invert the TMRCLK input.

The three frame sync signals are multiplexed with the three timer signals TMR0, TMR1, and TMR2. Timer 0 and timer 1 are internally looped back to the PPI module and can therefore be used for internal frame sync generation. If FS1 and FS2 are applied externally, timer 0 and timer 1 must disable their outputs by setting the OUT_DIS bit in the TIMER0_CONFIG and TIMER1_CONFIG registers, when working in PWM_OUT mode. Only the third frame sync input FS3, if used, must be explicitly enabled in the PORT_MUX register by setting the PFFE bit.

All pins of port F and port G function as GPIOs by default and must be individually enabled for either PPI or any other peripheral operation by setting the appropriate bits in the function enable registers PORTF_FER and PORTG_FER. For more information, refer to Chapter 14, "General-Purpose Ports". Since TMR0 and TMR1 are connected to the PPI module internally, the respective pins can be used in GPIO mode, if no external device is listening to the frame syncs.

Table 7-1. PPI Pins

Pin Name (Function)	PORT_MUX	PORTF_FER	PORTG_FER
PPI D0 (PPI data 0)			Set bit 0 (PG0)
PPI D1 (PPI data 1)			Set bit 1 (PG1)
PPI D2 (PPI data 2)			Set bit 2 (PG2)
PPI D3 (PPI data 3)			Set bit 3 (PG3)
PPI D4 (PPI data 4)			Set bit 4 (PG4)
PPI D5 (PPI data 5)			Set bit 5 (PG5)
PPI D6 (PPI data 6)			Set bit 6 (PG6)
PPI D7 (PPI data 7)			Set bit 7 (PG7)
PPI D8 (PPI data 8)	Clear bit 9 (PGSE)		Set bit 8 (PG8)
PPI D9 (PPI data 9)	Clear bit 9 (PGSE)		Set bit 9 (PG9)
PPI D10 (PPI data 10)	Clear bit 10 (PGRE)		Set bit 10 (PG10)
PPI D11 (PPI data 11)	Clear bit 10 (PGRE)		Set bit 11 (PG11)
PPI D12 (PPI data 12)	Clear bit 10 (PGRE)		Set bit 12 (PG12)
PPI D13 (PPI data 13)	Clear bit 11 (PGTE)		Set bit 13 (PG13)
PPI D14 (PPI data 14)	Clear bit 11 (PGTE)		Set bit 14 (PG14)
PPI D15 (PPI data 15)	Clear bit 11 (PGTE)		Set bit 15 (PG15)
PPI_CLK (PPI clock)		Set bit 15 (PF15)	
PPI_FS1 (PPI frame sync 1)		Set bit 9 (PF9)	

Table 7-1. PPI Pins (Cont'd)

Pin Name (Function)	PORT_MUX	PORTF_FER	PORTG_FER
PPI_FS2 (PPI frame sync 2)		Set bit 8 (PF8)	
PPI_FS3 (PPI frame sync 3)	Set bit 8 (PFFE)	Set bit 7 (PF7)	

Description of Operation

Table 7-2 shows all the possible modes of operation for the PPI.

Table 7-2. PPI Possible Operating Modes

PPI Mode	# of Syncs	PORT _DIR	PORT _CFG	XFR_ TYPE	POLC	POLS	FLD_ SEL
RX mode, 0 frame syncs, external trigger	0	0	11	11	0 or 1	0 or 1	0
RX mode, 0 frame syncs, internal trigger	0	0	11	11	0 or 1	0 or 1	1
RX mode, 1 external frame sync	1	0	00	11	0 or 1	0 or 1	0
RX mode, 2 or 3 external frame syncs	3	0	10	11	0 or 1	0 or 1	0
RX mode, 2 or 3 internal frame syncs	3	0	01	11	0 or 1	0 or 1	0
RX mode, ITU-R 656, active field only	embed-ded	0	00	00	0 or 1	0	0 or 1
RX mode, ITU-R 656, vertical blanking only	embed-ded	0	00	10	0 or 1	0	0
RX mode, ITU-R 656, entire field	embed-ded	0	00	01	0 or 1	0	0
TX mode, 0 frame syncs	0	1	00	00	0 or 1	0 or 1	0
TX mode, 1 internal or external frame sync	1	1	00	11	0 or 1	0 or 1	0

Table 7-2. PPI Possible Operating Modes (Cont'd)

PPI Mode	# of Syncs	PORT _DIR	PORT _CFG	XFR_ TYPE	POLC	POLS	FLD_ SEL
TX mode, 2 external frame syncs	2	1	01	11	0 or 1	0 or 1	0
TX mode, 2 or 3 internal frame syncs, FS3 sync'd to FS1 assertion	3	1	01	11	0 or 1	0 or 1	0
TX mode, 2 or 3 internal frame syncs, FS3 sync'd to FS2 assertion	3	1	11	11	0 or 1	0 or 1	0

Functional Description

The following sections describe the function of the PPI.

ITU-R 656 Modes

The PPI supports three input modes for ITU-R 656-framed data. These modes are described in this section. Although the PPI does not explicitly support an ITU-R 656 output mode, recommendations for using the PPI for this situation are provided as well.

ITU-R 656 Background

According to the ITU-R 656 recommendation (formerly known as CCIR-656), a digital video stream has the characteristics shown in Figure 7-2, and Figure 7-3 for 525/60 (NTSC) and 625/50 (PAL) systems. The processor supports only the bit-parallel mode of ITU-R 656. Both 8- and 10-bit video element widths are supported.

In this mode, the Horizontal (H), Vertical (V), and Field (F) signals are sent as an embedded part of the video datastream in a series of bytes that form a control word. The Start of Active Video (SAV) and End of Active

Video (EAV) signals indicate the beginning and end of data elements to read in on each line. SAV occurs on a 1-to-0 transition of H, and EAV begins on a 0-to-1 transition of H. An entire field of video is comprised of active video + horizontal blanking (the space between an EAV and SAV code) and vertical blanking (the space where V = 1). A field of video commences on a transition of the F bit. The “odd field” is denoted by a value of F = 0, whereas F = 1 denotes an even field. Progressive video makes no distinction between field 1 and field 2, whereas interlaced video requires each field to be handled uniquely, because alternate rows of each field combine to create the actual video image.

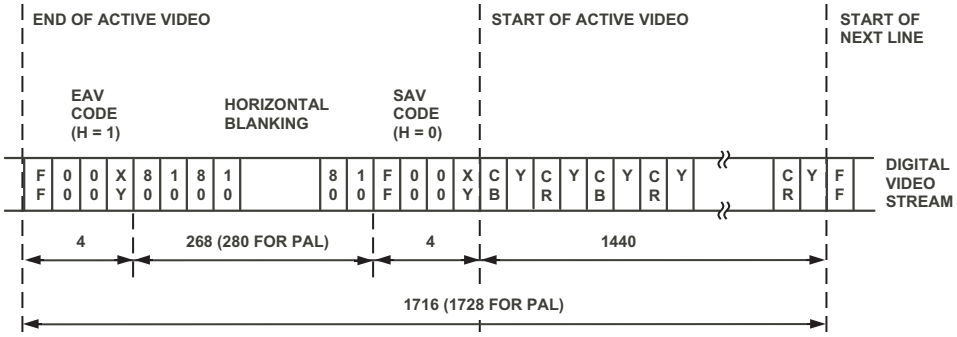
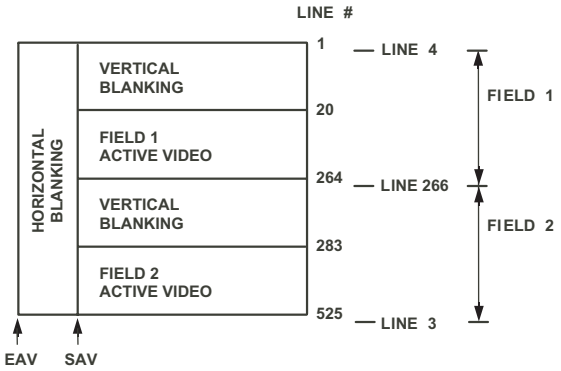
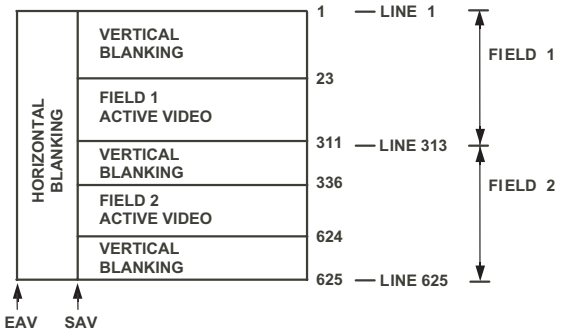


Figure 7-2. ITU-R 656 8-Bit Parallel Data Stream for NTSC (PAL) Systems



LINE NUMBER	F	V	H (EAV)	H (SAV)
1-3, 266-282	1	1	1	0
4-19, 264-265	0	1	1	0
20-263	0	0	1	0
283-525	1	0	1	0



LINE NUMBER	F	V	H (EAV)	H (SAV)
1-22, 311-312	0	1	1	0
23-310	0	0	1	0
313-335, 624-625	1	1	1	0
336-623	1	0	1	0

Figure 7-3. Typical Video Frame Partitioning for NTSC/PAL Systems for ITU-R BT.656-4

The SAV and EAV codes are shown in more detail in Table 7-3. Note there is a defined preamble of three bytes (0xFF, 0x00, 0x00), followed by the XY status word, which, aside from the `F` (field), `V` (vertical blanking) and `H` (horizontal blanking) bits, contains four protection bits for single-bit error detection and correction. Note `F` and `V` are only allowed to change as part of EAV sequences (that is, transition from `H = 0` to `H = 1`). The bit definitions are as follows:

- `F = 0` for field 1
- `F = 1` for field 2

- `V = 1` during vertical blanking
- `V = 0` when not in vertical blanking
- `H = 0` at SAV
- `H = 1` at EAV
- `P3 = V XOR H`
- `P2 = F XOR H`
- `P1 = F XOR V`
- `P0 = F XOR V XOR H`

In many applications, video streams other than the standard NTSC/PAL formats (for example, CIF, QCIF) can be employed. Because of this, the processor interface is flexible enough to accommodate different row and field lengths. In general, as long as the incoming video has the proper EAV/SAV codes, the PPI can read it in. In other words, a CIF image could be formatted to be "656-compliant," where EAV and SAV values define the range of the image for each line, and the `V` and `F` codes can be used to delimit fields and frames.

Table 7-3. Control Byte Sequences for 8-bit and 10-bit ITU-R 656 Video

	8-bit Data								**10-bit Data**	
	D9 (MSB)	D8	D7	D6	D5	D4	D3	D2	D1	D0
Preamble	1	1	1	1	1	1	1	1	1	1
	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0
Control Byte	1	F	V	H	P3	P2	P1	P0	0	0

ITU-R 656 Input Modes

Figure 7-4 shows a general illustration of data movement in the ITU-R 656 input modes. In the figure, the clock CLK is either provided by the video source or supplied externally by the system.

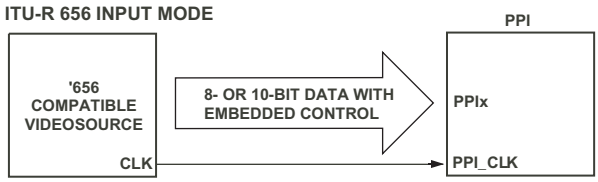


Figure 7-4. ITU-R 656 Input Modes

There are three submodes supported for ITU-R 656 inputs: entire field, active video only, and vertical blanking interval only. Figure 7-5 shows these three submodes.

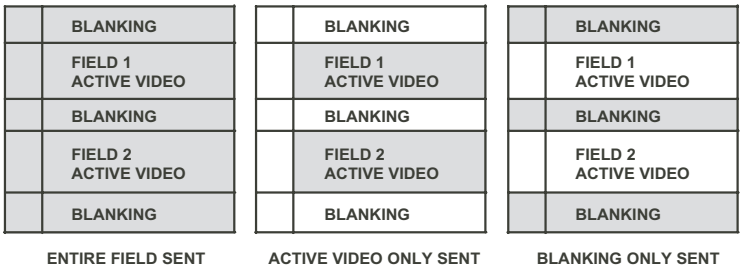


Figure 7-5. ITU-R 656 Input Submodes

Entire Field

In this mode, the entire incoming bitstream is read in through the PPI. This includes active video as well as control byte sequences and ancillary data that may be embedded in horizontal and vertical blanking intervals.

Data transfer starts immediately after synchronization to field 1 occurs, but does not include the first EAV code that contains the F = 0 assignment.

Note the first line transferred in after enabling the PPI will be missing its first 4-byte preamble. However, subsequent lines and frames should have all control codes intact.

One side benefit of this mode is that it enables a "loopback" feature through which a frame or two of data can be read in through the PPI and subsequently output to a compatible video display device. Of course, this requires multiplexing on the PPI pins, but it enables a convenient way to verify that 656 data can be read into and written out from the PPI.

Active Video Only

This mode is used when only the active video portion of a field is of interest, and not any of the blanking intervals. The PPI ignores (does not read in) all data between EAV and SAV, as well as all data present when V = 1. In this mode, the control byte sequences are not stored to memory; they are filtered out by the PPI. After synchronizing to the start of Field 1, the PPI ignores incoming samples until it sees an SAV.

In this mode, the user specifies the number of total (active plus vertical blanking) lines per frame in the `PPI_FRAME` MMR.

Vertical Blanking Interval (VBI) only

In this mode, data transfer is only active while V = 1 is in the control byte sequence. This indicates that the video source is in the midst of the Vertical Blanking Interval (VBI), which is sometimes used for ancillary data transmission. The ITU-R 656 recommendation specifies the format for these ancillary data packets, but the PPI is not equipped to decode the packets themselves. This task must be handled in software. Horizontal blanking data is logged where it coincides with the rows of the VBI.

Control byte sequence information is always logged. The user specifies the number of total lines (active plus vertical blanking) per frame in the PPI_FRAME MMR.

Note the VBI is split into two regions within each field. From the PPI's standpoint, it considers these two separate regions as one contiguous space. However, keep in mind that frame synchronization begins at the start of field 1, which doesn't necessarily correspond to the start of vertical blanking. For instance, in 525/60 systems, the start of field 1 (F = 0) corresponds to line 4 of the VBI.

ITU-R 656 Output Mode

The PPI does not explicitly provide functionality for framing an ITU-R 656 output stream with proper preambles and blanking intervals. However, with the TX mode with 0 frame syncs, this process can be supported manually. Essentially, this mode provides a streaming operation from memory out through the PPI. Data and control codes can be set up in memory prior to sending out the video stream. With the 2D DMA engine, this could be performed in a number of ways. For instance, one line of blanking (H + V) could be stored in a buffer and sent out N times by the DMA controller when appropriate, before proceeding to DMA active video. Alternatively, one entire field (with control codes and blanking) can be set up statically in a buffer while the DMA engine transfers only the active video region into the buffer, on a frame-by-frame basis.

Frame Synchronization in ITU-R 656 Modes

Synchronization in ITU-R 656 modes always occurs at the falling edge of F, the field indicator. This corresponds to the start of field 1. Consequently, up to two fields might be ignored (for example, if field 1 just started before the PPI-to-camera channel was established) before data is received into the PPI.

Because all H and V signalling is embedded in the datastream in ITU-R 656 modes, the PPI_COUNT register is not necessary. However, the PPI_FRAME register is used in order to check for synchronization errors. The user programs this MMR for the number of lines expected in each frame of video, and the PPI keeps track of the number of EAV-to-SAV transitions that occur from the start of a frame until it decodes the end-of-frame condition (transition from F = 1 to F = 0). At this time, the actual number of lines processed is compared against the value in PPI_FRAME. If there is a mismatch, the FT_ERR bit in the PPI_STATUS register is asserted. For instance, if an SAV transition is missed, the current field will only have NUM_ROWS - 1 rows, but resynchronization will reoccur at the start of the next frame.

Upon completing reception of an entire field, the field status bit is toggled in the PPI_STATUS register. This way, an interrupt service routine (ISR) can discern which field was just read in.

General-Purpose PPI Modes

The General-Purpose (GP) PPI modes are intended to suit a wide variety of data capture and transmission applications. Table 7-4 summarizes these modes. If a particular mode shows a given PPI_FSx frame sync not being used, this implies that the pin is available for its alternate, multiplexed functions.

Table 7-4. General-Purpose PPI Modes

GP PPI Mode	PPI_FS1 Direction	PPI_FS2 Direction	PPI_FS3 Direction	Data Direction
RX mode, 0 frame syncs, external trigger	Input	Not used	Not used	Input
RX mode, 0 frame syncs, internal trigger	Not used	Not used	Not used	Input
RX mode, 1 external frame sync	Input	Not used	Not used	Input

Table 7-4. General-Purpose PPI Modes (Cont'd)

GP PPI Mode	PPI_FS1 Direction	PPI_FS2 Direction	PPI_FS3 Direction	Data Direction
RX mode, 2 or 3 external frame syncs	Input	Input	Input (if used)	Input
RX mode, 2 or 3 internal frame syncs	Output	Output	Output (if used)	Input
TX mode, 0 frame syncs	Not used	Not used	Not used	Output
TX mode, 1 external frame sync	Input	Not used	Not used	Output
TX mode, 2 external frame syncs	Input	Input	Not used	Output
TX mode, 1 internal frame sync	Output	Not used	Not used	Output
TX mode, 2 or 3 internal frame syncs	Output	Output	Output (if used)	Output

Figure 7-6 illustrates the general flow of the GP modes. The top of the diagram shows an example of RX mode with 1 external frame sync. After the PPI receives the hardware frame sync pulse (PPI_FS1), it delays for the duration of the PPI_CLK cycles programmed into PPI_DELAY. The DMA controller then transfers in the number of samples specified by PPI_COUNT. Every sample that arrives after this, but before the next PPI_FS1 frame sync arrives, is ignored and not transferred onto the DMA bus.

If the next PPI_FS1 frame sync arrives before the specified PPI_COUNT samples have been read in, the sample counter reinitializes to 0 and starts to count up to PPI_COUNT again. This situation can cause the DMA channel configuration to lose synchronization with the PPI transfer process.

The bottom of Figure 7-6 shows an example of TX mode, 1 internal frame sync. After PPI_FS1 is asserted, there is a latency of 1 PPI_CLK cycle, and then there is a delay for the number of PPI_CLK cycles programmed into

PPI_DELAY. Next, the DMA controller transfers out the number of samples specified by PPI_COUNT. No further DMA takes place until the next PPI_FS1 sync and programmed delay occur.

If the next PPI_FS1 frame sync arrives before the specified PPI_COUNT samples have been transferred out, the sync has priority and starts a new line transfer sequence. This situation can cause the DMA channel configuration to lose synchronization with the PPI transfer process.

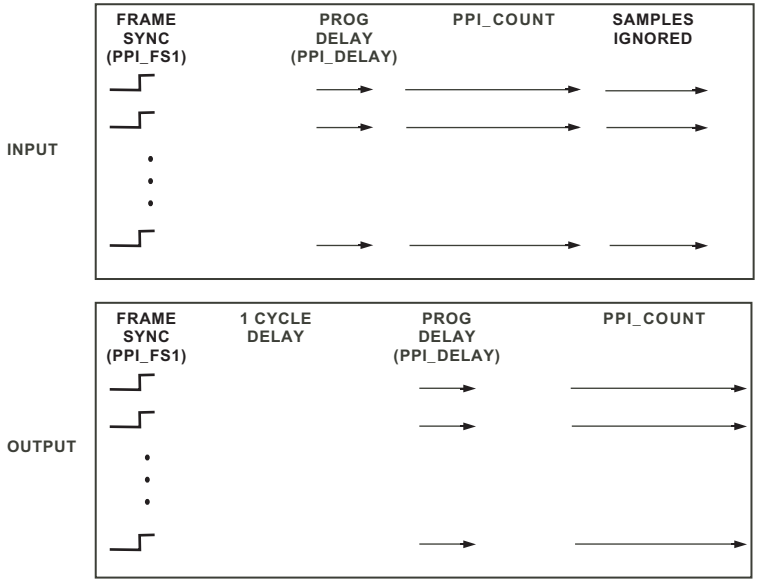


Figure 7-6. General Flow for GP Modes (Assumes Positive Assertion of PPI_FS1)

Data Input (RX) Modes

The PPI supports several modes for data input. These modes differ chiefly by the way the data is framed. Refer to Table 7-2 on page 7-5 for information on how to configure the PPI for each mode.

No Frame Syncs

These modes cover the set of applications where periodic frame syncs are not generated to frame the incoming data. There are two options for starting the data transfer, both configured by the PPI_CONTROL register.

- External trigger: An external source sends a single frame sync (tied to PPI_FS1) at the start of the transaction, when FLD_SEL = 0 and PORT_CFG = b#11.
- Internal trigger: Software initiates the process by setting PORT_EN = 1 with FLD_SEL = 1 and PORT_CFG = b#11.

All subsequent data manipulation is handled via DMA. For example, an arrangement could be set up between alternating 1K memory buffers. When one fills up, DMA continues with the second buffer, at the same time that another DMA operation is clearing the first memory buffer for reuse.

Due to clock domain synchronization in RX modes with no frame syncs, there may be a delay of at least 2 PPI_CLK cycles between when the mode is enabled and when valid data is received. Therefore, detection of the start of valid data should be managed by software.

1, 2, or 3 External Frame Syncs

The frame syncs are level-sensitive signals. The 1-sync mode is intended for Analog-to-Digital Converter (ADC) applications. The top part of Figure 7-7 shows a typical illustration of the system setup for this mode.

The 3-sync mode shown at the bottom of Figure 7-7 supports video applications that use hardware signalling (HSYNC, VSYNC, FIELD) in accordance with the ITU-R 601 recommendation. The mapping for the frame syncs in this mode is PPI_FS1 = HSYNC, PPI_FS2 = VSYNC, PPI_FS3 = FIELD. Please refer to "Frame Synchronization in GP Modes" on page 7-20 for more information about frame syncs in this mode.

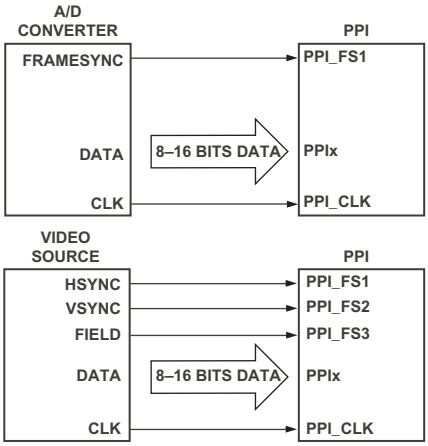


Figure 7-7. RX Mode, External Frame Syncs

A 2-sync mode is supported by not enabling the third frame sync pin in the PORT_MUX and PORTF_FER registers.

2 or 3 Internal Frame Syncs

This mode can be useful for interfacing to video sources that can be slaved to a master processor. In other words, the processor controls when to read from the video source by asserting PPI_FS1 and PPI_FS2, and then reading data into the PPI. The PPI_FS3 frame sync provides an indication of which field is currently being transferred, but since it is an output, it can simply be left floating if not used. Figure 7-8 shows a sample application for this mode.

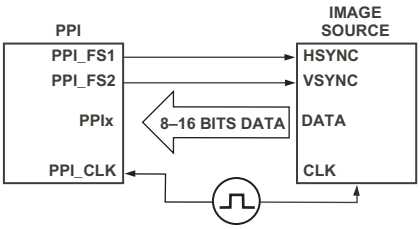


Figure 7-8. RX Mode, Internal Frame Syncs

Data Output (TX) Modes

The PPI supports several modes for data output. These modes differ chiefly by the way the data is framed. Refer to Table 7-2 on page 7-5 for information on how to configure the PPI for each mode.

No Frame Syncs

In this mode, data blocks specified by the DMA controller are sent out through the PPI with no framing. That is, once the DMA channel is configured and enabled, and the PPI is configured and enabled, data transfers will take place immediately, synchronized to PPI_CLK. See Figure 7-9 for an illustration of this mode.

In this mode, there is a delay of up to 16 SCLK cycles (for > 8-bit data) or 32 SCLK cycles (for 8-bit data) between enabling the PPI and transmission of valid data. Furthermore, DMA must be configured to transmit at least 16 samples (for > 8-bit data) or 32 samples (for 8-bit data).

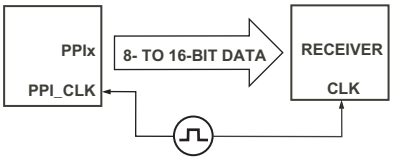


Figure 7-9. TX Mode, 0 Frame Syncs

1 or 2 External Frame Syncs

In these modes, an external receiver can frame data sent from the PPI. Both 1-sync and 2-sync modes are supported. The top diagram in Figure 7-10 shows the 1-sync case, while the bottom diagram illustrates the 2-sync mode.

There is a mandatory delay of 1.5 PPI_CLK cycles, plus the value programmed in PPI_DELAY, between assertion of the external frame sync(s) and the transfer of valid data out through the PPI.

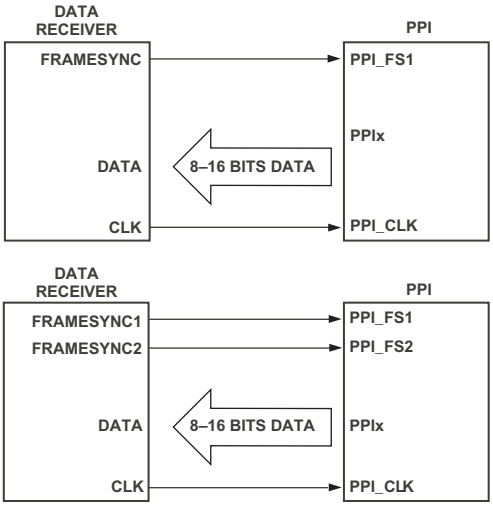


Figure 7-10. TX Mode, 1 or 2 External Frame Syncs

1, 2, or 3 Internal Frame Syncs

The 1-sync mode is intended for interfacing to Digital-to-Analog Converters (DACs) with a single frame sync. The top part of Figure 7-11 shows an example of this type of connection.

The 3-sync mode is useful for connecting to video and graphics displays, as shown in the bottom part of Figure 7-11. A 2-sync mode is implicitly supported by leaving PPI_FS3 unconnected in this case.

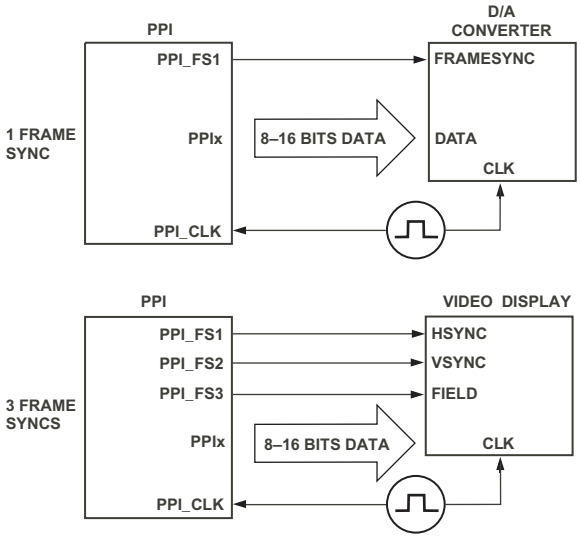


Figure 7-11. PPI GP Output

Frame Synchronization in GP Modes

Frame synchronization in GP modes operates differently in modes with internal frame syncs than in modes with external frame syncs.

Modes With Internal Frame Syncs

In modes with internal frame syncs, PPI_FS1 and PPI_FS2 link directly to the Pulsewidth Modulation (PWM) circuits of timer 0 and timer 1, respectively. This allows for arbitrary pulse widths and periods to be programmed for these signals using the existing TIMERx registers. This capability accommodates a wide range of timing needs. Note these PWM circuits are clocked by PPI_CLK, not by SCLK (as during conventional timer PWM operation). If PPI_FS2 is not used in the configured PPI mode, timer 1 operates as it normally would, unrestricted in functionality. The state of PPI_FS3 depends completely on the state of PPI_FS1 and/or PPI_FS2, so PPI_FS3 has no inherent programmability.

To program PPI_FS1 and/or PPI_FS2 for operation in an internal frame sync mode:

1. Configure and enable DMA for the PPI. See "DMA Operation" on page 7-23.
2. Configure the width and period for each frame sync signal via TIMER0_WIDTH and TIMER0_PERIOD (for PPI_FS1), or TIMER1_WIDTH and TIMER1_PERIOD (for PPI_FS2).
3. Set up TIMER0_CONFIG for PWM_OUT mode (for PPI_FS1). If used, configure TIMER1_CONFIG for PWM_OUT mode (for PPI_FS2). This includes setting CLK_SEL = 1 and TIN_SEL = 1 for each timer.
4. Write to PPI_CONTROL to configure and enable the PPI.
5. Write to TIMER_ENABLE to enable timer 0 and/or timer 1.

It is important to guarantee proper frame sync polarity between the PPI and timer peripherals. To do this, make sure that if PPI_CONTROL[15:14] = b#10 or b#11, the PULSE_HI bit is cleared in TIMER0_CONFIG and TIMER1_CONFIG. Likewise, if PPI_CONTROL[15:14] = b#00 or b#01, the PULSE_HI bit should be set in TIMER0_CONFIG and TIMER1_CONFIG.

To switch to another PPI mode not involving internal frame syncs:

1. Disable the PPI (using PPI_CONTROL).
2. Disable the timers (using TIMER_DISABLE).

Modes With External Frame Syncs

In RX modes with external frame syncs, the PPI_FS1 and PPI_FS2 pins become edge-sensitive inputs. In such a mode, timers 1 and 2 can be used for a purpose not involving the TMR0 and TMR1 pins. However, timer access to a TMRx pin is disabled when the PPI is using that pin for a PPI_FSx frame sync input function. For modes that do not require PPI_FS2, timer

1 is not restricted in functionality and can be operated as if the PPI were not being used (that is, the TMR1 pin becomes available for timer use as well). For more information on configuring and using the timers, please refer to Chapter 15, "General-Purpose Timers".

In RX mode with 3 external frame syncs, the start of frame detection occurs where a PPI_FS2 assertion is followed by an assertion of PPI_FS1 while PPI_FS3 is low. This happens at the start of field 1. Note that PPI_FS3 only needs to be low when PPI_FS1 is asserted, not when PPI_FS2 asserts. Also, PPI_FS3 is only used to synchronize to the start of the very first frame after the PPI is enabled. It is subsequently ignored.

In TX modes with external frame syncs, the PPI_FS1 and PPI_FS2 pins are treated as edge-sensitive inputs. In this mode, it is not necessary to configure the timer(s) associated with the frame sync(s) as input(s), or to enable them via the TIMER_ENABLE register. Additionally, the actual timers themselves are available for use, even though the timer pin(s) are taken over by the PPI. In this case, there is no requirement that the timebase (configured by TIN_SEL in TIMERx_CONFIG) be PPI_CLK.

However, if using a timer whose pin is connected to an external frame sync, be sure to disable the pin via the OUT_DIS bit in TIMERx_CONFIG. Then the timer itself can be configured and enabled for non-PPI use without affecting PPI operation in this mode. For more information, see Chapter 15, "General-Purpose Timers".

Programming Model

The following sections describe the PPI programming model.

DMA Operation

The PPI must be used with the processor's DMA engine. This section discusses how the two interact. For additional information about the DMA engine, including explanations of DMA registers and DMA operations, please refer to Chapter 5, "Direct Memory Access".

The PPI DMA channel can be configured for either transmit or receive operation, and it has a maximum throughput of `(PPI_CLK) x (16 bits/transfer)`. In modes where data lengths are greater than 8 bits, only one element can be clocked in per `PPI_CLK` cycle, and this results in reduced bandwidth (since no packing is possible). The highest throughput is achieved with 8-bit data and `PACK_EN = 1` (packing mode enabled). Note for 16-bit packing mode, there must be an even number of data elements.

Configuring the PPI's DMA channel is a necessary step toward using the PPI interface. It is the DMA engine that generates interrupts upon completion of a row, frame, or partial-frame transfer. It is also the DMA engine that coordinates the origination or destination point for the data that is transferred through the PPI.

The processor's 2D DMA capability allows the processor to be interrupted at the end of a line or after a frame of video has been transferred, as well as if a DMA error occurs. In fact, the specification of the `DMAx_XCOUNT` and `DMAx_YCOUNT` MMRs allows for flexible data interrupt points. For example,

assume the DMA registers XMODIFY = YMODIFY = 1. Then, if a data frame contains 320 x 240 bytes (240 rows of 320 bytes each), these conditions hold:

- Setting XCOUNT = 320, YCOUNT = 240, and DI_SEL = 1 (the DI_SEL bit is located in DMAx_CONFIG) interrupts on every row transferred, for the entire frame.
- Setting XCOUNT = 320, YCOUNT = 240, and DI_SEL = 0 interrupts only on the completion of the frame (when 240 rows of 320 bytes have been transferred).
- Setting XCOUNT = 38,400 (320 x 120), YCOUNT = 2, and DI_SEL = 1 causes an interrupt when half of the frame has been transferred, and again when the whole frame has been transferred.

Following is the general procedure for setting up DMA operation with the PPI. For details regarding configuration of DMA, please refer to Chapter 5, "Direct Memory Access".

1. Configure DMA registers as appropriate for desired DMA operating mode.
2. Enable the DMA channel for operation.
3. Configure appropriate PPI registers.
4. Enable the PPI by writing a 1 to bit 0 in PPI_CONTROL.
5. If internally generated frame syncs are used, write to the TIMER_ENABLE register to enable the timers linked to the PPI frame syncs.

Figure 7-12 shows a flow diagram detailing the steps on how to configure the PPI for the various modes of operation.

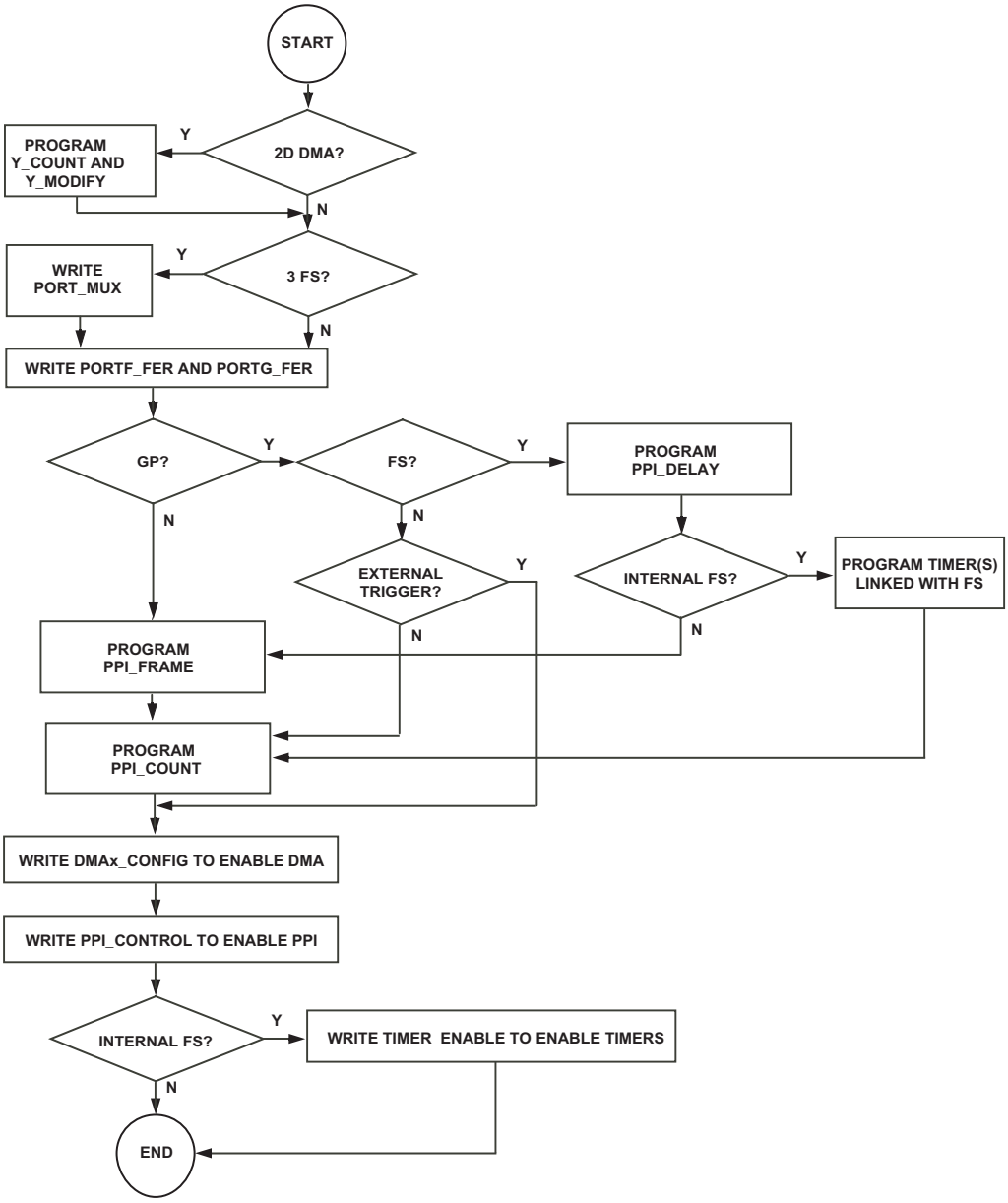


Figure 7-12. PPI Flow Diagram

PPI Registers

The PPI has five memory-mapped registers (MMRs) that regulate its operation. These registers are the PPI control register (PPI_CONTROL), the PPI status register (PPI_STATUS), the delay count register (PPI_DELAY), the transfer count register (PPI_COUNT), and the lines per frame register (PPI_FRAME).

Descriptions and bit diagrams for each of these MMRs are provided in the following sections.

PPI_CONTROL Register

The PPI_CONTROL register configures the PPI for operating mode, control signal polarities, and data width of the port. See Figure 7-13 for a bit diagram of this MMR.

The POLC and POLS bits allow for selective signal inversion of the PPI_CLK and PPI_FS1/PPI_FS2 signals, respectively. This provides a mechanism to connect to data sources and receivers with a wide array of control signal polarities. Often, the remote data source/receiver also offers configurable signal polarities, so the POLC and POLS bits simply add increased flexibility.

The DLEN[2:0] field is programmed to specify the width of the PPI port in any mode. Note any width from 8 to 16 bits is supported, with the exception of a 9-bit port width. Any pins unused by the PPI as a result of the DLEN setting are free for use in their other functions, as detailed in Chapter 14, “General-Purpose Ports”.

In ITU-R 656 modes, the DLEN field should not be configured for anything greater than a 10-bit port width. If it is, the PPI will reserve extra pins, making them unusable by other peripherals.

The SKIP_EN bit, when set, enables the selective skipping of data elements being read in through the PPI. By ignoring data elements, the PPI is able to conserve DMA bandwidth.

PPI Control Register (PPI_CONTROL)

0xFFC0 1000

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

Reset = 0x0000

POLS
0 - PPI_FS1 and PPI_FS2 are treated as rising edge asserted
1 - PPI_FS1 and PPI_FS2 are treated as falling edge asserted

POLC
0 - PPI samples data on rising edge and drives data on falling edge of PPI_CLK
1 - PPI samples data on falling edge and drives data on rising edge of PPI_CLK

DLEN[2:0] (Data Length)
000 - 8 bits
001 - 10 bits
010 - 11 bits
011 - 12 bits
100 - 13 bits
101 - 14 bits
110 - 15 bits
111 - 16 bits

SKIP_EO (Skip Even Odd)
In ITU-R 656 and GP Input modes:
0 - Skip odd-numbered elements
1 - Skip even-numbered elements

SKIP_EN (Skip Enable)
In ITU-R 656 and GP Input modes:
0 - Skipping disabled
1 - Skipping enabled

PACK_EN (Packing Mode Enable)
0 - Disabled
1 - Output mode, unpacking enabled; Input mode, packing enabled

FLD_SEL (Active Field Select)
In ITU-R 656 modes, when XFR_TYPE = 00:
0 - Field 1
1 - Fields 1 and 2
In RX mode with external frame sync, when PORT_CFG = 11:
0 - External trigger
1 - Internal trigger

PORT_EN (Enable)
0 - PPI disabled
1 - PPI enabled

PORT_DIR (Direction)
0 - PPI in Receive mode (input)
1 - PPI in Transmit mode (output)

XFR_TYPE[1:0] (Transfer Type)
In Input mode:
00 - ITU-R 656, Active Field Only
01 - ITU-R 656, Entire Field
10 - ITU-R 656, Vertical Blanking Only
11 - Non-ITU-R 656 mode
In Output mode:
00, 01, 10 - Sync-less Output mode
11 - Output mode with 1, 2, or 3 frame syncs

PORT_CFG[1:0] (Port Configuration)
In non-ITU-R 656 Input modes (PORT_DIR = 0, XFR_TYPE = 11):
00 - 1 external frame sync
01 - 2 or 3 internal frame syncs
10 - 2 or 3 external frame syncs
11 - 0 frame syncs, triggered
In Output modes with frame syncs (PORT_DIR = 1, XFR_TYPE = 11):
00 - 1 frame sync
01 - 2 or 3 frame syncs
10 - Reserved
11 - Sync PPI_FS3 to assertion of PPI_FS2 rather than of PPI_FS1.

Figure 7-13. PPI Control Register

When the SKIP_EN bit is set, the SKIP_EO bit allows the PPI to ignore either the odd or the even elements in an input datastream. This is useful, for instance, when reading in a color video signal in YCbCr format (Cb, Y, Cr, Y, Cb, Y, Cr, Y...). Skipping every other element allows the PPI to only read in the luma (Y) or chroma (Cr or Cb) values. This could also be useful when synchronizing two processors to the same incoming video stream. One processor could handle luma processing and the other (whose SKIP_EO bit is set differently from the first processor's) could handle chroma processing. This skipping feature is valid in ITU-R 656 modes and RX modes with external frame syncs.

The PACK_EN bit only has meaning when the PPI port width (selected by DLEN[2:0]) is 8 bits. Every PPI_CLK-initiated event on the DMA bus (that is, an input or output operation) handles 16-bit entities. In other words, an input port width of 10 bits still results in a 16-bit input word for every PPI_CLK; the upper 6 bits are 0s. Likewise, a port width of 8 bits also results in a 16-bit input word, with the upper 8 bits all 0s. In the case of 8-bit data, it is usually more efficient to pack this information so that there are two bytes of data for every 16-bit word. This is the function of the PACK_EN bit. When set, it enables packing for all RX modes.

Consider this data transported into the PPI via DMA:

0xCE, 0xFA, 0xFE, 0xCA....

- With PACK_EN set:

  This is read into the PPI, configured for an 8-bit port width:

  ```
  0xCE, 0xFA, 0xFE, 0xCA...
  ```

  This is transferred onto the DMA bus:

  ```
  0xFACE, 0xCAFE, ...
  ```

- With PACK_EN cleared:

  This is read into the PPI:

  0xCE, 0xFA, 0xFE, 0xCA, ...

  This is transferred onto the DMA bus:

  0x00CE, 0x00FA, 0x00FE, 0x00CA, ...

For TX modes, setting PACK_EN enables unpacking of bytes. Consider this data in memory, to be transported out through the PPI via DMA:

0xFACE CAFE.... (0xFA and 0xCA are the two Most Significant Bits (MSBs) of their respective 16-bit words)

- With PACK_EN set:

  This is DMAed to the PPI:

  0xFACE, 0xCAFE, ...

  This is transferred out through the PPI, configured for an 8-bit port width (note LSBs are transferred first):

  0xCE, 0xFA, 0xFE, 0xCA, ...

- With PACK_EN cleared:

  This is DMAed to the PPI:

  0xFACE, 0xCAFE, ...

  This is transferred out through the PPI, configured for an 8-bit port width:

  0xCE, 0xFE, ...

The FLD_SEL bit is used primarily in the active field only ITU-R 656 mode. The FLD_SEL bit determines whether to transfer in only field 1 of each video frame, or both fields 1 and 2. Thus, it allows a savings in DMA bandwidth by transferring only every other field of active video.

The PORT_CFG[1:0] field is used to configure the operating mode of the PPI. It operates in conjunction with the PORT_DIR bit, which sets the direction of data transfer for the port. The XFR_TYPE[1:0] field is also used to configure operating mode and is discussed below. See Table 7-2 on page 7-5 for the possible operating modes for the PPI.

The XFR_TYPE[1:0] field configures the PPI for various modes of operation. Refer to Table 7-2 on page 7-5 to see how XFR_TYPE[1:0] interacts with other bits in PPI_CONTROL to determine the PPI operating mode.

The PORT_EN bit, when set, enables the PPI for operation.

Note that, when configured as an input port, the PPI does not start data transfer after being enabled until the appropriate synchronization signals are received. If configured as an output port, transfer (including the appropriate synchronization signals) begins as soon as the frame syncs (timer units) are enabled, so all frame syncs must be configured before this happens. Refer to the section "Frame Synchronization in GP Modes" on page 7-20 for more information.

PPI_STATUS Register

The PPI_STATUS register, shown in Figure 7-14, contains bits that provide information about the current operating state of the PPI.

The ERR_DET bit is a sticky bit that denotes whether or not an error was detected in the ITU-R 656 control word preamble. The bit is valid only in ITU-R 656 modes. If ERR_DET = 1, an error was detected in the preamble. If ERR_DET = 0, no error was detected in the preamble.

The ERR_NCOR bit is sticky and is relevant only in ITU-R 656 modes. If ERR_NCOR = 0 and ERR_DET = 1, all preamble errors that have occurred have been corrected. If ERR_NCOR = 1, an error in the preamble was detected but not corrected. This situation generates a PPI error interrupt, unless this condition is masked off in the SIC_IMASK register.

The FT_ERR bit is sticky and indicates, when set, that a frame track error has occurred. It is valid for RX modes only. In this condition, the programmed number of lines per frame in PPI_FRAME does not match up with the "frame start detect" condition (see the information note on page 7-36). A frame track error generates a PPI error interrupt, unless this condition is masked off in the SIC_IMASK register.

The FLD bit is set or cleared at the same time as the change in state of F (in ITU-R 656 modes) or PPI_FS3 (in other RX modes). It is valid for input modes only. The state of FLD reflects the current state of the F or PPI_FS3 signals. In other words, the FLD bit always reflects the current video field being processed by the PPI.

The OVR bit is sticky and indicates, when set, that the PPI FIFO has overflowed and can accept no more data. A FIFO overflow error generates a PPI error interrupt, unless this condition is masked off in the SIC_IMASK register.

The PPI FIFO is 16 bits wide and has 16 entries.

The UNDR bit is sticky and indicates, when set, that the PPI FIFO has underrun and is data-starved. A FIFO underrun error generates a PPI error interrupt, unless this condition is masked off in the SIC_IMASK register.

The LT_ERR_OVR and LT_ERR_UNDR bits are sticky and indicate, when set, that a line track error has occurred. These bits are valid for RX modes with recurring frame syncs only. If one of these bits is set, the programmed number of samples in PPI_COUNT did not match up with the actual number of samples counted between assertions of PPI_FS1 (for general-purpose

modes) or “Start of Active Video (SAV)” codes (for ITU-R 656 modes). If the PPI error interrupt is enabled in the SIC_IMASK register, an interrupt request is generated when one of these bits is set.

The LT_ERR_OVR flag signifies that a horizontal tracking overflow has occurred, where the value in PPI_COUNT was reached before a new SAV code was received. This flag does not apply for non-ITU-R 656 modes; in this case, once the value in PPI_COUNT is reached, the PPI simply stops counting until receiving the next PPI_FS1 frame sync.

The LT_ERR_UNDR flag signifies that a horizontal tracking underflow has occurred, where a new SAV code or PPI_FS1 assertion occurred before the value in PPI_COUNT was reached.

PPI Status Register (PPI_STATUS)

0xFFC0 1004

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

Reset = 0x0000

ERR_NCOR (Error Not Corrected) - W1C

Used only in ITU-R 656 modes
0 - No uncorrected preamble error has occurred
1 - Preamble error detected but not corrected

ERR_DET (Error Detected) - W1C

Used only in ITU-R 656 modes
0 - No preamble error detected
1 - Preamble error detected

UNDR (FIFO Underrun) - W1C

0 - No interrupt
1 - FIFO Underrun Error interrupt occurred

OVR (FIFO Overflow) - W1C

0 - No interrupt
1 - FIFO Overflow Error interrupt occurred

LT_ERR_OVR (Horizontal Tracking Overflow Error) - W1C

Used only in ITU-R 656 modes
0 - No horizontal tracking overflow error
1 - PPI_COUNT expired before receiving SAV code

LT_ERR_UNDR (Horizontal Tracking Underflow Error) - W1C

0 - No horizontal tracking underflow error
1 - PPI_FS1 (or SAV code) received before PPI_COUNT expired for that line

FLD (Field Indicator)

0 - Field 1
1 - Field 2

FT_ERR (Frame Track Error) - W1C

0 - No interrupt
1 - Frame Track Error interrupt occurred

Figure 7-14. PPI Status Register

PPI_DELAY Register

The PPI_DELAY register, shown in Figure 7-15, can be used in all configurations except ITU-R 656 modes and GP modes with 0 frame syncs. It contains a count of how many PPI_CLK cycles to delay after assertion of PPI_FS1 before starting to read in or write out data.

Note in TX modes using at least one frame sync, there is a one-cycle delay beyond what is specified in the PPI_DELAY register.

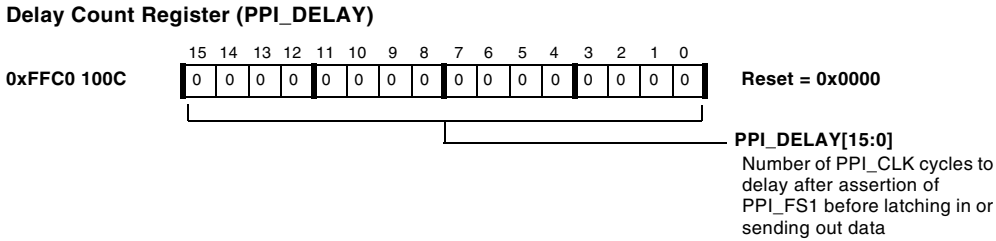


Figure 7-15. Delay Count Register

PPI_COUNT Register

The PPI_COUNT register, shown in Figure 7-15, is used in all modes except "RX mode with 0 frame syncs, external trigger" and "TX mode with 0 frame syncs." For RX modes, this register holds the number of samples to read into the PPI per line, minus one. For TX modes, it holds the number of samples to write out through the PPI per line, minus one. The register itself does not actually decrement with each transfer. Thus, at the beginning of a new line of data, there is no need to rewrite the value of this register. For example, to receive or transmit 100 samples through the PPI, set PPI_COUNT to 99.

Take care to ensure that the number of samples programmed into PPI_COUNT is in keeping with the number of samples expected during the "horizontal" interval specified by PPI_FS1.

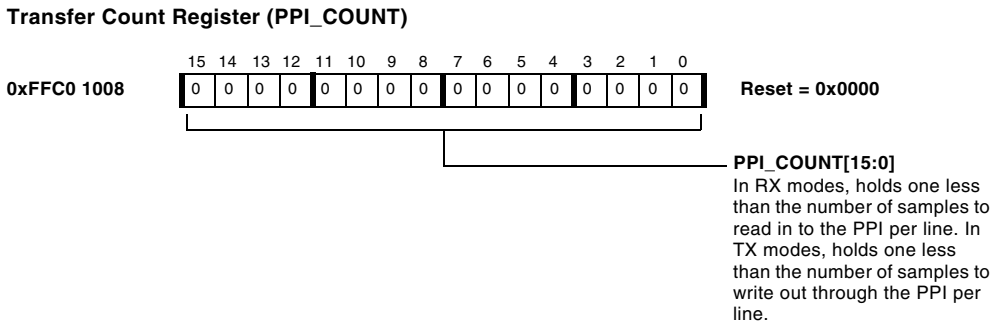


Figure 7-16. Transfer Count Register

PPI_FRAME Register

The PPI_FRAME register, shown in Figure 7-17, is used in all TX and RX modes with external frame syncs. For ITU-R 656 modes, this register holds the number of lines expected per frame of data, where a frame is defined as field 1 and field 2 combined, designated by the F indicator in the ITU-R stream. Here, a line is defined as a complete ITU-R 656 SAV-EAV cycle.

For non-ITU-R 656 modes with external frame syncs, a frame is defined as the data bounded between PPI_FS2 assertions, regardless of the state of PPI_FS3. A line is defined as a complete PPI_FS1 cycle. In these modes, PPI_FS3 is used only to determine the original “frame start” each time the PPI is enabled. It is ignored on every subsequent field and frame, and its state (high or low) is not important except during the original frame start.

If the start of a new frame (or field, for ITU-R 656 mode) is detected before the number of lines specified by PPI_FRAME have been transferred, a frame track error results, and the FT_ERR bit in PPI_STATUS is set. However, the PPI still automatically reinitializes to count to the value programmed in PPI_FRAME, and data transfer continues.

In ITU-R 656 modes, a frame start detect happens on the falling edge of F, the field indicator. This occurs at the start of field 1.

In RX mode with 3 external frame syncs, a frame start detect refers to a condition where a PPI_FS2 assertion is followed by an assertion of PPI_FS1 while PPI_FS3 is low. This occurs at the start of field 1. Note that PPI_FS3 only needs to be low when PPI_FS1 is asserted, not when PPI_FS2 asserts. Also, PPI_FS3 is only used to synchronize to the start of the very first frame after the PPI is enabled. It is subsequently ignored.

When using RX mode with 3 external frame syncs, and only 2 syncs are needed, configure the PPI for three-frame-sync operation and provide an external pull-down to GND for the PPI_FS3 pin.

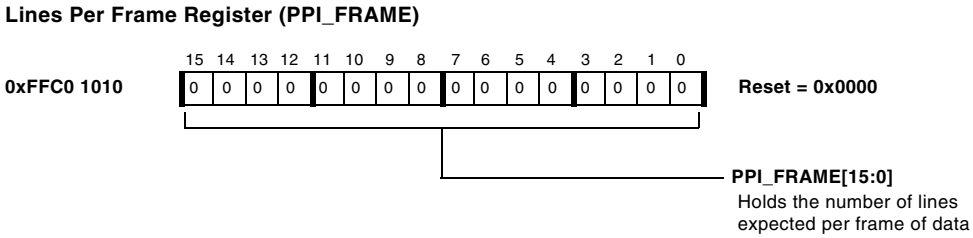


Figure 7-17. Lines Per Frame Register

Programming Examples

As shown in the data transfer scenario in Figure 7-18 on page 7-40, the PPI can be configured to receive data from a video source in several RX modes. The following programming examples (Listing 7-1 through Listing 7-5) describe the ITU-R 656 entire field input mode.

Listing 7-1. Configure DMA Registers

```
config_dma:

  /* DMA0_START_ADDR */
  R0.L = rx_buffer;
  R0.H = rx_buffer;
  P0.L = lo(DMA0_START_ADDR);
  P0.H = hi(DMA0_START_ADDR);
  [P0] = R0;

  /* DMA0_CONFIG */
  R0.L = DI_EN | WNR;
  P0.L = lo(DMA0_CONFIG);
  P0.H = hi(DMA0_CONFIG);
  W[P0] = R0.L;

  /* DMA0_X_COUNT */
  R0.L = 256;
  P0.L = lo(DMA0_X_COUNT);
  P0.H = hi(DMA0_X_COUNT);
  W[P0] = R0.L;

  /* DMA0_X_MODIFY */
  R0.L = 0x0001;
  P0.L = lo(DMA0_X_MODIFY);
  P0.H = hi(DMA0_X_MODIFY);
  W[P0] = R0.L;
  ssync;
config_dma.END:  RTS;
```

Listing 7-2. Configure PPI Registers

```
config_ppi:

 /* PPI_CONTROL */
  P0.L = lo(PPI_CONTROL);
  P0.H = hi(PPI_CONTROL);
  R0.L = 0x0004;
  W[P0] = R0.L;
  ssync;

config_ppi.END:  RTS;
```

Listing 7-3. Enable DMA

```
  /* DMA0_CONFIG */
  P0.L = lo(DMA0_CONFIG);
  P0.H = hi(DMA0_CONFIG);
  R0.L = W[P0];
  bitset(R0,0);
  W[P0] = R0.L;
  ssync;
```

Listing 7-4. Enable PPI

```
  /* PPI_CONTROL */
  P0.L = lo(PPI_CONTROL);
  P0.H = hi(PPI_CONTROL);
  R0.L = W[P0];
  bitset(R0,0);
  W[P0] = R0.L;
  ssync;
```

Listing 7-5. Clear DMA Completion Interrupt

```
/* DMA0_IRQ_STATUS */
P2.L = lo(DMA0_IRQ_STATUS);
P2.H = hi(DMA0_IRQ_STATUS);
R2.L = W[P2];
BITSET(R2,0);
W[P2] = R2.L;
ssync;
```

Data Transfer Scenarios

Figure 7-18 shows two possible ways to use the PPI to transfer in video. These diagrams are very generalized, and bandwidth calculations must be made only after factoring in the exact PPI mode and settings (for example, transfer field 1 only, transfer odd and even elements).

The top part of the diagram shows a situation appropriate for, as an example, JPEG compression. The first N rows of video are DMAed into L1 memory via the PPI. Once in L1, the compression algorithm operates on the data and sends the compressed result out from the processor via the SPORT. Note that no SDRAM access was necessary in this approach.

The bottom part of the diagram takes into account a more formidable compression algorithm, such as MPEG-2 or MPEG-4. Here, the raw video is transferred directly into SDRAM. Independently, a memory

DMA channel transfers data blocks between SDRAM and L1 memory for intermediate processing stages. Finally, the compressed video exits the processor via the SPORT.

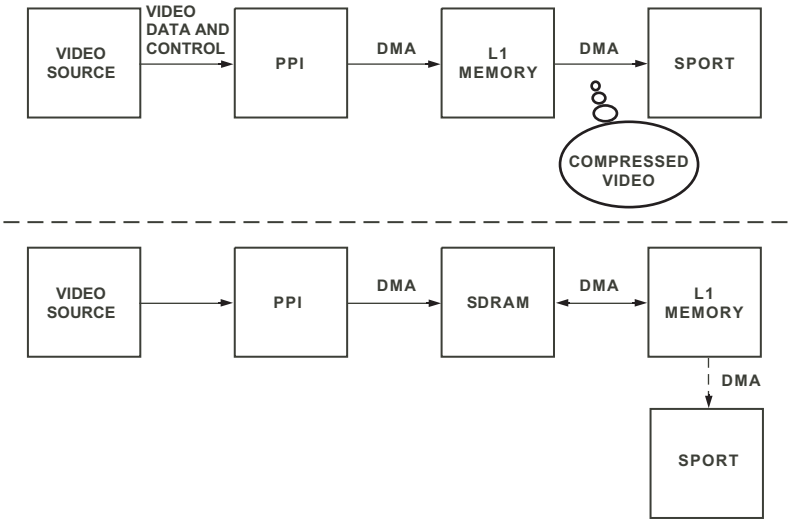


Figure 7-18. PPI Possible Data Transfer Scenarios

8 ETHERNET MAC

This chapter describes the Ethernet Media Access Controller (MAC) peripheral for the ADSP-BF536 and ADSP-BF537 processors. Following an overview and list of key features is a description of operation and functional modes of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

Overview

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.

Features

The Ethernet MAC includes these features:

- Independent DMA-driven RX and TX channels
- MII/RMII interface
- 10Mbit/s and 100Mbit/s operation (full or half duplex)
- VLAN support (full or half duplex)
- Automatic network monitoring statistics

- Flexible address filtering
- Flexible event detection for interrupt handling
- Validation of IP and TCP (payload) checksum
- Remote-wakeup Ethernet frames
- Network-aware system power management

The MAC is fully compliant to IEEE Std. 802.3-2002.

Interface Overview

Figure 8-1 illustrates the overall architecture of the Ethernet controller. The central MAC block implements the Carrier Sense Multiple Access with Collision Detection (CSMA/CD) protocol for both half-duplex and full-duplex modes.

The System Interface (SIF) block contains FIFOs for RX and TX data and handles the synchronization of data between the MAC RX and TX data streams and the Blackfin DMA controller.

The System Interface Registers (SIF_REG) block is an interface from the Blackfin Peripheral Access Bus (PAB) to the internal registers in the MAC. This block also generates the Ethernet event interrupt, and supports the $\overline{\text{PHYINT}}$ pin by which the PHY can notify the Blackfin processor when the PHY detects changes to the link status, such as auto-negotiation or duplex mode change.

The MAC Management Counters (MMC) block is an extended set of registers that collect various statistics compliant with IEEE 802.3 definitions regarding the operation of the interface. They are updated for each new transmitted or received frame.

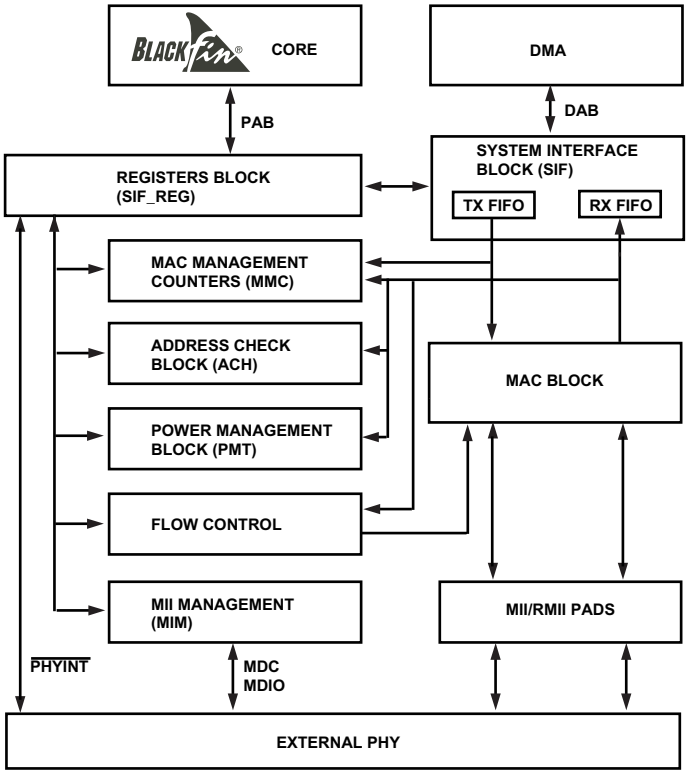


Figure 8-1. Ethernet MAC Block Diagram

The Power Management (PMT) block adds support for wakeup frames and magic packet technology that allows waking up the processor from low power operating modes. Further details regarding these low-power operating modes and voltage regulator wakeup functionality can be found in the "Operating Modes and States" chapter of the *ADSP-BF53x/BF56x Blackfin Processor Programming Reference*.

The Address Check (ACH) block checks the destination address field of all incoming packets. Based on the type of address filtering selected, this indicates the result of the address checking to the MAC block.

The MII Management (MIM) block handles all transactions to the control and status registers on the external PHY.

External Interface

Clocking

The Ethernet MAC is clocked internally from SCLK on the processor. A buffered version of CLKIN may be used to drive the external PHY via the CLKBUF pin. See Figure 8-2.

The CLKBUF signal is not generated by a PLL and supports jitter and stability functions comparable to XTAL. The CLKBUF pin is enabled by the PHYCLKOE bit in the VR_CTL register. See Chapter 20, "Dynamic Power Management" for more information.

A 25 MHz clock (whether driven with the CLKBUF pin or an external crystal) should be used with an MII PHY. A 50 MHz clock source is required to drive an RMII PHY.

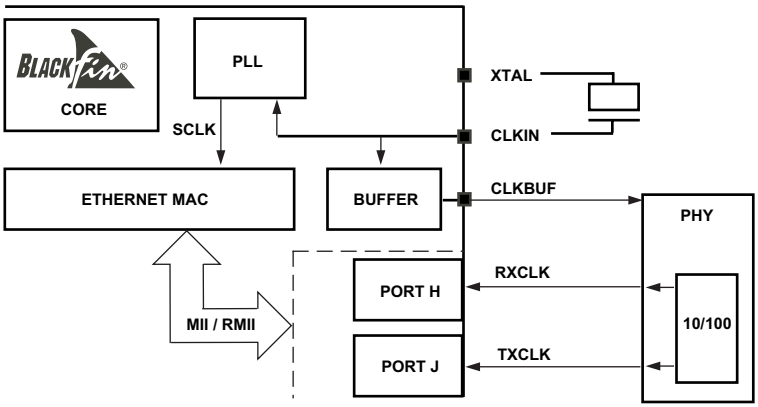


Figure 8-2. Clock Function Diagram

Pins

MII and RMII peripherals are multiplexed into the general-purpose ports, with port H and port J supporting this functionality. To use MII and RMII operations, set the PORTH_FER register accordingly. See Chapter 14, "General-Purpose Ports" for more information. The two MII and RMII signals (MDIO/MDC) in port J are not multiplexed, and are directly connected to pins PJ0 and PJ1.

Table 8-1 shows the pins for the MAC.

Table 8-1. Ethernet MAC Pins

Pin Name	MII Multiplexed Name	MII Input/ Output	RMII Multiplexed Name	RMII Input/ Output	Description
PH0	MII TXD0	O	RMII TXD0	O	Ethernet MII or RMII transmit D0
PH1	MII TXD1	O	RMII TXD1	O	Ethernet MII or RMII transmit D1
PH2	MII TXD2	O			Ethernet MII transmit D2
PH3	MII TXD3	O			Ethernet MII transmit D3
PH4	MII TXEN	O	RMII TXEN	O	Ethernet MII or RMII transmit enable
PH5	MII TXCLK	I	RMII REFCLK	I	Ethernet MII transmit clock/RMII reference clock
PH6	MII $\overline{\text{PHYINT}}$	I	RMII $\overline{\text{MDINT}}$	I	Ethernet MII PHY interrupt/RMII management data interrupt
PH7	MII COL	I			Ethernet collision
PH8	MII RXD0	I	RMII RXD0	I	Ethernet MII or RMII receive D0
PH9	MII RXD1	I	RMII RXD1	I	Ethernet MII or RMII receive D1

Table 8-1. Ethernet MAC Pins (Cont'd)

Pin Name	MII Multiplexed Name	MII Input/ Output	RMII Multiplexed Name	RMII Input/ Output	Description
PH10	MII RXD2	I			Ethernet MII receive D2
PH11	MII RXD3	I			Ethernet MII receive D3
PH12	MII RXDV	I			Ethernet MII receive data valid
PH13	MII RXCLK	I			Ethernet MII receive clock
PH14	MII RXER	I	RMII RXER	I	Ethernet MII or RMII receive error
PH15	MII CRS	I	RMII CRS_DV	I	Ethernet MII carrier sense/RMII carrier sense and receive data valid
PJ0	MDC	O	MDC	O	Ethernet management channel clock
PJ1	MDIO	I/O	MDIO	I/O	Ethernet management channel serial data

IEEE802.3-2002, section two, clause 22.2.1.6, characterizes the MII TX_ER pin as an option for certain applications (for example, repeater applications). Therefore, the TX_ER pin is not present in this design.

Internal Interface

Communication between the MAC and the Blackfin processor peripheral subsystem takes place over the Peripheral Access Bus (PAB) and the DMA Access Bus (DAB). The PAB is used by the Blackfin processor core to configure and monitor the peripheral's control and status registers. All data transfers to and from the peripheral are handled by the Blackfin DMA controller and take place via the DAB.

Power Management

The ADSP-BF536/ADSP-BF537 provides power management states which allow programming the MAC to wake the processor upon reception of specific Ethernet frames and/or upon selected events detected by the PHY. The MAC itself requires no additional power management intervention; its internal clocks power down automatically when not required. The MAC clocks run in any of these conditions (provided the ADSP-BF536/ADSP-BF537 is in the sleep, active, or full on state):

1. Either the receiver or transmitter is enabled (`RE` or `TE = 1`)
2. During an MII Management transfer (on MDC/MDIO)
3. During a core access to an MAC control/status register
4. While PHY interrupts are enabled in the MAC (`PHYIE` in the `EMAC_SYSCTL` register is set)

Description of Operation

The following sections describe the operation of the MAC.

Protocol

The Ethernet MAC complies with IEEE Std. 802.3-2002. The MII management interface is described below.

MII Management Interface

The IEEE 802.3 MII management interface, also known as the MDIO station management interface, allows the Blackfin processor to monitor and control one or more external Ethernet physical-layer transceivers (PHYs). The MII management interface physically consists of a 2-wire

serial connection composed of the MDC (management data clock) output signal and the MDIO (management data input/output) bidirectional data signal. See Figure 8-3 and Figure 8-4.

The MII management logical interface specifies:

- A set of 16-bit device control/status registers within PHYs, including both required registers with standardized bit definitions as well as optional vendor-specified registers
- A 5-bit device addressing scheme which allows the MAC to select one of up to 32 externally-connected PHY devices
- A 5-bit register addressing scheme for selecting the target register within the addressed device
- A transfer frame protocol for 16-bit read and write accesses to PHY registers via the MDC and MDIO signals under control of the MAC (PHY devices may not directly initiate MDIO transfers.)

Standard PHY control and status registers provide device capability status bits (for example, auto-negotiation, duplex modes, 10/100 speeds and protocols), device status bits (for example, auto-negotiation complete, link status, remote fault), and device control bits (for example, reset, speed selection, loopback, and auto-negotiation start).

The transfer frame protocol defines a MDC clock at a nominal period of 400ns, and an MDIO frame up to 64 bits in length. The MDIO frame consists of an optional 32-bit preamble driven by the MAC, 14 control bits driven by the MAC including the opcode and addresses, a 2-bit turn-around sequence, and a 16-bit data transfer driven either by the MAC or the PHY. Note that various PHYs support optional features such as

reduced preamble or increased clock rate. The features supported by the PHY may be determined at powerup by a MDIO read access (at default rates) of device capabilities in PHY status registers.

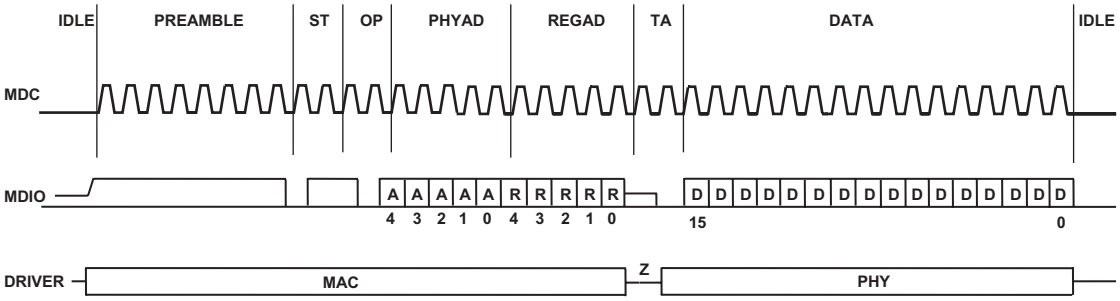


Figure 8-3. Station Management Read

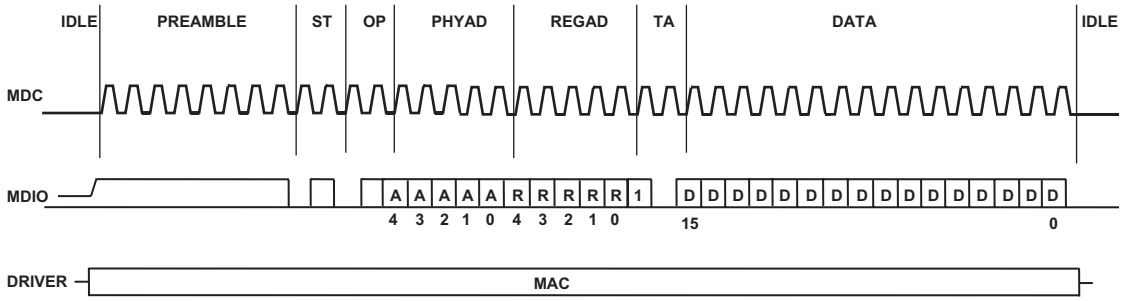


Figure 8-4. Station Management Write

Operation

The following sections describe the detailed operation of the Ethernet MAC peripheral.

MII Management Interface Operation

The MAC peripheral performs MDIO-protocol transfers in response to register read/write commands issued by the Blackfin processor. Three registers are provided to support MII management transfers:

- The EMAC_SYSCTL register contains the MDCDIV field which specifies the frequency of the MDC clock output in a ratio to the SCLK frequency, and must be initialized before any transfers.
- The EMAC_STADAT register holds the 16-bit data for read or write transfers.
- The EMAC_STAADD register supports several functions.
    - It commands the access—writes to it may initiate station management transfers, provided the STABUSY bit is set and provided that the interface is not already busy.
    - It selects the addressed device, register, and direction of the access.
    - It provides mode controls for MDIO preamble generation and station management transfer done interrupt.
    - It provides the STABUSY status bit indicating whether the interface is still busy performing a prior transfer.

As these serial accesses may require significant time (25.6us, or several thousand processor clock cycles at default rates), the Blackfin MAC provides an end-of-transfer interrupt to allow the processor to perform other

functions while station management transfers are in progress. Alternatively, the processor may determine the status of the transfer in progress by reading the STABUSY bit in the EMAC_STAADD register.

Receive DMA Operation

Data flow between the MAC and the Blackfin peripheral subsystem takes place via bidirectional descriptor based DMA. The element size for any DMA transfer to and from the Ethernet MAC is restricted to 32 bits. In the receive case, a queue or ring of DMA descriptor pairs are used, as illustrated in Figure 8-5. In the figure, data descriptors are labeled with an "A" and status descriptors are labeled with a "B."

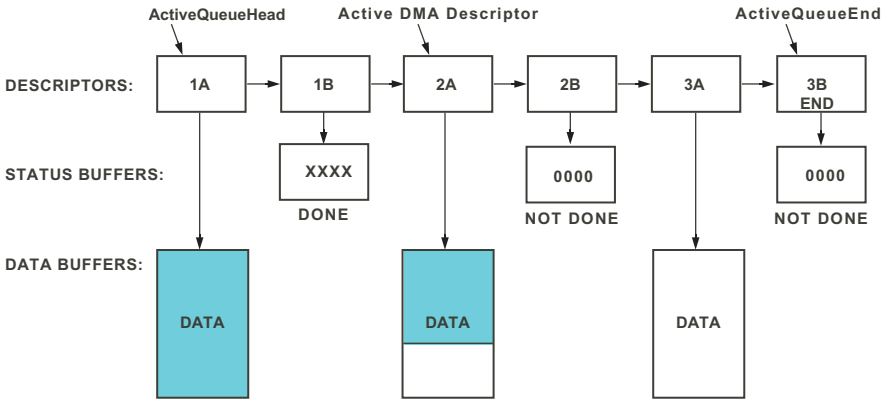


Figure 8-5. Ethernet MAC Receive DMA Operation

Receive DMA works with a queue or ring of DMA descriptor pairs structured as data and status.

- **Data** – The first descriptor in each pair points to a data buffer that is at least 1556 (0x614) bytes long and is 32-bit aligned. The descriptor `XCOUNT` field should be set to 0, because the MAC controls the actual buffer length.

- **Status** – The second descriptor points to a status buffer of either 4 or 8 bytes. The descriptor `XCOUNT` field should be set to 0, because the MAC controls the actual buffer length. After receiving and accepting any RX frame, the MAC writes a status word and optionally two IP checksum words to this status buffer. The `RXCKS` bit in the `EMAC_SYSCTL` register controls the generation of the two checksum words.

  Status words written by the MAC after frame reception have the same format as the current RX frame status register, and always have the receive complete bit set to 1. If the driver software initializes the length/status words to 0, it can reliably interrogate (poll) an RX frame's length/status word to determine if the DMA transfer of the data buffer is complete. Alternatively, status descriptors may be individually enabled to signal an interrupt when frame reception is complete.

The MAC and DMA operate on the active queue in this manner:

- **Start** – The queue is activated by initializing the DMA next descriptor pointer and then writing the `DMA_CONFIG` register. Meanwhile, the MAC listens to the MII, looking for a frame that passes its address filter.

- **Data** – When a matching frame is seen, the MAC transfers the frame data into the data buffer. The MAC does not initiate the DMA transfer until either the destination address filtering is complete, or the frame ends (if a runt frame).

- **End of frame** – At the end of the frame, the MAC issues a finish command to the DMA controller, causing it to advance to the next (status) descriptor.
- **Status** – The MAC then transfers the frame status into the status buffer. The frame status structure contains the length of the frame data. The MAC then issues another finish command to complete the status DMA buffer.
- **Interrupt** – Upon completion, the DMA may issue an interrupt, if the descriptor was programmed to do so. The DMA then advances to the next (data) descriptor, if any.

Frame Reception and Filtering

Frame data written to memory normally includes the Ethernet header (destination MAC address, source MAC address, and length/type field), the Ethernet payload, and the Frame Check Sequence (FCS) checksum, but not the preamble. If the `RXDWA` bit in `EMAC_SYSCTL` is 1, then the first 16-bit word is all-zero to pad the frame. The data written includes all complete bytes for which the received data valid (`ERxDV`) pin on the MII interface was asserted after but not including the start of frame delimiter (SFD) nibble (1011).The preamble and any other nibbles prior to the SFD are also not included.

The MAC applies two filtering mechanisms to received frames: the address filter and the frame filter. The address filter considers only the destination MAC address and provides control over the reception of unicast, multicast, and broadcast addresses. The frame filter considers the entire frame and provides control over reception of frames with errors and of MAC control frames.

The address filter is evaluated in the following sequence. Note that this sequence is in the same order as the related bits in the operating mode register, from LSB to MSB: HU, HM, PAM, PR, IFE, and DBF. The first few filter decisions are additive, while the last two are subtractive.

1. Initially, the address filter is true if the frame's MAC destination address (DA) is either the broadcast address (all 1s) or exactly matches the 48-bit station MAC address in the EMAC_ADDRHI and EMAC_ADDRLO registers.
2. **HU (hash unicast)** – If the HU bit is 1 and the DA is a unicast address which matches the hash table, the address filter is set to true.
3. **HM (hash multicast)** – If the HM bit is 1 and if the DA is a multicast address which matches the hash table, the address filter is set to true.
4. **PAM (pass all multicast)** – If the PAM bit is 1 and the DA is any multicast address, the address filter is set to true.
5. **PR (promiscuous)** – If the PR bit is 1, the address filter is set to true regardless of the frame DA.
6. **FLCE (flow control enable)** – If the FLCE bit in the flow control register is 1, and if the DA is an exact match to either the global multicast pause address or to the station MAC address, the address filter is set to true.
7. **IFE (inverse filter)** – If the IFE bit is 1 and the DA exactly matches the 48-bit station MAC address, the address filter is set to false.
8. **DBF (disable broadcast frames)** – If the DBF bit is 1 and the DA is the broadcast address, the address filter is set to false.

The hash table address filtering is configured with the EMAC_HASHLO and EMAC_HASHHI registers described on page 8-74.

The frame filter is evaluated in the following sequence. Note that the frame filter is updated as each byte of data is received. The frame filter can change from true to false during a frame, for example, upon DMA overrun, but can never change from false back to true.

1. Initially, the frame filter is set to true if the address filter is true, otherwise the frame filter is set to false.

2. **PCF (pass control frames)** – If the PCF bit is 0 and the frame is any valid supported MAC control frame (destination address is either the MAC address or the global multicast pause address; and the length/type field = 88-08, opcode = 0001, length = 64 bytes, and receiveOK = 1), then the frame filter is set to false.

3. **PBF (pass bad frames)** – If the PBF bit is 0 and the frame has any type of error except a frame fragment error, the frame filter is set to false. This rejects any frame for which any of these status bits are set: frame too long, alignment error, frame-CRC error, length error, or unsupported control frame. The frame filter does not reject frames on the basis of the out of range length field status bit. Note that this step may reject MAC control frames passed by PCF.

4. **PSF (pass short frames)** – If the PSF bit is 0 and the frame has a frame fragment error (frame contains less than 64 bytes), the frame filter is set to false. This step may reject frames which were passed by PCF or PBF.

5. **DMA RX overrun** – If the RX DMA FIFO overflows, the frame filter is set to false. If the FIFO overflows at a point where it contains parts of two frames, that is, the last data and status of frame A and the beginning data of frame B, then frame B is rejected by the frame filter and the MAC continues to try to deliver frame A's data and status.

Discarded Frames

Frames that fail the address filter are discarded immediately after the destination address is received, and neither their data nor their status values are written to memory via DMA. Frames that pass the address filter but fail the frame filter before 32 bytes are received are also discarded immediately. Once at least 32 bytes of a frame have been received, and if the address and frame filters both pass, the MAC begins to write the frame to memory via DMA RX.

Aborted Frames

Frames that fail the frame filter after 32 bytes have been received are aborted. The MAC issues a restart DMA control command, causing the current RX data DMA descriptor to be reinitialized with its starting address and counts. The aborted frame's status is not written to memory. Instead, the current DMA data and status buffers are recycled for the next RX frame. For all frames that pass both the address and frame filters, both data and status are written to memory via DMA.

Control Frames

If the FLCE (flow control enable) bit is set, MAC control frames (with the control type 88-08) whose DAs match either the station MAC address (with inverse filtering disabled) or the global pause multicast address will pass the address filter, and thus may also have status of receiveOK. If the frame also is a supported pause control frame (with length = 64 bytes, and opcode = pause = 00-01, and in full-duplex mode), then the frame filter condition is determined by the PCF (pass control frames) bit. If the frame is not also a supported pause control frame, then it is in error, and its frame filter condition depends on the PBF (pass bad frames) bit.

Examples

- To perform standard IEEE-802.3 filtering, clear the operating mode register bits HU, PR, IFE, DBF, PBF, and PSF. With these selections, the Ethernet MAC accepts error-free broadcast frames and

only those error-free unicast frames that exactly match the station MAC address. Set PAM to accept all multicast addresses, or set HM and program the multicast hash table registers to accept only a subset of multicast addresses.

- To accept all addresses, set PR and clear IFE and DBF in the operating mode register.
- To accept a set of several unicast addresses, set the HU bit and set the multicast hash table register bits which correspond to the desired addresses. Note that there is one set of hash table registers that apply to both unicast and multicast addresses, as selected by the HU or HM bits.
- To reject all addresses, set IFE and DBF, and clear HU, HM, PAM, and PR in the operating mode register.

RX Automatic Pad Stripping

If the ASTP bit in the MAC operating mode register is set, the pad bytes and FCS are stripped from any IEEE-type frame which was lengthened (padded) to reach the minimum Ethernet frame length of 64 bytes. This applies to frames where the Ethernet length/type field is less than 46 bytes, since the Ethernet header and FCS add 18 bytes. When pad stripping occurs, only the first Length/Type + 14 bytes are written to memory via DMA, and the frame length reported in the RX status register and in the RX status DMA buffer will be Length/Type + 14 rather than the actual number of received bytes.

Pad bytes are never stripped from typed Ethernet frames. Typed Ethernet frames are frames with a length/type field that takes the type interpretation because it is greater than or equal to 0x600 (1536).

RX DMA Data Alignment

If the RXDWA bit in the MAC system control register is clear, the MAC delivers the frame data via DMA to a 32-bit-aligned buffer in memory, including the Ethernet header and FCS. Because the Ethernet header is an odd number of 16-bit words long, this results in the frame payload being odd-aligned, which may be inconvenient for later processing.

If the RXDWA bit is set, however, the MAC prefixes one 16-bit pad word to the frame data with value 0x0000, resulting in a frame payload aligned on an even 16-bit boundary. See Figure 8-6.

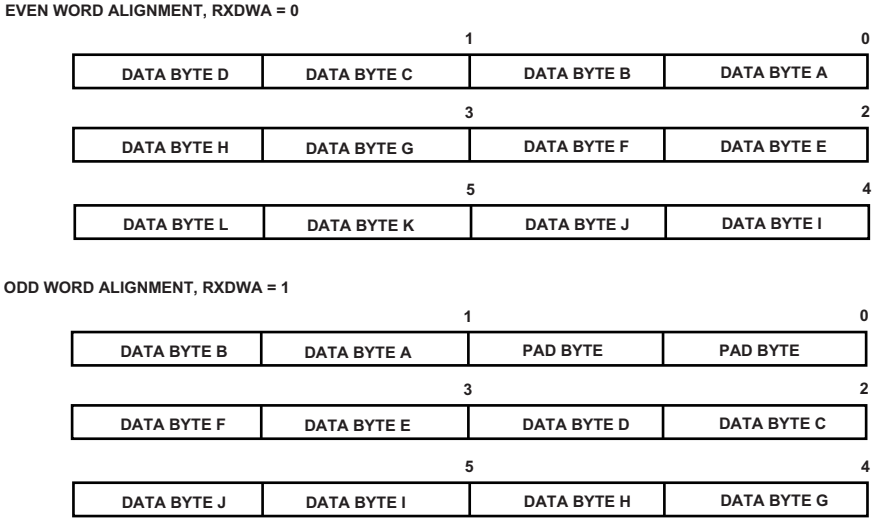


Figure 8-6. RX DMA Data Alignment

RX DMA Buffer Structure

The length of each RX DMA buffer must be at least 1556 (0x614) bytes. This is the maximum number of bytes that the MAC can deliver by DMA on any receive frame. Frames longer than the 1556-byte hardware limit

are truncated by the MAC. The 1556-byte hardware limit accommodates the longest legal Ethernet frames (1518 bytes for untagged frames, or 1522 bytes for tagged 802.1Q frames) plus a small margin to accommodate future standards extensions.

The MAC does not support RX DMA data buffers composed of more than one descriptor.

RX Frame Status Buffer

The RX frame status buffer is always an integer multiple of 32-bit words in length (either 1 or 2) and must always be aligned on a 32-bit boundary. The RX frame status buffer always contains a frame status word, and may also contain two 16-bit IP checksum words if the `RXCKS` bit in the MAC system control register is set.

To synchronize RX DMA and software, the `RX_COMP` semaphore bit may be used in the RX frame status word. This word is always the last word written via DMA in both status buffer formats, so a transition from 0 to 1 as seen by the processor always means that both the RX data and the status buffers are entirely valid.

Table 8-2 and Table 8-3 describe each of the status buffer formats.

Table 8-2. Receive Status DMA Buffer Format (Without IP Checksum)

Offset	Size	Description
0	32	RX frame status (Same format as the current RX frame status register)

Table 8-3. Receive Status DMA Buffer Format (With IP Checksum)

Offset	Size	Description
0	16	IP header checksum

Table 8-3. Receive Status DMA Buffer Format (With IP Checksum) (Cont'd)

Offset	Size	Description
2	16	IP payload checksum
4	32	RX frame status (Same format as the current RX frame status register)

RX Frame Status Classification

The RX frame status buffer and the RX current frame status register provide a convenient classification of each received frame, representing the IEEE-802.3 "receive status" code. The bit layout in the RX frame status buffer is identical to that in the RX current frame status register, and is arranged so that exactly one status bit is asserted for each of the possible receive status codes defined in IEEE-802.3 section 4.3.2. Note in the case of a frame that does not pass the frame filter, neither the frame data nor the status are delivered by DMA into the RX frame status buffer.

The priority order for determination of the receive status code is shown in Table 8-4.

Table 8-4. RX Receive Status Priority

Priority	Bit	Bit Name	IEEE receive status	Condition
1	20	DMA overrun	Undefined	The frame was not completely delivered by DMA
2	18	Frame fragment	Not received	The frame was less than the minimum 64 bytes and was discarded without reporting any other error
3	19	Address filter failed	Not received	The frame did not pass the address filter

Table 8-4. RX Receive Status Priority (Cont'd)

Priority	Bit	Bit Name	IEEE receive status	Condition
4	14	Frame too long	Frame too long	The frame size was more than the maximum allowable frame size (1518, 1522, or 1538 bytes for normal, VLAN1, or VLAN2 frames)
5	15	Alignment error	Alignment error	The frame did not contain an integer number of bytes, and also failed the CRC check
6	16	Frame CRC error	Frame check error	The frame failed CRC validation, and/or RX_ER was asserted during reception of the frame
7	17	Length error	Length error	The frame's length/type field was < 0x600 but did not match the actual length of the data received
8	13	Receive OK	receiveOK	The frame had none of the above conditions

RX IP Frame Checksum Calculation

The MAC calculates TCP/IP-style "raw" checksums of two useful segments of the frame data. Checksum calculation is enabled when the RXCKS bit is set to 1 in the MAC system control register.

The two checksum segments correspond to the typical position of the IP header and of the IP payload (see Table 8-5). The checksums are computed as a 16-bit one's-complement sum of the selected big-endian data words. In each summand, the most significant byte is stored in byte[1] and the least significant byte is stored in byte[2], counting bytes starting at 1. If an odd number of data bytes is to be summed, the final value is stored in the most significant byte and zero is stored in the least significant byte. One's complement addition can be done in ordinary unsigned integer arithmetic by adding the two numbers, followed by adding the carry-out bit value in at the least significant bit. This gives one's-complement

addition the property of being endian invariant, which makes it possible for software running on Blackfin's little-endian architecture to adjust the sums without explicit byte swapping. See also *RFC 1624* and its references.

The checksum calculation hardware provides an enormous boost to TCP/IP throughput and bandwidth, but requires checksum corrections in software to properly adapt to the details of each packet protocol. For example, TCP packets require the payload checksum to include a TCP pseudo-header made up of certain fields of the IP header. These fields should be added to the "raw" hardware-generated checksum. Similarly, the Ethernet FCS at the end of the frame should be deducted. These adjustments must be made before the IP checksum can be validated.

Table 8-5. IP Checksum Byte Ranges

Byte Number	Description	Included in IP Header Checksum?	Included in IP Payload Checksum?
1–14	Standard Ethernet header: dest address, src address, length/type	No	No
15–34	Typical IP header, without IP header options	Yes	No
35–N	IP payload, including Ethernet FCS	No	Yes

RX DMA Direction Errors

The RX DMA channel halts immediately after any transfer that sets the `RXDMAERR` bit in the `EMAC_SYSTAT` register. This bit is set if an RX data or RX status DMA request is granted by the RX DMA channel, but the DMA channel is programmed to transfer in the wrong (memory-read) direction. This could indicate a software problem in managing the RX DMA descriptor queue.

In order to facilitate software debugging, the RX DMA channel guarantees that the last transfer to occur is the one with the direction error. On an error, usually the current frame is corrupted. All later frames are ignored until the error is cleared. Since the MAC may have lost synchronization with the DMA descriptor queue, the RX channel must be disabled in order to clear the error condition.

To clear the error and resume operation, perform these steps:

1. Disable the MAC RX channel (clear the `RE` bit in the `EMAC_OPCODE` register).
2. Disable the DMA channel.
3. Clear the `RXDMAERR` bit in the `EMAC_SYSSTAT` register by writing 1 to it.
4. Reconfigure the MAC and the DMA engine as if starting from scratch.
5. Re-enable the DMA channel.
6. Re-enable the MAC RX channel.

Transmit DMA Operation

Figure 8-7 shows the transmit DMA operation.

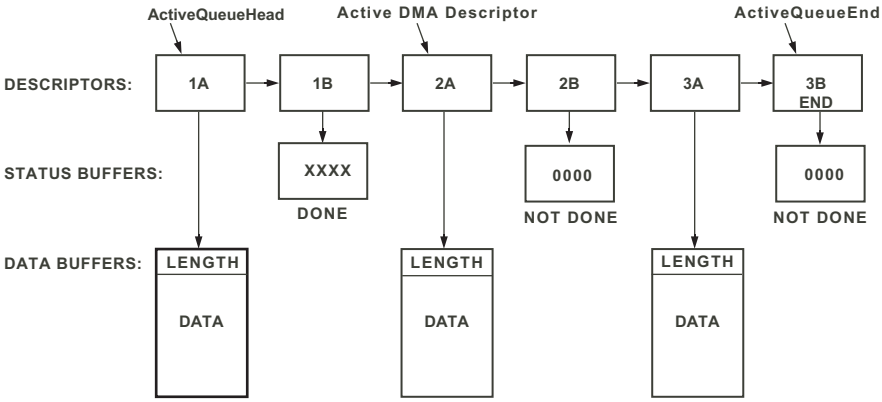


Figure 8-7. Ethernet MAC Transmit DMA Operation

Transmit DMA normally works with a queue or ring of DMA descriptor pairs.

- **Data** – The first descriptor in each pair points to a memory-read data buffer aligned on a 32-bit boundary. The first 16-bit word contains the length in bytes of the frame data, not including the length word or FCS. The descriptor `XCOUNT` field should be set to 0.
- **Status** – The second descriptor points to a 4-byte status buffer which is written via DMA at the end of the frame. The descriptor `XCOUNT` field should be set to 0, because the MAC controls the termination of the status buffer DMA. The driver software should initialize the status words to zero in advance.

  Status words written by the MAC after frame reception have the same format as the current TX frame status register and always have the transmit complete bit set to 1. Software can therefore interrogate (poll) a TX frame's status word to determine if the

transmission of its frame data is complete. Alternatively, status descriptors can be individually enabled to signal an interrupt when frame transmission is complete.

The MAC and DMA operate on the active queue in this manner:

- **Start** – The queue is activated by initializing the DMA `NEXT_DESC_PTR` register and then writing the `DMA_CONFIG` register.
- **Data** – The MAC transfers the frame length word and the first bytes of frame data into its TX data FIFO via DMA. When 32 bytes of data are present in the FIFO, and if the medium is unoccupied, the MAC begins transmission on the MII.
- **Collisions** – The MAC transfers data from memory via DMA into its FIFO, and then from the FIFO over the MII to the PHY. Collisions (in half-duplex mode) can occur at any time in the first 64 bytes of MII transmission, however, the MAC does not discard any of the data in its 96-byte TX FIFO until the first 64 bytes have been successfully transmitted. If a collision occurs during this collision window, and if retry is enabled (`DRTY = 0`), the MAC rewinds its FIFO pointer back to the start of the frame data and begins transmission again. No redundant DMA transfers are performed in such collisions. The MAC makes up to 16 attempts to transmit the frame in response to collisions (if not disabled by `DRTY`), each time backing off and waiting. After the 16th attempt, the frame is aborted—the MAC terminates data transmission by sending a finish command to the DMA controller, then sending frame status, and then proceeding to the next frame data.
- **Late collisions** – After the collision window is passed, the MAC allows DMA into the FIFO to resume and to overwrite older data. If a collision occurs after the 96th byte has been transferred into the FIFO by DMA (that is, after the FIFO has “wrapped around”),

then the MAC issues a restart command to the DMA controller to repeat the DMA of the current descriptor's data buffer (if enabled by the LCTRE bit).

- **End of frame** – At the end of the frame, the MAC issues a finish command to the DMA controller, causing it to advance to the next (status) descriptor. If the TX frame exceeds the maximum length limit (1560 bytes, or 0x618), the frame's DMA transfer is truncated. Only 1543 (0x607) are transmitted on the MII.
- **Status** – The MAC transfers the frame status into the status buffer.
- **Interrupt** – Upon completion, the DMA may issue an interrupt, if the descriptor was programmed to do so. The DMA then advances to the next (data) descriptor, if any.

Figure 8-8 shows an alternative descriptor structure. The frame length value and Ethernet MAC header are separated from the data payload in each frame.

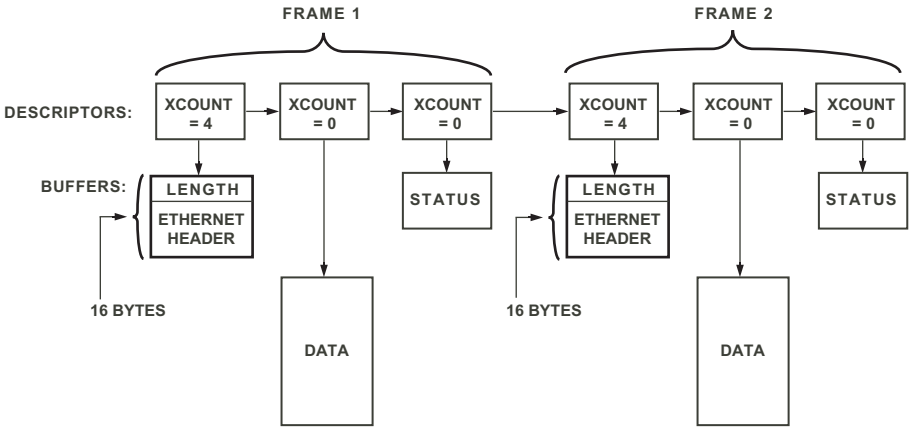


Figure 8-8. Alternative Descriptor Structure

Flexible Descriptor Structure

The Blackfin processor's DMA structure allows flexibility in the arrangement of TX frame data in memory. The frame data can be partitioned into segments, each with a separate DMA descriptor, which allows any of the first 88 bytes of DMA data (86 bytes of frame data) to reside in a separate data segment from the remainder of the frame. This permits the frame length word, the Ethernet MAC header, and even some higher level stack headers to be in one area of memory, while the payload data might be in another. The header and payload may even be in different memory spaces (some internal, some external). Each data buffer segment must be 32-bit aligned. In each frame, the `XCOUNT` field of all but the last data descriptor should be set to the actual length of the data buffers that they reference. As usual, the `XCOUNT` field of the last data descriptor should be set to 0 and the `XCOUNT` field of the status descriptor should be set to 0. The data after the first 88 bytes must all be contained in the data buffer of the last descriptor in the packet.

Multi-descriptor data formatting is not supported if retry is enabled upon late collisions (`LCRTE = 1` in the MAC operating mode register). The `LCRTE` bit must be 0 in order to use multiple DMA descriptors for transmit.

TX DMA Data Alignment

The MAC receives TX frame data via DMA from a 32-bit-aligned buffer in memory. If the `TXDWA` bit in the MAC system control register is clear, the first word of the MAC frame destination address should immediately follow the TX DMA length word. The MAC frame header starts at an odd word address and the MAC frame payload starts at an even word address.

If the `TXDWA` bit is set, the 16-bit TX DMA length word should be followed by a 16-bit pad word that the MAC ignores. The pad word is transferred over DMA but is not transmitted by the MAC to the PHY. The first word of the MAC frame destination address should immediately follow the pad word. The MAC frame header starts at an even word address and the MAC frame payload starts at an odd word address.

In all cases, the TX DMA length word specifies the number of bytes to be transfered via DMA, excluding the TX DMA length word itself. Specifically, when TXDWA is set, the TX DMA length word includes the length of the two pad bytes. See Figure 8-9.

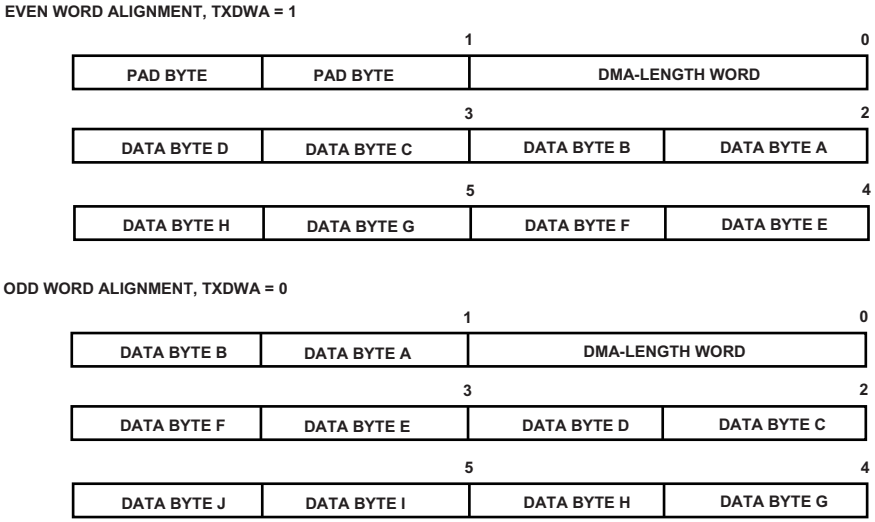


Figure 8-9. TX DMA Data Alignment

Late Collisions

If a frame's transmission is interrupted (for example, by a late collision) after the transmission of the first 64 bytes, the MAC can be programmed to either automatically retry the frame or to discard the frame. If the LCRTE bit in the MAC operating mode register is set, the MAC issues a restart command to the TX DMA channel and resets the DMA current address pointer to the start of the current DMA descriptor. This requires the frame data to be entirely contained in a single DMA descriptor.

If the LCRTE bit is clear and a late collision is detected, the MAC issues a finish command to the TX DMA controller, advancing the DMA channel to the status descriptor. The MAC then transfers the TX frame status to memory and advances to the next frame descriptor for data.

TX Frame Status Classification

The TX frame status buffer and the TX current frame status register provide a convenient classification of each received frame, representing the IEEE-802.3 "transmit status" code. The bit layout in the TX frame status buffer is identical to that in the TX current frame status register, and is arranged so that exactly one status bit is asserted for each of the possible transmit status codes defined in IEEE-802.3 section 4.3.2.

The priority order for determination of the transmit status code is shown in Table 8-6.

Table 8-6. TX Transmit Status Priority

Priority	Bit	Bit Name	IEEE transmit status	Condition
1	4	DMA underrun	Undefined	The frame was not completely delivered by DMA
2	2	Excessive collision	Excessive collision error	The frame was aborted because of too many (16) collisions, or because of excessive deferral
3	3	Late collision error	Late collision error status	The frame was aborted because of a late collision
4	14, 13	Loss of carrier, no carrier		Carrier sense was deasserted during some or all of the frame transmission (half-duplex only).
5	1	Transmit OK	Transmit OK	The frame had none of the above conditions

TX DMA Direction Errors

The TX DMA channel halts immediately after any transfer that sets the TXDMAERR bit in the EMAC_SYSTAT register. This bit is set if a TX data or status DMA request is granted by the DMA channel, but the DMA channel is programmed to transfer in the wrong direction. Data DMA should be memory-read; status DMA should be memory-write. TX DMA errors could indicate a software problem in managing the TX DMA descriptor queue.

In order to facilitate software debugging, the TX DMA channel guarantees that the last transfer to occur is the one with the direction error. On an error, usually the current frame is corrupted. Any later frames in the descriptor queue are not sent until the error is cleared. Since the MAC may have lost synchronization with the DMA descriptor queue, the TX channel must be disabled in order to clear the error condition.

To clear the error and resume operation, perform these steps:

1. Disable the MAC TX channel (clear the TE bit in the EMAC_OPCODE register).
2. Disable the DMA channel.
3. Clear the TXDMAERR bit in the EMAC_SYSSTAT register by writing 1 to it.
4. Reconfigure the MAC and the DMA engine as if starting from scratch.
5. Re-enable the DMA channel.
6. Re-enable the MAC TX channel.

Power Management

The Blackfin MAC can be programmed to trigger the following two types of power state transitions:

1. Wake from hibernate

   When the processor is in hibernate state (V_{DDINT} powered off) or any higher state, a low level on the $\overline{\text{PHYINT}}$ pin can wake the processor to the full on state (via RESET). This transition is enabled by setting the PHYWE bit to 1 in the VR_CTL register prior to powerdown (See "Dynamic Supply Voltage Control" in Chapter 20, Dynamic Power Management.)

   This pin may be connected to an INT output of the external PHY, if applicable. Many PHY devices provide such a pin (sometimes called $\overline{\text{MDINT}}$ or INTR). PHYs with interrupt capability may be programmed in advance via the MII management interface (MDC/MDIO) to assert the INT pin asynchronously upon detecting various conditions. Examples of INT conditions include link up, remote fault, link status change, auto-negotiation complete, and duplex and speed status change.

   Note that the $\overline{\text{PHYINT}}$ pin is general-purpose, and may be driven by any external device or left unused (pulled up to V_{DDIO}). It is not limited to use with external PHYs.

   When the ADSP-BF536/ADSP-BF537 is in either the hibernate or deep sleep state, the MAC is powered down. It is not possible to receive or transmit Ethernet frames in these states.

2. Wake from sleep

   When the processor is in the sleep state (or any higher state), the Ethernet MAC can remain powered up and can wake the processor to the active or full on states upon signalling an Ethernet event interrupt. The Ethernet event interrupts most useful for power management include:

   - Remote wakeup frame received, matching one of four programmable frame filters (see "Remote Wake-up Filters" on page 8-35).
   - Magic Packet™ detected (see "Magic Packet Detection" on page 8-34).
   - Any of the RX or TX frame status interrupts. Examples of these interrupts include: frame received (any frame), Broadcast frame received, VLAN1 frame received, and good frame received (which includes passing the address filters.).

   For example, the MAC could be programmed to wake the system upon receiving a frame with a particular group destination address, by setting the multicast frame received interrupt enable bit in the `EMAC_RX_IRQE` register and by selecting the appropriate address hash bins in the `EMAC_HASHLO/HI` multicast hash bin address filter registers.

Ethernet Operation in the Sleep State

When the ADSP-BF536/ADSP-BF537 is in the sleep state, the Ethernet MAC supports several levels of operation.

- The MAC may be powered down, by clearing RE and TE in the operating modes register. In this lowest-power state, the MAC's internal clocks do not run, and the MAC neither transmits nor responds to received frames. Note that the MAC will not receive a PAUSE control frame in this state.
- The MAC receiver may be partially powered up in a "wake-detect-only" state, but without enabling either the MAC transmitter or MAC DMA. This state is selected by:
    1. Setting RE and clearing TE in the operating modes register.
    2. Setting either the MPKE (magic packet wake enable) or RWKE (remote wakeup frame enable) bits in the MAC wakeup frame control and status register (EMAC_WKUP_CTL).
    3. Clearing the capture wakeup frame (CAPWKFRM) bit in EMAC_WKUP_CTL.

    When in the wake-detect-only state, the MAC receiver disables its DMA interface, and does not request any DMA transfers (whether data or status). Instead, the MAC receiver processes good incoming frames through its remote wake-up and/or Magic Packet filters. When a match is detected, the MAC signals a WAKEDET interrupt (setting the WAKEDET status bit in the EMAC_SYSSTAT register). DMA transfers do not resume until the CAPWKFRM bit is cleared.
- The MAC receiver may be fully powered up to both receive and/or transmit frames, provided that only external memory (for example, SDRAM) is used. Both the DMA data buffers and descriptor structures must be in external memory, since internal L1 is unavailable when core clocks are stopped.

This state is intended to be used with very restricted receive-frame filters, so that only certain specific frames are stored via DMA—perhaps only the frame(s) which caused the wakeup event itself. The transmit functionality permits the processor to enqueue a list of final frame transmissions before going to sleep.

The MAC can only transmit frames contained in DMA buffers set up by the processor prior to entering the sleep state. Once the last transmit frame has been sent, the transmitter and DMA channel pauses. Note that if the last TX DMA descriptor was programmed to signal an interrupt, the ADSP-BF536/ADSP-BF537 wakes from sleep at the conclusion of that transmission.

Similarly, the MAC can only receive as many frames as can be contained in the DMA buffers and descriptors allocated by the processor prior to entering the sleep state. Once the last receive frame has been filled, the DMA channel pauses, and if any further frames are received (beyond the capacity of the MAC RX FIFO), a DMA overrun occurs. Note that if the last RX DMA descriptor was programmed to signal an interrupt, the ADSP-BF536/ADSP-BF537 wakes from sleep after that frame was received.

Magic Packet Detection

The MAC can be programmed to detect a Magic Packet as a wakeup event. This is enabled by setting the `MPKE` bit (Magic Packet enable) bit in the `EMAC_WKUP_CTL` register. When the MAC receives the Magic Packet, it sets the `MPKS` (Magic Packet status) bit in the `EMAC_WKUP_CTL` register, which causes the Ethernet event interrupt to be asserted. The associated ISR should clear the interrupt by writing a 1 to the `MPKS` bit; writing a 0 has no effect.

A Magic Packet is any valid Ethernet frame which contains a specific 102-byte pattern derived from the MAC's 48-bit MAC address anywhere within the frame after the 12th byte (after the destination and source

address fields). This byte pattern consists of 6 consecutive bytes of 0xFFs followed by sixteen consecutive repeats of the MAC address of the MAC which is targeted for wakeup. See Figure 8-10.

Good Magic Packet frames exclude frame-too-short error, frame-too-long error, FCS error, Alignment error, and PHY error conditions.

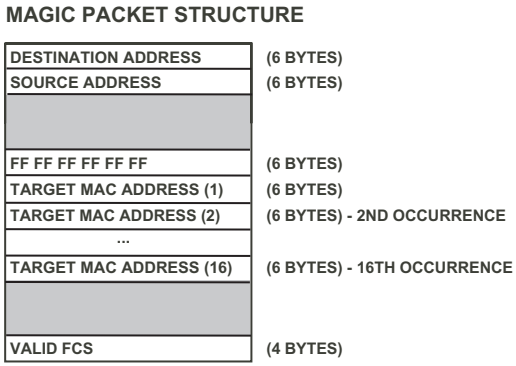


Figure 8-10. Magic Packet Structure

Remote Wake-up Filters

The Blackfin Ethernet MAC provides four independent remote wakeup frame filters for use while in powerdown. See Figure 8-11. These filters are enabled by setting the `RWKE` (remote wakeup enable) bit in the `EMAC_WKUP_CTL` register. Each filter works in parallel, simultaneously examining each incoming frame for a specific byte pattern. Each pattern is described by a byte offset to the start of the pattern within the frame, a 32-bit byte mask selecting bytes at that offset to include in the pattern, and a CRC-16 hash value of the selected bytes which identifies the pattern.

Each of the four filters sets a separate status bit (RWKS0–RWKS3) in the EMAC_WKUP_CTL register upon detection of their programmed frame pattern. The Ethernet event interrupt is asserted when any of these four status bits is set to 1; the WAKEDET bit in the EMAC_SYSSTAT register indicates the logical OR of all four of these bits and the MPKS (Magic Packet status) bit.

The remote wakeup interrupt is cleared by writing a 1 to the appropriate RWKS0–RWKS3 status bit(s). The WAKEDET bit is read-only and does not need to be explicitly cleared.

To program each remote wakeup filter:

1. The RWKE bit in the EMAC_WKUP_CTL register must be set to 1 (enables all four filters.).
2. The enable wakeup filter N bit in the EMAC_WKUP_FFCMD register must be set to 1 to enable filter N.
3. The wakeup filter N address type bit in the EMAC_WKUP_FFCMD register selects whether the target frame is unicast (if 0) or multicast (if 1).
4. The 8-bit pattern offset N field in the wakeup frame filter offsets register (EMAC_WKUP_FFOFF) selects the starting byte offset for the target data pattern, counting from 0 for the first byte of the MAC frame. The preamble and SFD bytes are not included.
5. The 32-bit wakeup frame byte mask register (EMAC_WKUP_FFMSKn) selects which of the 32 bytes starting at the selected offset into the frame will be considered in the pattern match. If the EMAC_WKUP_FFOFF register field contains the value K, then bit J of the EMAC_WKUP_FFMSKn register controls whether byte (J+K) of the frame will be compared, counting from 0. A value of 1 in the mask bit enables comparison.

6. The 16-bit wakeup filter N pattern CRC field in the `EMAC_WKUP_FFCRC0/1` register specifies the 16-bit CRC hash value expected for the wake-up pattern.

Each filter has a separate 16-bit CRC state register which is independently updated as the frame is received. The CRC state for filter N is only updated when an enabled byte is received; the CRC state remains unchanged if the current byte is not enabled by the filter's byte offset and mask registers.

Good frames whose CRC-16 value matches the specified value at the end of the selected pattern window will cause a wake-up event at the end of the frame. Good wake-up frames exclude frame-too-short error, frame-too-long error, alignment error, FCS error, PHY error, and length error conditions.

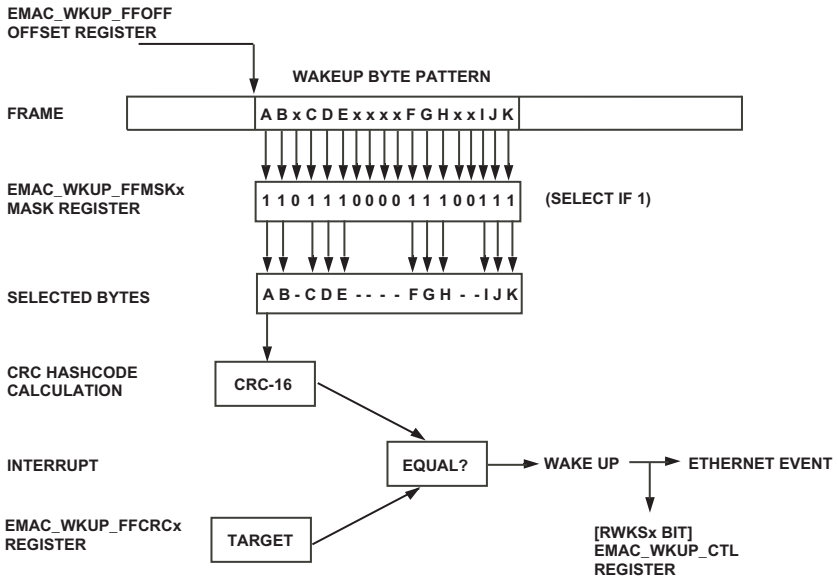


Figure 8-11. Remote Wakeup Filters

The CRC-16 hash value for a sequence of bytes may be calculated serially, with each byte processed LSB-first. The initial value of the CRC state is 0xFFFF (all 1s). For each input bit, the LFSR is shifted left one position, and the bit shifted out is XORed with the new input bit. The resulting feedback bit is then XORed into the LFSR at bit positions 15, 2, and 1. Thus the generator polynomial for this CRC is:

$G(x) = x^{16} + x^{15} + x^{2} + 1$

For example, if the wakeup pattern specified the single byte 0x12, or 0100_1000 (LSB first), the calculation of the wakeup CRC_16 is performed as shown in Table 8-7:

G polynomial = 1000 0000 0000 0101

Table 8-7. CRC-16 Hash Value Calculation

Bit In	XOR	MSB Bit	Feedback Bit	CRC State
				1111 1111 1111 1111, Initial = 0xFFFF
0		1	1	0111 1111 1111 1011
1		0	1	0111 1111 1111 0011
0		0	0	1111 1111 1110 0110
0		1	1	0111 1111 1100 1001
1		0	1	0111 1111 1001 0111
0		0	0	1111 1111 0010 1110
0		1	1	0111 1110 0101 1001
0		0	0	1111 1100 1011 0010, Final = 0xFCB2

Ethernet Event Interrupts

The Ethernet event interrupt is signalled to indicate that any or all of the conditions listed below are pending. Figure 8-12 shows the Ethernet event interrupts. In the ADSP-BF536 and ADSP-BF537, the Ethernet event

interrupt is signaled on peripheral interrupt ID 2 in the System Interrupt Controller (SIC), together with error conditions from a number of other peripherals. By default, peripheral interrupt ID 2 is mapped to `IVG7`.

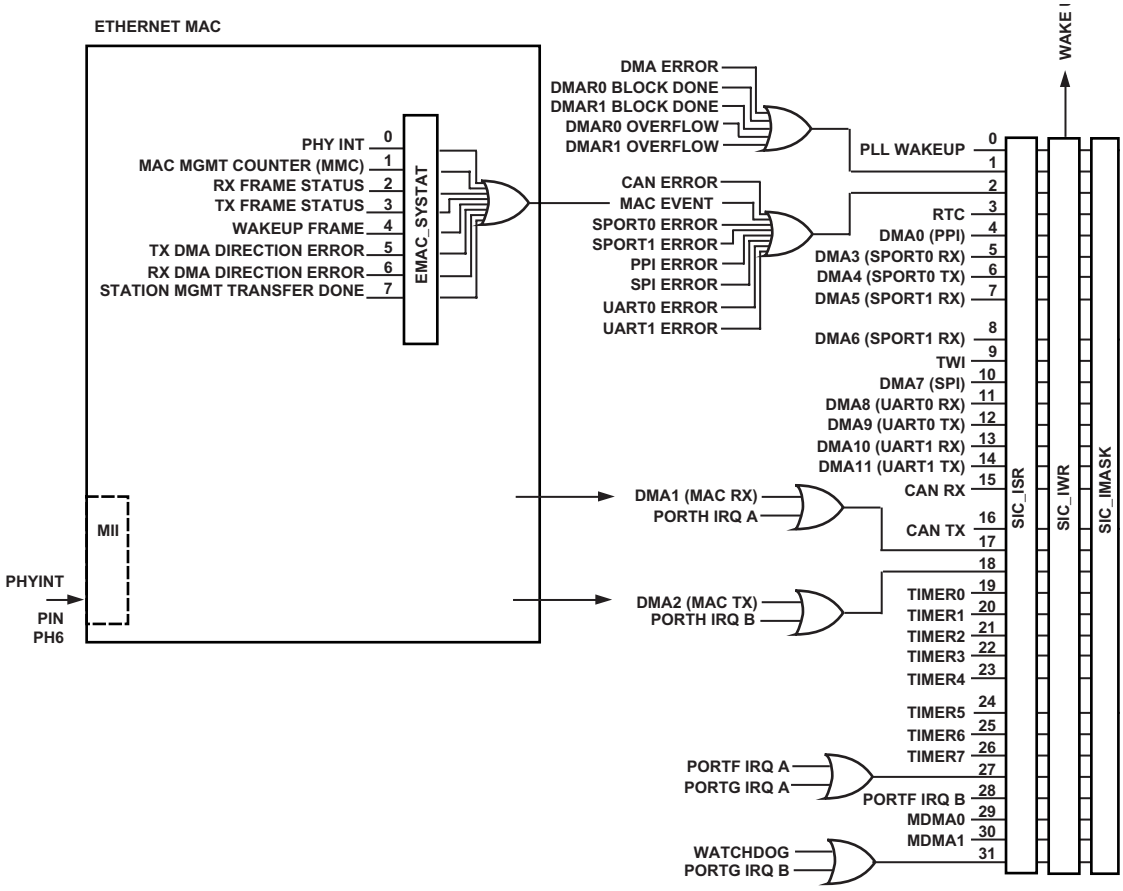


Figure 8-12. Ethernet MAC Event Interrupt

The handler for peripheral interrupt ID 2 should interrogate each of the peripherals assigned to peripheral interrupt ID 2 to determine which peripheral or peripherals are asserting an interrupt. To interrogate the

Ethernet MAC, the handler should read the Ethernet MAC system status register, as all of the MAC Ethernet event interrupt condition types are represented in that register.

These conditions result in an Ethernet event interrupt:

- **PHYINT interrupt** – Whenever the asynchronous $\overline{\texttt{PHYINT}}$ pin is asserted low, the PHYINT sticky bit in the MAC system status register is set to 1. The PHYINT interrupt condition is asserted whenever the logical AND of the PHYINT bit and the PHYIE enable bit in the Ethernet MAC system control register is 1. This condition is cleared by writing a 1 to the PHYINT bit.
- **Mac management counter (MMC) interrupt** – When any MMC counter reaches half of its maximum value (that is, transitions from 0x7FFF FFFF to 0x8000 0000), the corresponding bit in the MMC RX interrupt status register is set. An MMC interrupt is asserted whenever either:
    - the logical AND of the MMC RX interrupt status register and the MMC RX interrupt enable register is nonzero, or
    - the logical AND of the MMC TX interrupt status register and the MMC TX interrupt enable register is nonzero.

  The MMC interrupt condition is cleared by writing 1s to all of the MMC RX and/or TX interrupt status register bits which are enabled in the MMC RX/TX interrupt enable register.
- **RX frame status interrupt** – The RX frame status interrupt condition is signalled whenever the logical AND of the RX sticky frame status register and the RX frame status interrupt enable register is nonzero. This condition is cleared by writing 1s to all of the RX sticky frame status register bits that are enabled in the RX frame status interrupt enable register.

- **TX frame status interrupt** – The TX frame status interrupt condition is signalled whenever the logical AND of the TX sticky frame status register and the TX frame status interrupt enable register is nonzero. This condition is cleared by writing 1s to all of the TX sticky frame status register bits that are enabled in the TX frame status interrupt enable register.
- **Wakeup frame detected** – This bit is set when a wakeup event is detected by the MAC core (either a magic packet or a remote wakeup packet is accepted by the wakeup filters). This condition is cleared by writing a 1 to the `MPKS` and/or `RWKS` status bits in the wakeup control status register.
- **RX DMA direction error detected** – This bit is set if an RX data or status DMA request is granted by the DMA channel, but the DMA is programmed to transfer in the wrong (memory-read) direction. This could indicate a software problem in managing the RX DMA descriptor queue. This interrupt is non-maskable in the MAC and must always be handled. This condition is cleared by writing a 1 to the `RXDMAERR` bit in the MAC system status register.
- **TX DMA direction error detected** – This bit is set if a TX data or status DMA request is granted by the DMA channel, but the DMA is programmed to transfer in the wrong direction. Data DMA should be memory-read, status DMA should be memory-write. This could indicate a software problem in managing the TX DMA descriptor queue. This interrupt is non-maskable in the MAC and must always be handled. This condition is cleared by writing a 1 to the `TXDMAERR` bit in the MAC system status register.
- **Station management transfer done** – This bit is set when a station management transfer (on MDC/MDIO) has completed, provided the `STAIE` interrupt enable control bit is set in the station management address register.

When the MAC DMA engine is disabled, all the MAC peripheral requests are routed directly into the interrupt controller. This can manifest itself at startup as spurious interrupts. The solution is to configure the system in such a way that the DMA controller is always enabled before the MAC peripheral.

RX/TX Frame Status Interrupt Operation

The contents of the RX current frame status register indicate the result of the most recent frame receive operation. The register contents are updated just after the end of the frame is received on the MII and synchronized into the system clock domain.

The contents of the RX sticky frame status register are updated at the same time. Each applicable bit in the RX sticky frame status register is set if the corresponding bit in the RX current frame status register is set, otherwise the bit in the RX sticky frame status register keeps its prior value.

The RX frame status interrupt enable register is continuously bitwise ANDed with the contents of the RX sticky frame status register, and then all of the resulting bits are ORed together to produce the RX frame status interrupt condition. The state of the RX frame status interrupt condition is readable in the `RXFSINT` bit of the MAC system status register. This interrupt condition is cleared by writing 1s to all the bits in the RX sticky frame status register for which corresponding bits are set in the RX frame status interrupt enable register. Do not attempt to clear this interrupt condition by writing a 1 to the read only `RXFSINT` bit; such a write has no effect.

The three TX frame status registers (TX current frame status register, TX sticky frame status register, and TX frame status interrupt enable register) operate in a similar manner.

RX Frame Status Register Operation at Startup and Shutdown

After the RE bit in the EMAC_OPMODE register is cleared, the RX current frame status register, the RX sticky frame status register, and the RX frame status interrupt enable register hold their last state. Of course, the two writable registers can still be written.

In order to not confuse status from old and new frames, the RX current frame status register and the RX sticky frame status register are automatically cleared at a 0-to-1 transition of the RE bit. The RX frame status interrupt enable register is not cleared when the RE bit transitions from 0 to 1. It changes state only when written.

All three of these registers are cleared at system reset.

TX Frame Status Register Operation at Startup and Shutdown

After the TE bit in the EMAC_OPMODE register is cleared, the TX current frame status register, the TX sticky frame status register, and the TX frame status interrupt enable register hold their last state. Of course, the two writable registers can still be written.

In order to not confuse status from old and new frames, the TX current frame status register and the TX sticky frame status register are automatically cleared at a 0-to-1 transition of the TE bit. The TX frame status interrupt enable register is not cleared when the TE bit transitions from 0 to 1. It changes state only when written.

All three of these registers are cleared at system reset.

MAC Management Counters

The Blackfin Ethernet MAC provides a comprehensive set of 32-bit read-only MAC management counters, 24 for receive and 23 for transmit, in accordance with the “Layer Management for DTEs” specification in IEEE 802.3 Sec. 30.3. When enabled by setting the MMCE bit in the EMAC_MMC_CTL register, the counters are updated automatically at the

conclusion of each frame. The counters may be read at any time, but may not be written. The counters can be reset to zero all at once by writing the RSTC bit to 1.

The counters can be configured to be cleared individually after each read access if the CCOR bit is set to 1. This mode guarantees that no counts are dropped between the value returned by the read and the value remaining in the register.

Although this read operation has a side effect, the speculative read operation of the Blackfin core pipeline is properly handled by the MAC. During the time between the speculative read stage and the commit stage of the read instruction, the MMC block freezes the addressed counter so that intervening updates are deferred until the MMR read instruction is resolved.

For best results, to minimize the amount of time that any given MMC counter is frozen, it is suggested not to intentionally place MMC counter read instructions in positions that result in frequent speculative reads which are not ultimately executed. For example, MMC counter reads should not be placed in the shadow of frequently-mispredicted flow-of-control operations.

Continuous polling of any MMC register is not recommended. The MMC update process requires at least one SCLK cycle between successive reads to the same register, which may not occur if the register read is placed in a tight code loop. If the polling operation excludes the MMC update process, loss of information results.

The overflow behavior of the counters is configurable using the CROLL bit. The counters may be configured either to saturate at maximum value (CROLL = 0) or to roll over to zero and continue counting (CROLL = 1).

The range of the counters can be extended into software-managed counters (for example, 64-bit counters) by use of selectable MMC interrupts. The EMAC_MMC_RIRQE and EMAC_MMC_TIRQE MMC interrupt enable registers allow the programmer to select which counters should signal an

MMC interrupt on the Ethernet event interrupt line when they pass half of the maximum counter value. Even if interrupt latency is large, this mechanism makes it unlikely that any counter data is lost to overrun.

A recommended structure for the ISR for the MMC interrupt would be as follows. In this example, the CCOR (clear counter on read) bit is set to 1, and the CROLL (counter rollover) bit may also be set to 1.

1. In the ISR, read the SIC to determine which peripheral ID caused the interrupt.
2. If an Ethernet MAC event interrupt is pending, then read the EMAC_SYSTAT register. If any of the interrupt bits are set, then an Ethernet event interrupt is pending.
3. If the MMCINT bit is set, then read the EMAC_MMC_RIRQS and EMAC_MMC_TIRQS interrupt status registers. Then, for each bit that is set, read the corresponding MMC counter using CCOR (clear counter on read) mode, and add the result to the software-maintained counter.

   As an option, if the CROLL bit is set to 1, the ISR can check the count value to see if it is less than 0x8000 0000. This would indicate that the counter has somehow incremented beyond the maximum value (0xFFFF FFFF) and wrapped around to zero while the interrupt awaited servicing. In this case, the software could add an additional 2^{32} to its extended counter to repair the count deficit.
4. Write the interrupt-status values previously read from EMAC_MMC_RIRQS and EMAC_MMC_TIRQS back to those same registers, so that the bits which were 1 cause the corresponding interrupt status bits to be cleared in a write-1-to-clear operation. This guarantees that all the counter interrupts that are cleared are those that correspond to counters that have been read by the interrupt handler. If other counter(s) cross the half-maximum interrupt threshold after the “snapshot” of the EMAC_MMC_RIRQS and

EMAC_MMC_TIRQS was taken, then those interrupts are still correctly pending at the RTI; the interrupt handler is then re-entered and the remaining counter interrupts are handled in a second pass.

Programming Model

The following sections describe the Ethernet MAC programming model for a typical system. The initialization sequence can be summarized as follows.

1. Configure MAC MII pins.
    - Multiplexing scheme
    - CLKBUF
2. Configure interrupts.
3. Configure MAC registers.
    - MAC address
    - MII station management
4. Configure PHY.
5. Receive and transmit data through the DMA engine.

Configure MAC Pins

The first step is to configure the hardware interface between the MAC and the external PHY device.

Multiplexing Scheme

The MII interface pins are multiplexed with GPIO pins on port H. To configure a pin on port H for Ethernet MAC functionality, the PORTH_FER bit corresponding to that pin must be set to 1.

The MII management pins (MDC and MDIO) are available on port J. Note that these two pins are not multiplexed.

CLKBUF

The external PHY chip can be clocked with the buffered clock (CLKBUF) output from the Blackfin processor. In order to enable this clock output, the PHYCLKOE bit in the VR_CTL register must be set. Note that writes to VR_CTL take effect only after the execution of a PLL programming sequence.

Configure Interrupts

Next, the MAC interrupts and MAC DMA interrupts need to be configured to properly. Interrupt service routines should be installed to handle all applicable events. Refer to Figure 8-12 on page 8-39 for a graphical representation of how event signals are propagated through the interrupt controller. The status of the MAC interrupts can be sensed with the EMAC_SYSTAT register. However, the process of enabling these interrupts is achieved through a number of different registers.

- The PHYINT interrupt is enabled by setting the PHYIE bit in the EMAC_SYSCTL register.
- The MAC management counter (MMC) interrupt can be enabled through the EMAC_MMC_RIRQE and EMAC_MMC_TIRQE registers.
- The RX frame status and TX frame status interrupts can be enabled through the EMAC_RX_IRQE and EMAC_TX_IRQE registers, respectively.

- The wakeup frame events are controlled through the EMAC_WKUP_CTL register.
- The TX DMA direction error detected and RX DMA direction error detected interrupts are non-maskable. Therefore, an interrupt service routine to handle them should always be installed.
- The station management transfer done interrupt is enabled through the STAIE bit of the EMAC_STAADD register.

The DMA MAC receive and DMA MAC transmit functions are initialized to the DMA1 and DMA2 channels by default. The interrupts for the channels corresponding to the Ethernet MAC transfers should be unmasked and a corresponding ISR should be installed if a polling technique is not used.

Configure MAC Registers

After the interrupts are set up correctly, the MAC address registers and the MII protocol must be initialized.

MAC Address

Set the MAC address by writing to the EMAC_ADDRHI and EMAC_ADDRLO registers. Since the MAC address is a unique number, it is usually stored in a non-volatile memory like a flash device. In this way, every system using the Blackfin MAC peripheral can be easily programmed with a different MAC address during mass production.

MII Station Management

The following procedure should be used to set up the MII communications protocol with the external PHY device.

To perform a station management write transfer:

1. Initialize MDCDIV in the EMAC_SYSCTL register. The frequency of the MDC clock is SCLK / [2 * (MDCDIV + 1)]. Thus MDCDIV = (SCLK_Freq / MDC_Freq)/2 - 1. For typical 400ns (2.5MHz) MDC rate at SCLK = 125MHz, set MDCDIV to (125MHz / 2.5MHz) / 2 -1 = 50/2-1 = 24.
2. Write the data into EMAC_STADAT.
3. Write EMAC_STAADD with the PHY address, register address, STAOP = 1, STABUSY = 1, and desired selections for preamble enable and interrupt enable.
4. Do not initiate another read or write access until STABUSY reads 0 or until the station management done interrupt (if enabled) has been received. Accesses attempted while STABUSY = 1 are discarded.

To perform a station management read transfer:

1. Initialize MDCDIV.
2. Write EMAC_STAADD with the PHY address, register address, STAOP = 0, STABUSY = 1, and desired selections for preamble enable and interrupt enable.
3. Wait either while polling STABUSY or until the station management done interrupt (if enabled) has been received. Note that subsequent accesses attempted while STABUSY=1 are discarded. Proceed when STABUSY reads 0.
4. Read the data from EMAC_STADAT.

Configure PHY

After the MII interface is configured, the PHY can be programmed with the EMAC_STAADD and EMAC_STADAT registers. Before configuration, the PHY is usually issued a soft reset. Depending of the capabilities of the specific PHY device, the configurable options might include auto-negotiation, link speed, and whether the transfers are full-duplex or half-duplex. The PHY device may also be set up to assert an interrupt on certain conditions, such as a change of the link status.

Receive and Transmit Data

Data transferred over the MAC DMA must be handled with a descriptor-based DMA queue. Refer to Figure 8-5 on page 8-11 and Figure 8-7 on page 8-24 for a graphical representation of a receive queue and transmit queue, respectively.

An Ethernet frame header is placed in front of the payload of each data buffer. The data buffer structure is described in Table 8-8.

Table 8-8. Frame Header

Field	Size in Bytes
Frame size (Tx only)	2
Destination MAC address	6
Source MAC address	6
Length/type	2
Data Payload	Determined by the length/type field

Receiving Data

In order to receive data, memory buffers must be allocated to construct a queue of DMA data and status descriptors. If the RXDWA bit in EMAC_SYSCTL is 0, then the first item in the receive frame header is the destination MAC address. If the RXDWA bit in EMAC_SYSCTL is 1, then the first 16-bit word is all-zero to pad the frame, and the second item is the destination MAC address. The DMA engine is then configured through the DMA_CONFIG register. After the DMA is set up, the MAC receive functionality is enabled by setting the RE bit in EMAC_OPMODE. Completion can be signaled by interrupts or by polling the DMA status registers.

Transmitting Data

To transmit data, memory buffers must be allocated to construct a queue of DMA data and status descriptors. The first 16-bit word of the data buffers is written to signify the number of bytes in the frame. The DMA engine is then configured through the DMA_CONFIG register. After the DMA is set up, the MAC transmit functionality is enabled by setting the TE bit in EMAC_OPMODE. Completion can be signaled by interrupts or by polling the DMA status registers.

Ethernet MAC Register Definitions

The MAC register set is broken up into three groups corresponding to the peripheral's major system blocks:

- Control-status register group (MAC block)
- System interface register group (SIF block)
- MAC management counter register group (MMC block)

Most registers require 32-bit accesses, but certain registers have only 16 or fewer functional bits and can be accessed with either 16-bit or 32-bit MMR accesses.

Table 8-9 shows the functions of the MAC registers. MMC counter registers are found in Table 8-10 on page 8-54.

Table 8-9. MAC Register Mapping

Register Name	Function	Notes
Control-Status Register Group		
EMAC_OPMODE	MAC operating mode	Enables the Ethernet MAC transmitter.
EMAC_ADDRLO	MAC address low	Used with EMAC_ADDRHI to set the MAC address.
EMAC_ADDRHI	MAC address high	Used with EMAC_ADDRLO to set the MAC address.
EMAC_HASHLO	MAC multicast hash table low	Used with EMAC_HASHHI to hold the multicast hash table.
EMAC_HASHHI	MAC multicast hash table high	Used with EMAC_HASHLO to hold the multicast hash table.
EMAC_STAADD	MAC station management address	
EMAC_STADAT	MAC station management data	
EMAC_FLC	MAC flow control	
EMAC_VLAN1	MAC VLAN1 tag	
EMAC_VLAN2	MAC VLAN2 tag	
EMAC_WKUP_CTL	MAC wakeup frame control and status	
EMAC_WKUP_FFMSK0	MAC wakeup frame 0 byte mask	

Table 8-9. MAC Register Mapping (Cont'd)

Register Name	Function	Notes
EMAC_WKUP_FFMSK1	MAC wakeup frame 1 byte mask	
EMAC_WKUP_FFMSK2	MAC wakeup frame 2 byte mask	
EMAC_WKUP_FFMSK3	MAC wakeup frame 3 byte mask	
EMAC_WKUP_FFCMD	MAC wakeup frame filter commands	
EMAC_WKUP_FFOFF	MAC wakeup frame filter offsets	
EMAC_WKUP_FFCRC0	MAC wakeup frame filter CRC0/1	
EMAC_WKUP_FFCRC1	MAC wakeup frame filter CRC2/3	
System Interface Register Group		
EMAC_SYSCTL	MAC system control	
EMAC_SYSTAT	MAC system status	
EMAC_RX_STAT	Ethernet MAC RX current frame status	
EMAC_RX_STKY	Ethernet MAC RX sticky frame status	
EMAC_RX_IRQE	Ethernet MAC RX frame status interrupt enable	
EMAC_TX_STAT	Ethernet MAC TX current frame status	
EMAC_TX_STKY	Ethernet MAC TX sticky frame status	
EMAC_TX_IRQE	Ethernet MAC TX frame status interrupt enable	

Table 8-9. MAC Register Mapping (Cont'd)

Register Name	Function	Notes
EMAC_MMC_RIRQS	Ethernet MAC MMC RX interrupt status	
EMAC_MMC_RIRQE	Ethernet MAC MMC RX interrupt enable	
EMAC_MMC_TIRQS	Ethernet MAC MMC TX interrupt status	
EMAC_MMC_TIRQE	Ethernet MAC MMC TX interrupt enable	
MAC Management Counter Register Group		
EMAC_MMC_CTL	MAC management counters control	For a list of the MMC counter registers, see Table 8-10.

Table 8-10. MAC Management Counter Registers

MMR Address	Register Name (IEEE Name) IEEE 802.3 Reference	Description
0xFFC0 3100	EMAC_RXC_OK (FramesReceivedOK) 30.3.1.1.5	Holds a count of frames that are successfully received. This does not include frames received with frame-too-long, FCS, length or alignment errors, or frames lost due to internal MAC sublayer (DMA/FIFO) errors. This also excludes frames with frame-too-short errors, or frames that do not pass the address filter as indicated by the receive frame accepted status bit. Such frames are not considered to be received by the station, and are not considered errors.

Table 8-10. MAC Management Counter Registers (Cont'd)

MMR Address	Register Name (IEEE Name) IEEE 802.3 Reference	Description
0xFFC0 3104	EMAC_RXC_FCS (FrameCheckSequenceErrors) 30.3.1.1.6	Holds a count of receive frames that are an integral number of octets in length and do not pass the FCS check. This does not include frames received with frame-too-long or frame-too-short (frame fragment) errors. This also excludes frames with frame-too-short errors, or which do not pass the address filter.
0xFFC0 3108	EMAC_RXC_ALIGN (AlignmentErrors) 30.3.1.1.7	Holds a count of frames that are not an integral number of octets in length and do not pass the FCS check. This counter is incremented when the receive status is reported as alignment error. This also excludes frames with frame-too-short errors, or which do not pass the address filter.
0xFFC0 310C	EMAC_RXC_OCTET (OctetsReceivedOK) 30.3.1.1.14	Holds a count of data and padding octets in frames that are successfully received. This does not include octets in frames received with frame-too-long, FCS, length or alignment errors, or frames lost due to internal MAC sub-layer errors. This also excludes frames with frame-too-short errors, or which do not pass the address filter.
0xFFC0 3110	EMAC_RXC_DMAOVF (FramesLostDueToIntMAC RcvError) 30.3.1.1.15	Holds a count of frames that would otherwise be received by the station, but could not be accepted due to an internal MAC sublayer receive error. If this counter is incremented, then none of the other receive counters are incremented. This counts frames truncated during DMA transfer to memory, as indicated by the DMA overrun status bit.

Table 8-10. MAC Management Counter Registers (Cont'd)

MMR Address	Register Name (IEEE Name) IEEE 802.3 Reference	Description
0xFFC0 3114	EMAC_RXC_UNICST (UnicastFramesReceivedOK) No IEEE reference	Holds a count of frames counted by the EMAC_RXC_OK register that are not counted by the EMAC_RXC_MULTI or the EMAC_RXC_BROAD register.
0xFFC0 3118	EMAC_RXC_MULTI (MulticastFramesReceivedOK) 30.3.1.1.21	Holds a count of frames that are successfully received and are directed to an active non-broadcast group address. This does not include frames received with frame-too-long, FCS, length or alignment errors, or frames lost due to internal MAC sublayer error. This also excludes frames with frame-too-short errors, or that do not pass the address filter.
0xFFC0 311C	EMAC_RXC_BROAD (BroadcastFramesReceivedOK) 30.3.1.1.22	Holds a count of frames that are successfully received and are directed to the broadcast group address. This does not include frames received with frame-too-long, FCS, length or alignment errors, or frames lost due to internal MAC sublayer error. This also excludes frames with frame-too-short errors, or that do not pass the address filter.

Table 8-10. MAC Management Counter Registers (Cont'd)

MMR Address	Register Name (IEEE Name) IEEE 802.3 Reference	Description
0xFFC0 3120	EMAC_RXC_LNERRI (InRangeLengthErrors) 30.3.1.1.23	Holds a count of frames with a length/type field value between the minimum unpadded MAC client data size and the maximum allowed MAC client data size, inclusive, that does not match the number of MAC client data octets received. The counter also increments when a frame has a length/type field value less than the minimum allowed unpadded MAC client data size and the number of MAC client data octets received is greater than the minimum unpadded MAC client data size. This also excludes frames with frame-too-short errors (less than the minimum unpadded MAC client data size), or that do not pass the address filter.
0xFFC0 3124	EMAC_RXC_LNERRO (OutOfRangeLengthField) 30.3.1.1.24	Holds a count of frames with a Length field value greater than the maximum allowed LLC data size. This also excludes frames with frame-too-short errors, or that do not pass the address filter.
0xFFC0 3128	EMAC_RXC_LONG (FrameTooLongErrors) 30.3.1.1.25	Holds a count of frames received that exceed the maximum permitted frame size. This counter is incremented when the status of a frame reception is "frame too long." This also excludes frames with frame-too-short errors, or that do not pass the address filter.

Table 8-10. MAC Management Counter Registers (Cont'd)

MMR Address	Register Name (IEEE Name) IEEE 802.3 Reference	Description
0xFFC0 312C	EMAC_RXC_MACCTL (MACControlFramesReceived) 30.3.3.4	Holds a count of MAC control frames passed by the MAC sublayer to the MAC control sublayer. This counter is incremented upon receiving a valid frame with a Length/Type field value equal to 88-08. While the control frame may be received by the Ethernet MAC and yet not be delivered to the MAC client by DMA, depending on the state of the PCF bit, the control frame is still counted by this counter.
0xFFC0 3130	EMAC_RXC_OPCODE (UnsupportedOpcodesReceived) 30.3.3.5	Holds a count of MAC control frames received that contain an opcode that is not supported by the device. This counter is incremented when a receive frame function call returns a valid frame with a length/type field value equal to the reserved type, and with an opcode for a function that is not supported by the device. Only opcode 00-01(pause) is supported by the Ethernet MAC.
0xFFC0 3134	EMAC_RXC_PAUSE (PAUSEMACCtrlFramesReceived) 30.3.4.3	Holds a count of MAC control frames passed by the MAC sublayer to the MAC control sublayer. This counter is incremented when a receive frame function call returns a valid frame with both a length/type field value equal to 88-08 and an opcode indicating the pause operation (00-01). This counter does not include or exclude frames on the basis of address, even though pause frames are required to contain the MAC control pause multicast address.

Table 8-10. MAC Management Counter Registers (Cont'd)

MMR Address	Register Name (IEEE Name) IEEE 802.3 Reference	Description
0xFFC0 3138	EMAC_RXC_ALLFRM (FramesReceivedAll) No IEEE reference	Holds a count of all frames or frame fragments detected by the Ethernet MAC, regardless of errors and regardless of address, except for DMA overrun frames.
0xFFC0 313C	EMAC_RXC_ALLOCT (OctetsReceivedAll) No IEEE reference	Holds a count of all octets in frames or frame fragments detected by the Ethernet MAC, regardless of errors and regardless of address, except for DMA overrun frames.
0xFFC0 3140	EMAC_RXC_TYPED (TypedFramesReceived) No IEEE reference	Holds a count of all frames received with a length/type field greater than or equal to 0x600. This does not include frames received with frame-too-long, frame-too-short, FCS, length or alignment errors, frames lost due to internal MAC sublayer error, or that do not pass the address filter.
0xFFC0 3144	EMAC_RXC_SHORT (FramesLenLt64Received) No IEEE reference	Holds a count of all frame fragments detected with frame-too-short errors (length < 64 bytes), regardless of address filtering or of any other errors in the frame.
0xFFC0 3148	EMAC_RXC_EQ64 (FramesLenEq64Received) No IEEE reference	Holds a count of all good frames (with status receiveOK) that have a length of exactly 64 bytes.
0xFFC0 314C	EMAC_RXC_LT128 (FramesLen65_127Received) No IEEE reference	Holds a count of all good frames (with status receiveOK) that have a length between 65 and 127 bytes, inclusive.
0xFFC0 3150	EMAC_RXC_LT256 (FramesLen128_255Received) No IEEE reference	Holds a count of all good frames (with status receiveOK) that have a length between 128 and 255 bytes, inclusive.

Table 8-10. MAC Management Counter Registers (Cont'd)

MMR Address	Register Name (IEEE Name) IEEE 802.3 Reference	Description
0xFFC0 3154	EMAC_RXC_LT512 (FramesLen256_511Received) No IEEE reference	Holds a count of all good frames (with status receiveOK) that have a length between 256 and 511 bytes, inclusive.
0xFFC0 3158	EMAC_RXC_LT1024 (FramesLen512_1023Received) No IEEE reference	Holds a count of all good frames (with status receiveOK) that have a length between 512 and 1023 bytes, inclusive.
0xFFC0 315C	EMAC_RXC_GE1024 (FramesLen1024_MaxReceived) No IEEE reference	Holds a count of all good frames (with status receiveOK) that have a length greater than or equal to 1024 bytes. This does not include frames with a frame-too-long error.
0xFFC0 3180	EMAC_TXC_OK (FramesTransmittedOK) 30.3.1.1.2	Holds a count of frames that are successfully transmitted. This counter is incremented when the transmit status is reported as transmit OK.
0xFFC0 3184	EMAC_TXC_1COL (SingleCollisionFrames) 30.3.1.1.3	Holds a count of frames that are involved in a single collision and are subsequently transmitted successfully. This counter is incremented when the result of a transmission is reported as transmit OK and the attempt value is 2.
0xFFC0 3188	EMAC_TXC_GT1COL (MultipleCollisionFrames) 30.3.1.1.4	Holds a count of frames that are involved in more than one collision and are subsequently transmitted successfully. This counter is incremented when the transmit status is reported as transmit OK and the value of the attempts variable is greater than 2 and less then or equal to 16.

Table 8-10. MAC Management Counter Registers (Cont'd)

MMR Address	Register Name (IEEE Name) IEEE 802.3 Reference	Description
0xFFC0 318C	EMAC_TXC_OCTET (OctetsTransmittedOK) 30.3.1.1.8	Holds a count of data and padding octets in frames that are successfully transmitted. This counter is incremented when the transmit status is reported as transmit OK.
0xFFC0 3190	EMAC_TXC_DEFER (FramesWithDeferredXmissions) 30.3.1.1.9	Holds a count of frames whose transmission was delayed on its first attempt because the medium was busy (that is, at the start of frame, CRS is asserted, or was previously asserted within the minimum interframe gap). Frames involved in any collisions are not counted.
0xFFC0 3194	EMAC_TXC_LATECL (LateCollisions) 30.3.1.1.10	Holds a count of times that a collision has been detected later than one slot time from the start of the frame transmission. A late collision is counted twice, both as a collision and as a late collision. This counter is incremented when the number of late collisions detected in transmission of any one frame is nonzero.
0xFFC0 3198	EMAC_TXC_XS_COL (FramesAbortedDueToXSColls) 30.3.1.1.11	Holds a count of frames that are not transmitted successfully due to excessive collisions. This counter is incremented when the number of attempts equals 16 during a transmission. Note this does not include frames that are successfully transmitted on the last possible attempt.

Table 8-10. MAC Management Counter Registers (Cont'd)

MMR Address	Register Name (IEEE Name) IEEE 802.3 Reference	Description
0xFFC0 319C	EMAC_TXC_DMAUND (FramesLostDueToIntMACXmit Error) 30.3.1.1.12	Holds a count of frames that would otherwise be transmitted by the station, but could not be sent due to an internal MAC sublayer transmit error. If this counter is incremented, then none of the other transmit counters are incremented. This counts frames whose transmission is interrupted by incomplete DMA transfer from memory, as indicated by the DMA underrun status bit.
0xFFC0 31A0	EMAC_TXC_CRSERR (CarrierSenseErrors) 30.3.1.1.13	Holds a count of the number of times that carrier sense was not asserted or was deasserted during the transmission of a frame without collision.
0xFFC0 31A4	EMAC_TXC_UNICST (UnicastFramesXmittedOK) No IEEE reference	Holds a count of frames counted by the EMAC_TXC_OK register that are not counted by the EMAC_TXC_MULTI or the EMAC_TXC_BROAD register.
0xFFC0 31A8	EMAC_TXC_MULTI (MulticastFramesXmittedOK) 30.3.1.1.18	Holds a count of frames that are successfully transmitted to a group destination address other than broadcast.
0xFFC0 31AC	EMAC_TXC_BROAD (BroadcastFramesXmittedOK) 30.3.1.1.19	Holds a count of frames that are successfully transmitted to the broadcast address as indicated by the transmit status of OK.
0xFFC0 31B0	EMAC_TXC_XS_DFR (FramesWithExcessiveDeferral) 30.3.1.1.20	Holds a count of frames that deferred for an excessive period of time. This counter can only be incremented once per LLC transmission.

Table 8-10. MAC Management Counter Registers (Cont'd)

MMR Address	Register Name (IEEE Name) IEEE 802.3 Reference	Description
0xFFC0 31B4	EMAC_TXC_MACCTL (MACControlFramesTransmitted) 30.3.3.3	Holds a count of MAC control frames passed to the MAC sublayer for transmission. Note this counter is incremented only when a MAC pause frame is generated by writing to the EMAC_FLC register. The counter is not incremented for frames transmitted via the normal DMA mechanism which happen to contain valid MAC pause data.
0xFFC0 31B8	EMAC_TXC_ALLFRM (FramesTransmittedAll) No IEEE reference	Holds a count of all frames whose transmission has been attempted, regardless of success. Each frame is counted only once, regardless of the number of retry attempts.
0xFFC0 31BC	EMAC_TXC_ALLOCT (OctetsTransmittedAll) No IEEE reference	Holds a count of all octets in all frames whose transmission has been attempted, regardless of success. Each frame's length is counted only once, regardless of the number of retry attempts.
0xFFC0 31C0	EMAC_TXC_EQ64 (FramesLenEq64Transmitted) No IEEE reference	Holds a count of all frames with status transmit OK that have a length of exactly 64 bytes.
0xFFC0 31C4	EMAC_TXC_LT128 (FramesLen65_127Transmitted) No IEEE reference	Holds a count of all frames transmitted with status transmit OK that have a length between 65 and 127 bytes, inclusive.

Control-Status Register Group

This set of registers is used by the application software to configure and monitor the functionality of the MAC block.

EMAC_OPMODE Register

The EMAC_OPMODE register, shown in Figure 8-13, controls the address filtering and collision response characteristics of the Ethernet controller in both the RX and TX modes.

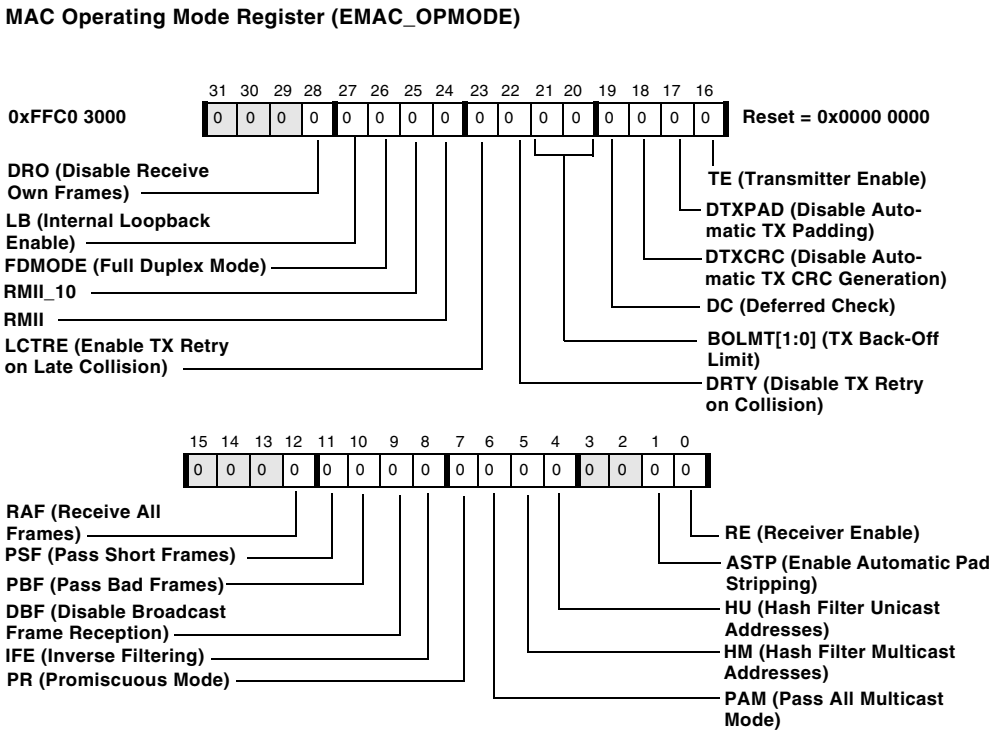


Figure 8-13. MAC Operating Mode Register

Additional information for the EMAC_OPMODE register bits includes:

- **Disable receive own frames** (DRO)

  When set in half-duplex mode, this bit blocks all frames transmitted by the MAC from being read into the receive path. This bit should be reset when the MAC is operating in full-duplex mode.

[1] Receive own frames disabled.

[0] Receive own frames enabled.

- **Internal loopback enable** (LB)

  When internal loopback is enabled, the frames transmitted by the MAC are internally redirected to the receive MAC port. The external MII port is inactive. The RX pins are ignored and the TX pins are set to TXEN = 0, TXD = 1111.

  [1] Internal loopback enabled.

  [0] Internal loopback not enabled.

- **Full duplex mode** (FDMODE)

  [1] Full duplex mode selected.

  [0] Half duplex mode selected.

- **RMII port speed selector** (RMII_10)

  When the interface is configured for RMII operation, software must query the PHY after any automatic negotiation to determine the link speed, and set the RMII port speed selector accordingly. This is because in RMII mode, the REFCLK input is always a constant speed regardless of link speed. In MII mode, by contrast, the PHY decreases the speed of the RXCLK and TXCLK to 2.5 MHz when the link speed is 10 M bits.

  [1] Speed for RMII port is 10 M bits.

  [0] Speed for RMII port is 100 M bits.

- **RMII mode** (RMII)

  This bit is used to select which interface, RMII or MII, is used by the MAC to transfer data to and from the external PHY. Note that MII and RMII modes use slightly different sets of package pins. Program different values into the PORTH_FER register accordingly.

  [1] RMII mode.

  [0] MII mode.

- **Enable TX retry on late collision** (LCTRE)

  [1] TX retry on late collision enabled.

  [0] TX retry on late collision not enabled.

- **Disable TX retry on collision** (DRTY)

  [1] TX retry on collision disabled.

  [0] TX retry on collision not disabled.

- **TX back-off limit** (BOLMT[1:0])

  This field sets an upper bound on the random back-off interval time before the MAC resends a packet in the event of a collision. The bound can be set to 1, 15, 255, or 1023 slot times (1 slot time = 128 MII clock cycles). Thus, varying levels of aggressiveness with regard to packet re-transmission can be selected.

  [00] The number of bits is 10 and the maximum back-off time is 1023 slots (relaxed, standard-compliant behavior).

  [01] The number of bits is 8 and the maximum back-off time is 255 slots.

[10] The number of bits is 4 and the maximum back-off time is 15 slots.

[11] The number of bits is 1 and the maximum back-off time is 1 slot (aggressive)

- **Deferral check** (`DC`)

  In half-duplex operation, a frame whose transmission defers to incoming traffic for longer than two maximum-length frame times is considered to have been excessively deferred. This time is `(2 x 1518 x 2) = 6072` MII clocks. See IEEE 802.3 section 5.2.4.1 for more information.

  [1] Enables the MAC to abort transmission of frames that encounter excessive deferral.

  [0] The MAC cannot abort transmission of frames due to excessive deferral.

- **Disable automatic TX CRC generation** (`DTXCRC`)

  [1] Automatic TX CRC generation is disabled.

  [0] Automatic TX CRC generation is enabled. Four CRC bytes are appended to the frame data.

- **Disable automatic TX padding** (`DTXPAD`)

  [1] Automatic TX padding of frames shorter than 64 bytes is disabled.

  [0] Automatic TX padding is enabled. Pad bytes with value 0 are appended to the data, followed by the CRC, so that the minimum frame size is 64 bytes.

- **Transmitter enable** (TE)

  The MAC transmitter is reset when TE is 0. A rising (0 to 1) transition on TE causes the TX current frame status register and the TX sticky frame status register to be reset. Note in RMII mode, only one reference clock (REFCLK) is used, and it belongs to the transmitter TXCLK pin. For this reason, always enable the transmitter, even in receive mode.

- **Receive all frames** (RAF)

  [1] Overrides the address and frame filters and causes all frames or frame fragments to be transferred to memory by DMA.

  [0] Does not override filters.

- **Pass short frame** (PSF)

  [1] Short frames are not rejected by the frame filter.

  [0] The frame filter rejects frames with frame-too-short errors (runt frames, or frames with total length less than 64 bytes not including preamble).

- **Pass bad frames** (PBF)

  [1] Pass bad frames enabled.

  [0] The frame filter rejects frames with FCS errors, alignment errors, length errors, frame-too-long errors, and DMA overrun errors.

- **Disable broadcast frame reception** (DBF)

  [1] Removes the broadcast address (all 1s) from the set of addresses passed by the address filter, overriding promiscuous mode.

  [0] Broadcast frame reception not disabled.

- **Inverse filtering** (IFE)

  [1] Removes the MAC address programmed in the EMAC_ADDRHI and EMAC_ADDRLO registers from the set of addresses passed by the address filter, overriding PR (promiscuous) and HU (hash unicast) modes. The effect is to block reception of a specific destination address.

  [0] Inverse filtering not enabled.

- **Promiscuous mode** (PR)

  [1] Promiscuous mode enabled, the address filter accepts all addresses.

  [0] Promiscuous mode not enabled.

- **Pass all multicast mode** (PAM)

  [1] All multicast frames are added to the set of addresses passed by the address filter.

  [0] Do not pass all multicast frames.

- **Hash filter multicast addresses** (HM)

  [1] Adds multicast addresses that match the hash table to the set of addresses passed by the address filter.

  [0] Does not add multicast addresses that match the hash table to the set of addresses passed by the address filter.

- **Hash filter unicast addresses** (HU)

  [1] Adds unicast addresses that match the hash table to the set of addresses passed by the address filter.

  [0] Does not add unicast addresses that match the hash table to the set of addresses passed by the address filter.

- **Automatic pad stripping enable** (ASTP)

  A received frame contains pad bytes if it is in IEEE format (the length/type field contains a length value < 0x600) and if the length value is less than 46 (corresponding to a frame whose total length including header and FCS is less than 64 bytes). If ASTP = 1, both the pad and the FCS bytes are removed from the received data.

  [1] Automatic pad stripping is enabled.

  [0] Automatic pad stripping is not enabled.

- **Receiver enable** (RE)

  The MAC transmitter is reset when RE is 0. A rising (0 to 1) transition on RE causes the RX current frame status register and the RX sticky frame status register to be reset.

EMAC_ADDRLO Register

The EMAC_ADDRLO register, shown in Figure 8-14, holds the low part of the unique 48-bit station address of the MAC hardware. Writes to this register must be performed while the MAC receive and transmit paths are both disabled. The byte order of address transfer is lowest significant byte first and lowest significant bit first on the MII. Thus EMAC_ADDRLO[3:0] is the first nibble transferred and EMAC_ADDRHI[15:12] is the last nibble.

For example, the address `00:12:34:56:78:9A` (where 00 is transferred first and 9A is transferred last) would be programmed as:

```
EMAC_ADDRLO = 0x56341200
EMAC_ADDRHI = 0x00009A78
```

MAC Address Low Register (EMAC_ADDRLO)
R/W, except cannot be written if RX or TX is enabled in the EMAC_OPMODE register.

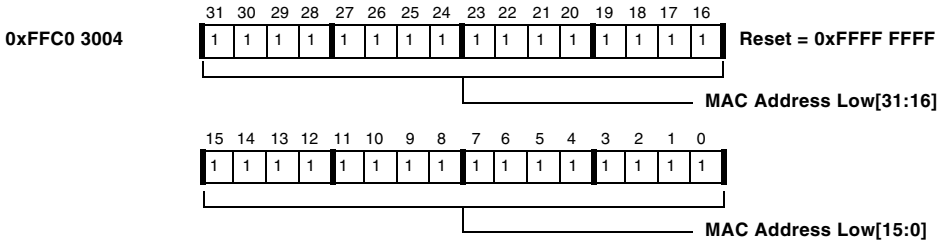


Figure 8-14. MAC Address Low Register

EMAC_ADDRHI Register

The `EMAC_ADDRHI` register, shown in Figure 8-15, holds the high part of the unique 48-bit station address of the MAC hardware. Writes to this register must be performed while the MAC receive and transmit paths are both disabled.

MAC Address High Register (EMAC_ADDRHI)
R/W, except cannot be written if RX or TX is enabled in the EMAC_OPMODE register.

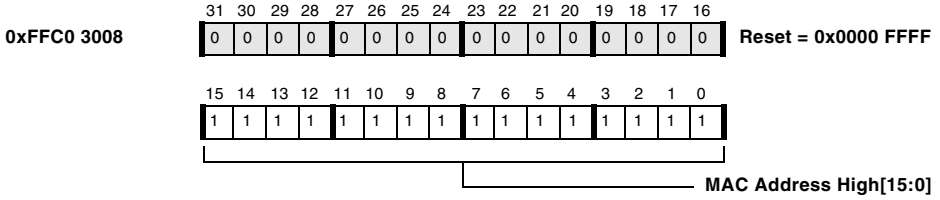


Figure 8-15. MAC Address High Register

EMAC_HASHLO and EMAC_HASHHI Registers

The EMAC_HASHLO register holds the values for bins 31–0 of the multicast hash table. The EMAC_HASHHI register holds the values for bins 63–32 of the multicast hash table. See Figure 8-16 and Figure 8-17.

The 64-bit multicast table is used for multicast frame address filtering. A cyclic redundancy check (CRC) based hash table scheme is used. After the destination address (6th byte) of the frame is received, the state of the CRC-32 checksum unit is sampled. This CRC-32 unit implements the IEEE 802.3 CRC algorithm used in validating the FCS field of the frame. The 6 most significant bits from this state identify one of 64 hash bins representing the frame's destination address. These 6 bits are then used to index into the two hash table registers and extract the corresponding hash bin enable bit. The most significant bit of this value determines the register to be used (high/low) while the other five bits determine the bit position within the register. A CRC value of 000000 selects bit 0 of the MAC multicast hash table low register and a CRC value of 111111 selects bit 31 of the MAC multicast hash table high register.

If the corresponding bit in the hash table register is set, the multicast frame is accepted. Otherwise, it is rejected. If the PM bit in the EMAC_OPMODE register is set, all multicast frames are accepted regardless of the hash values.

For example, consider the calculation of the hash bin for the MAC address 01.23.45.67.89.AB. The CRC algorithm uses an LFSR with the prime generator polynomial specified in IEEE 802.3 Sec 3.2.8:

$$G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^{8} + x^{7} + x^{5} + x^{4} + x^{2} + x + 1$$

The bits of the MAC address are fed in leftmost byte first, least significant bit first, in this sequence (left to right):

1000 0000 1100 0100 1010 0010 1110 0110 1001 0001 1101 0101

The 32-bit CRC register is initialized to all 1s. Then each input bit is processed as follows: first, the register is shifted left one place, shifting in a zero and shifting out the former MSB. The bit just shifted out is XORed with the current input bit, yielding the feedback bit. If this feedback bit is a 1, then the shift register contents are XORed with the generator polynomial value:

0x04C1 1BD7 = 0000 0100 1100 0001 0001 1101 1011 0111

Following this procedure, the CRC-32 for the MAC address is calculated. See Table 8-11.

Table 8-11. CRC-32 Calculation

Bit Number	Input Bit	MSB Bit	Feedback Bit	Next CRC Shift Register
Start				1111 1111 1111 1111 1111 1111 1111 1111
0	1	1	0	1111 1111 1111 1111 1111 1111 1111 1110
1	0	1	1	1111 1011 0011 1110 1110 0010 0100 1011
2	0	1	1	1111 0010 1011 1100 1101 1001 0010 0001
3	0	1	1	1110 0001 1011 1000 1010 1111 1111 0101
4	0	1	1	1100 0111 1011 0000 0100 0010 0101 1101
5	0	1	1	1000 1011 1010 0001 1001 1001 0000 1101
6	0	1	1	0001 0011 1000 0010 0010 1111 1010 1101
7	0	0	0	0010 0111 0000 0100 0101 1111 0101 1010
...				
46	0	1	1	1101 0011 1001 0111 1111 0100 0100 1001
47	1	1	0	1010 0111 0010 1111 1110 1000 1001 0010

The resulting six MSBs are 101001 = 0x29 = 41 decimal. The hash bin enable bit for this address is then bit 41 – 32 = 9 of the `EMAC_HASHHI` register.

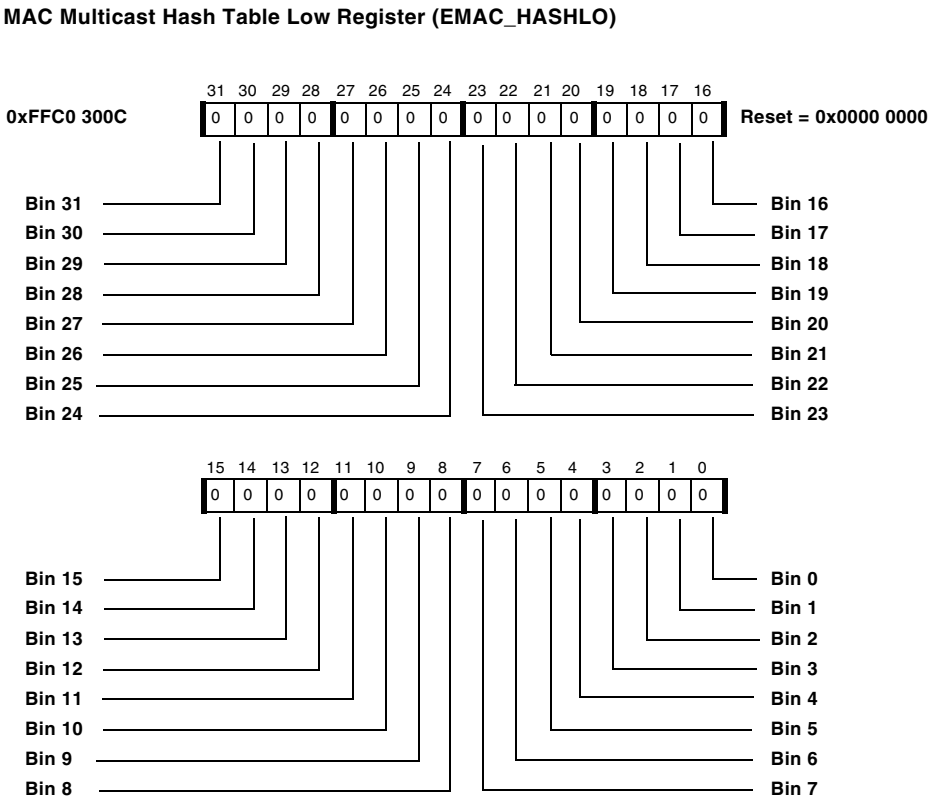


Figure 8-16. MAC Multicast Hash Table Low Register

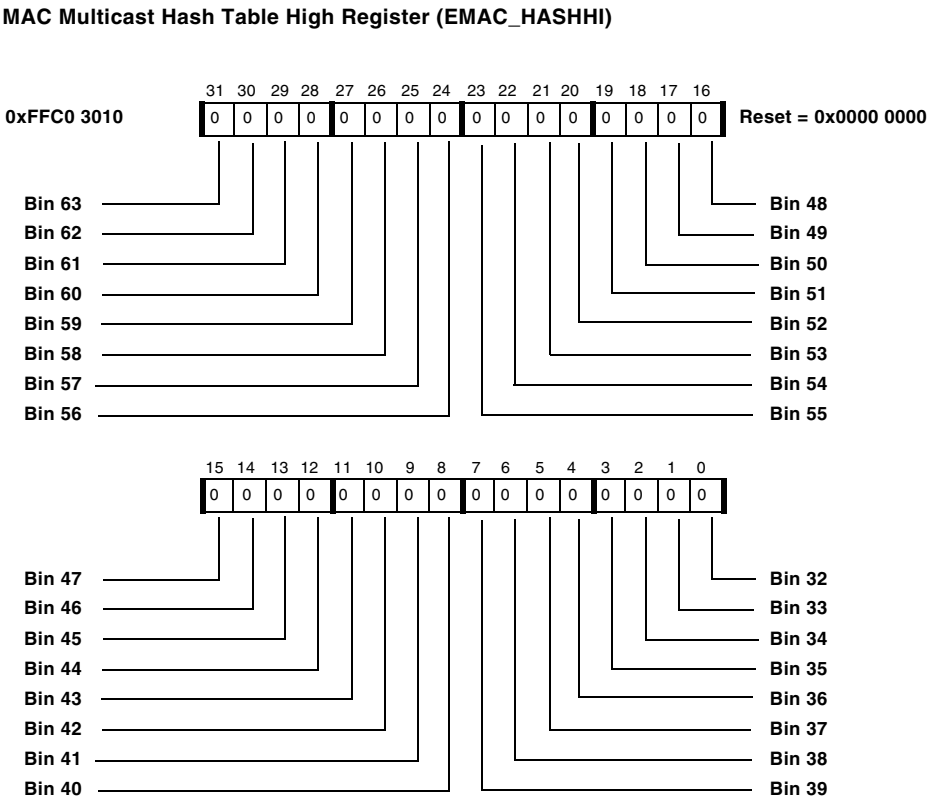


Figure 8-17. MAC Multicast Hash Table High Register

EMAC_STAADD Register

The EMAC_STAADD register, shown in Figure 8-18, controls the transactions between the MII management (MIM) block and the registers on the external PHY. These transactions are used to appropriately configure the PHY and monitor its performance.

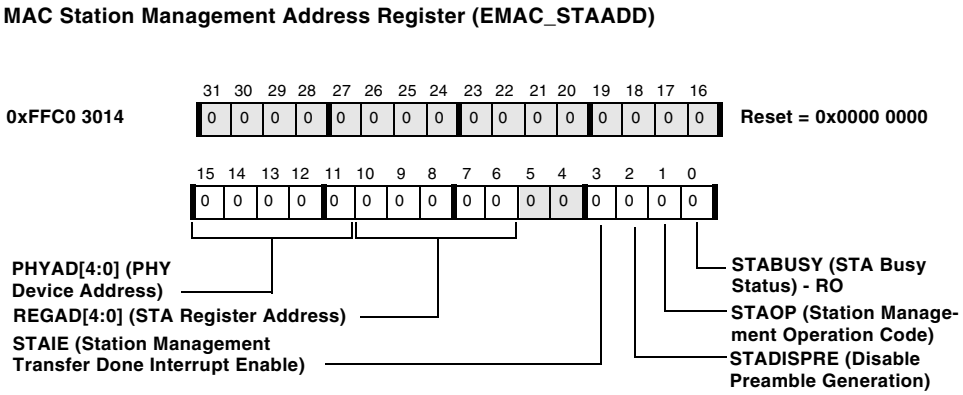


Figure 8-18. MAC Station Management Address Register

Additional information for the EMAC_STAADD register bits includes:

- **Station management transfer done interrupt enable** (STAIE)

  [1] Enables an Ethernet event interrupt at the completion of a station management register access (when STABUSY changes from 1 to 0).

  [0] Interrupt not enabled.

- **Disable preamble generation** (STADISPRE)

  [1] Preamble generation (32 ones) for station management transfers disabled.

  [0] Preamble generation for station management transfers not disabled.

- **Station management operation code** (STAOP)

  [1] Write.

  [0] Read.

- **STA busy status** (STABUSY)

  This bit should be set by the application software in order to initiate a station management register access. This bit is automatically cleared when the access is complete. The MAC ignores new transfer requests made while the serial interface is busy. Writes to the STA address or data registers are discarded if STABUSY is 1.

  [1] Initiate a station management register access across MDC/MDIO.

  [0] No operation.

EMAC_STADAT Register

The EMAC_STADAT register, shown in Figure 8-19, contains either the data to be written to the PHY register specified in the MAC station management address register, or the data read from the PHY register whose address is specified in the MAC station management address register.

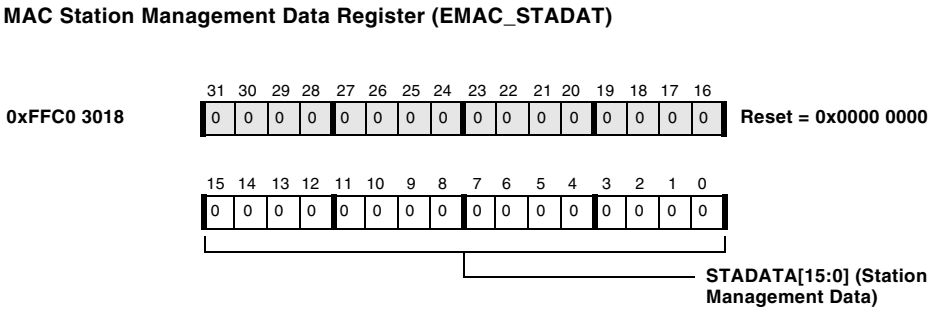


Figure 8-19. MAC Station Management Data Register

EMAC_FLC Register

The EMAC_FLC register, shown in Figure 8-20, controls the generation and reception of control frames by the MAC. The control frame fields are selected as specified in the IEEE 802.3 specification. When flow control is enabled, the MAC acts upon MAC control pause frames received without errors. When an error-free MAC control pause frame is received (with length/type = MacControl = 88-08 and with opcode = pause = 00-01), the transmitter defers starting new frames for the number of slot times specified by the pause time field in the control frame.

The MAC can also generate and transmit a MAC control pause frame when the EMAC_FLC register is written with FLCBUSY = 1 and FLCPAUSE equal to the number of slot times of deferral being requested.

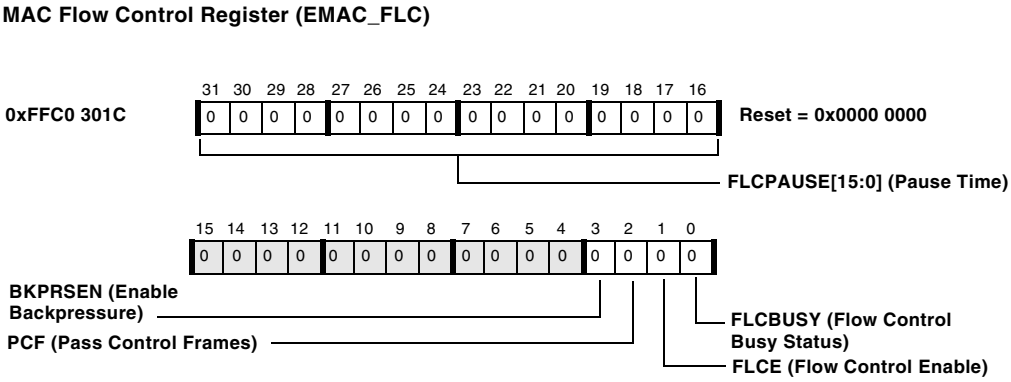


Figure 8-20. MAC Flow Control Register

Additional information for the EMAC_FLC register bits includes:

- **Pause time** (FLCPAUSE)

  The number of slot times for which the transmission of new frames is deferred.

- **Enable back pressure** (BKPRSEN)

  Available only in half-duplex mode, this bit can be used as a form of flow control.

  [1] Prevents frame reception by colliding with (continuously transmitting a jam pattern during) every incoming frame.

  [0] Transmit and receive function is normal.

- **Pass control frames** (PCF)

  When cleared, the PCF bit causes the frame filter to reject all control frames (frames with length/type field equal to 88-08). When cleared, error-free pause control frames are still interpreted (if enabled by FLCE) but are not delivered via DMA.

[1] Pass control frames.

[0] Do not pass control frames.

- **Flow control enable** (FLCE)

  When set, this bit enables interpretation of MAC control pause frames that are received without errors.

  [1] Flow control enabled.

  [0] Flow control not enabled.

- **FLC busy status** (FLCBUSY)

  Setting this bit triggers the MAC to send a control frame. The MAC automatically clears the FLCBUSY bit once the control frame has been transferred onto the physical medium. Writes to the flow control register are discarded if FLCBUSY is 1.

  [1] Initiate sending flow control frame.

  [0] No operation.

EMAC_VLAN1 and EMAC_VLAN2 Registers

The EMAC_VLAN1 register, shown in Figure 8-21, and the EMAC_VLAN2 register, shown in Figure 8-22, contain the tag fields used to identify VLAN frames. The MAC compares the 13th and 14th bytes of the incoming frame field to the values contained in these registers, so that the 13th frame byte is compared to the most significant byte of the registers and the 14th frame byte is compared to the least significant byte of the registers. If a match is found, the appropriate bit is set in the RX status register. The legal length of the frame is then increased from 1518 bytes to either 1522 bytes in the case of a VLAN1 match or 1538 bytes for a VLAN2 match.

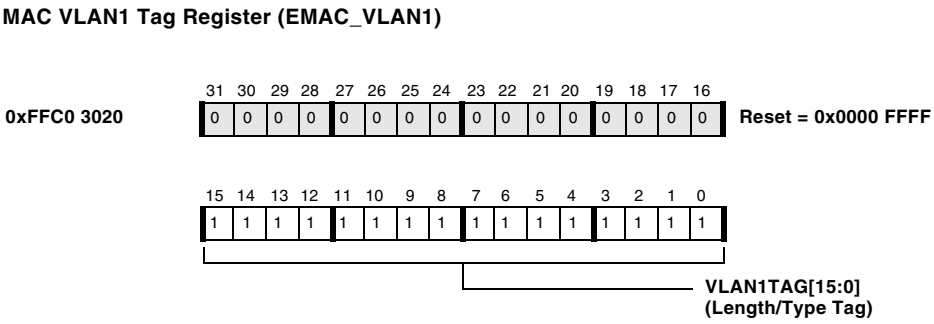


Figure 8-21. MAC VLAN1 Tag Register

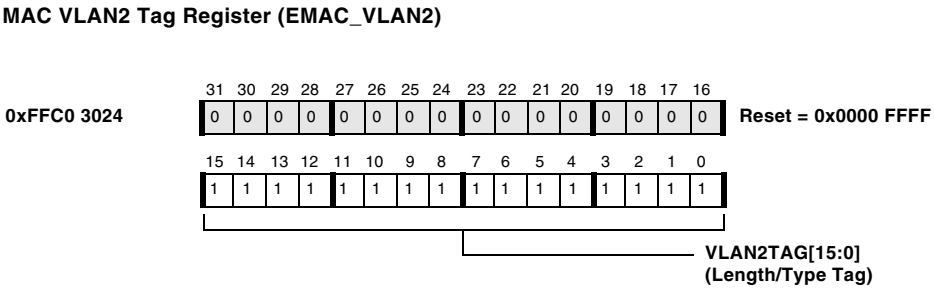


Figure 8-22. MAC VLAN2 Tag Register

EMAC_WKUP_CTL Register

The `EMAC_WKUP_CTL` register, shown in Figure 8-23, contains data pertaining to the MAC's remote wakeup status and capabilities. A write to the `EMAC_WKUP_CTL` register causes an update into the receive clock domain of all the wakeup filter registers. Changes to these other registers do not affect the operation of the MAC until the `EMAC_WKUP_CTL` register is written. For this reason, it is recommended that the wakeup filters be programmed by writing all of the other registers first, and writing the `EMAC_WKUP_CTL` register last.

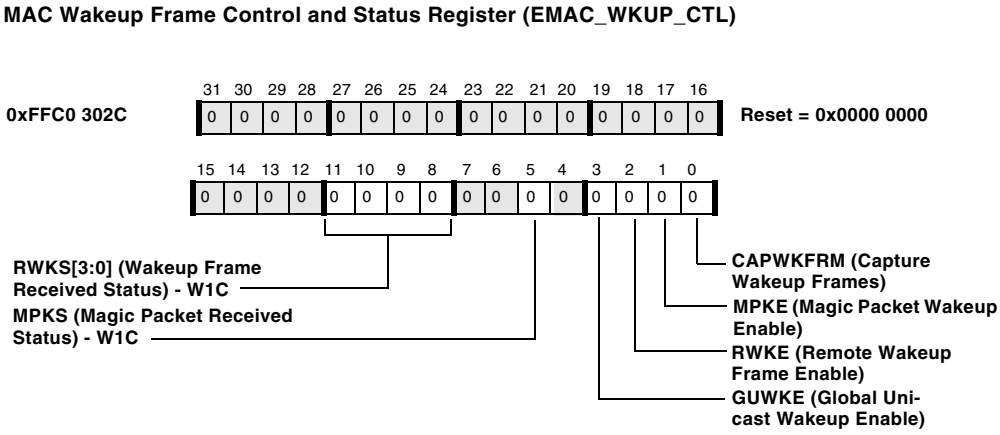


Figure 8-23. MAC Wakeup Frame Control and Status Register

Additional information for the `EMAC_WKUP_CTL` register bits includes:

- **Wakeup frame received status** (`RWKS`)

  These four frame status bits flag the receipt of wakeup frames corresponding to the respective wakeup frame filters.

- **Magic packet received status** (`MPKS`)

  This bit is set by the MAC when it receives the magic packet received wakeup call. The MAC then resumes operation in the normal powered-up mode.

  [1] Magic packet received.

  [0] Magic packet not received.

- **Global unicast wake enable** (GUWKE)

  When set, configures the MAC to wake up from the power-down mode on receipt of a global unicast frame. Such a frame has the MAC address [1:0] bits cleared.

  [1] Global unicast wake enabled.

  [0] Global unicast wake not enabled.

- **Remote wakeup frame enable** (RWKE)

  When set, this bit enables the remote wakeup frame power-down mode.

  [1] Remote wakeup frame enabled.

  [0] Remote wakeup frame not enabled.

- **Magic packet wakeup enable** (MPKE)

  When set, this bit enables the magic packet wakeup power-down mode.

  [1] Magic packet wakeup enabled.

  [0] Magic packet wakeup not enabled.

- **Capture wakeup frames** (CAPWKFRM)

  [1] RX frames are delivered via DMA while in power-down mode (when either MPKE or RWKE is set).

  [0] The RX DMA pathway is disabled when MPKE or RWKE is set.

EMAC_WKUP_FFMSK0, EMAC_WKUP_FFMSK1, EMAC_WKUP_FFMSK2, and EMAC_WKUP_FFMSK3 Registers

The EMAC_WKUP_FFMSK0, EMAC_WKUP_FFMSK1, EMAC_WKUP_FFMSK2, and EMAC_WKUP_FFMSK3 registers (see Figure 8-24 through Figure 8-27) are a part of the mechanism used to select which bytes in a received frame are used for CRC computation. Each bit in these registers functions as a byte enable. If a bit i is set, then the byte (offset + i) is used for CRC computation, where offset is contained in the EMAC_WKUP_FFOFF register.

For example, to identify a wakeup packet containing the byte sequence (0x80, 0x81, 0x82) in bytes 14, 15, and 17, the filter offset register should be set to 14 and the byte mask should be set to 0x000B. This byte mask has bits 0, 1, and 3 set, so that bytes 14+0, 14+1, and 14+3 are selected.

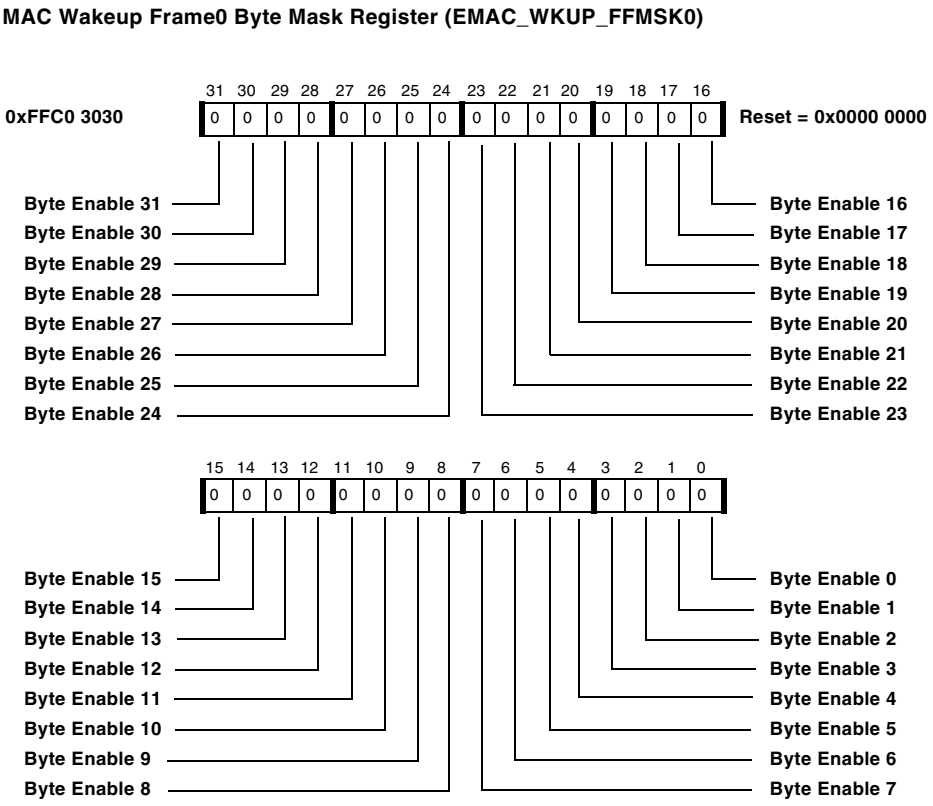


Figure 8-24. MAC Wakeup Frame0 Byte Mask Register

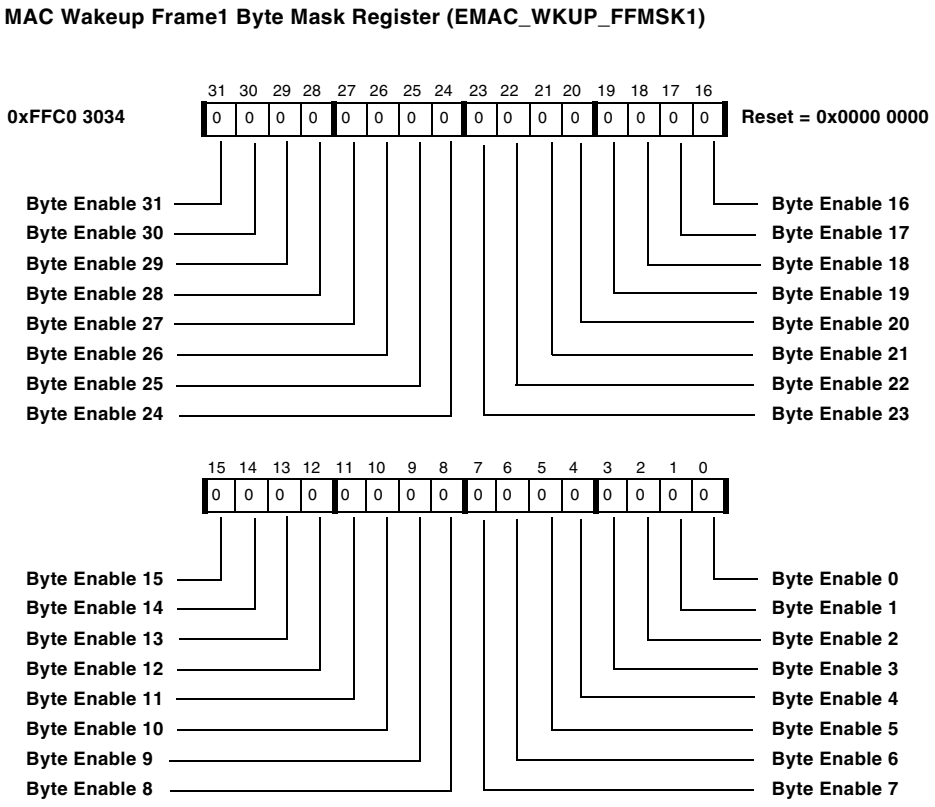


Figure 8-25. MAC Wakeup Frame1 Byte Mask Register

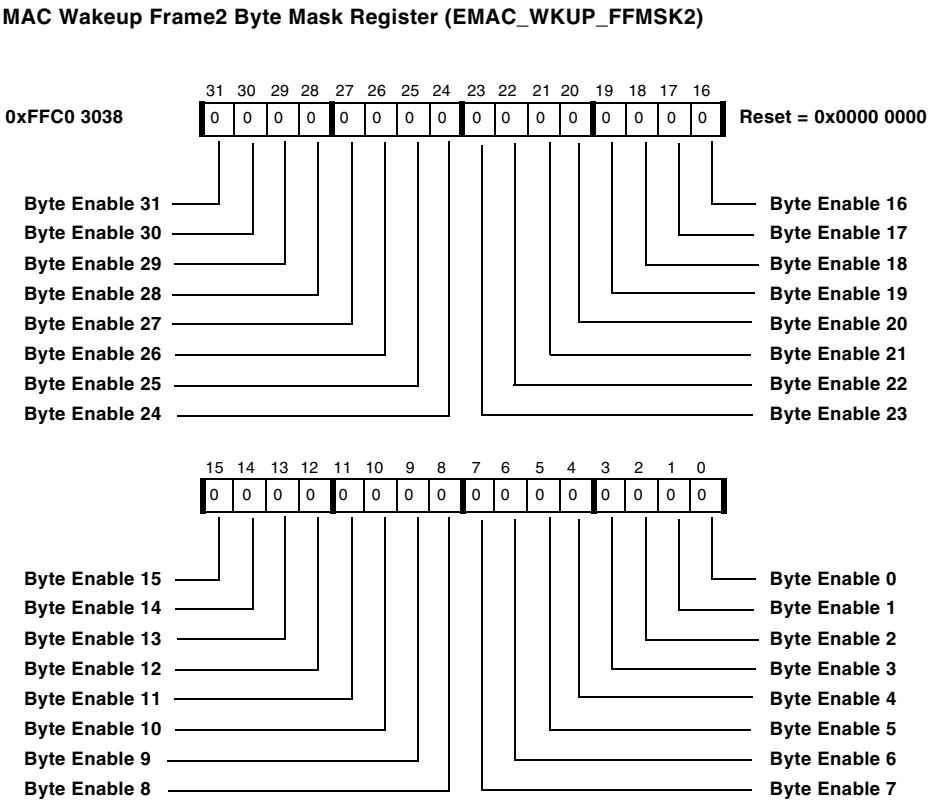


Figure 8-26. MAC Wakeup Frame2 Byte Mask Register

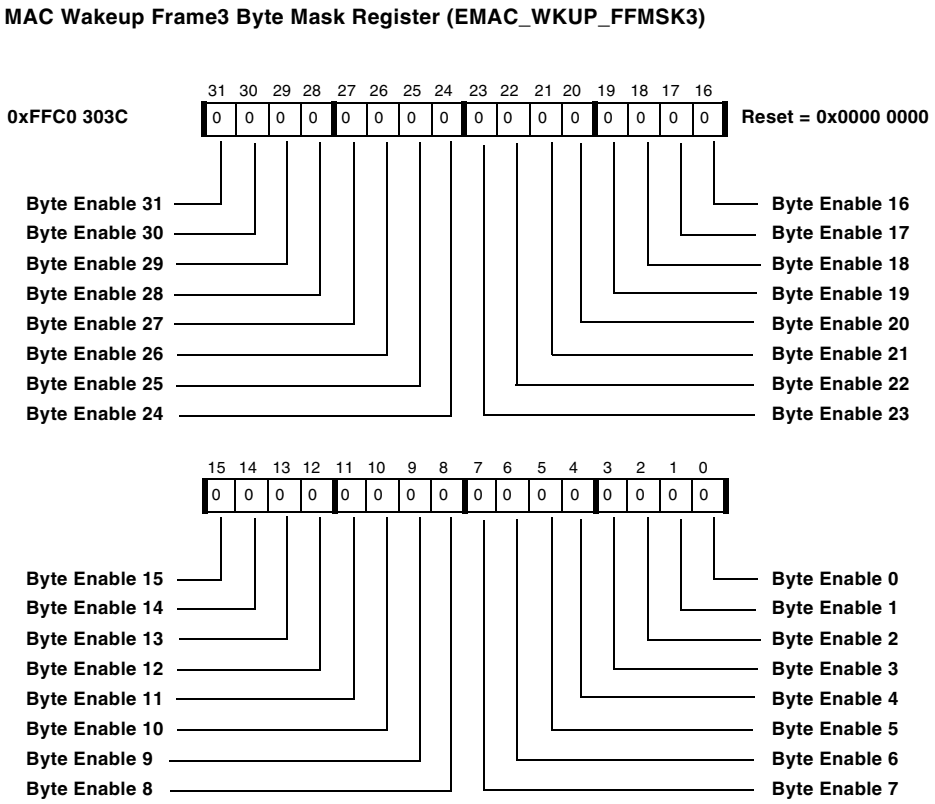


Figure 8-27. MAC Wakeup Frame3 Byte Mask Register

EMAC_WKUP_FFCMD Register

The EMAC_WKUP_FFCMD register, shown in Figure 8-28, regulates which of the four frame filter registers are enabled and if so, whether they are configured for unicast or multicast address filtering.

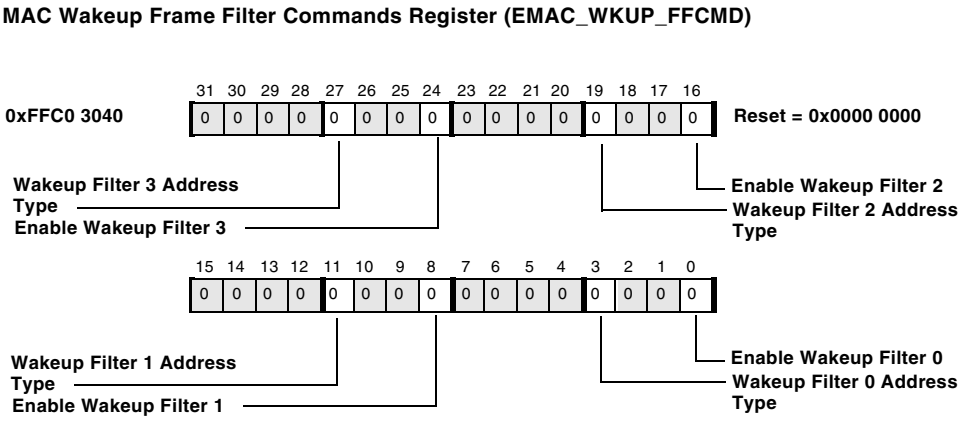


Figure 8-28. MAC Wakeup Frame Filter Commands Register

Additional information for the EMAC_WKUP_FFCMD register bits includes:

- **Wakeup filter 3 address type**

  [1] Multicast

  [0] Unicast

- **Enable wakeup filter 3**

  [1] Wakeup filter 3 enabled.

  [0] Wakeup filter 3 not enabled.

- **Wakeup filter 2 address type**

  [1] Multicast

  [0] Unicast

- **Enable wakeup filter 2**

  [1] Wakeup filter 2 enabled.

  [0] Wakeup filter 2 not enabled.

- **Wakeup filter 1 address type**

  [1] Multicast

  [0] Unicast

- **Enable wakeup filter 1**

  [1] Wakeup filter 1 enabled.

  [0] Wakeup filter 1 not enabled.

- **Wakeup filter 0 address type**

  [1] Multicast

  [0] Unicast

- **Enable wakeup filter 0**

  [1] Wakeup filter 0 enabled.

  [0] Wakeup filter 0 not enabled.

EMAC_WKUP_FFOFF Register

The EMAC_WKUP_FFOFF register, shown in Figure 8-29, contains the byte offsets for CRC computation to be performed on potential wakeup frames.

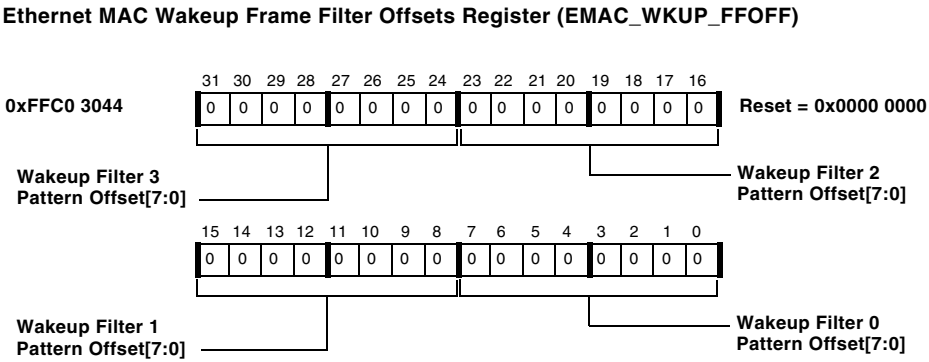


Figure 8-29. Ethernet MAC Wakeup Frame Filter Offsets Register

EMAC_WKUP_FFCRC0 and EMAC_WKUP_FFCRC1 Registers

The EMAC_WKUP_FFCRC0 register, shown in Figure 8-30, and the EMAC_WKUP_FFCRC1 register, shown in Figure 8-31, should be loaded with the results of the CRC computations for the relevant wakeup frame bytes. See "Remote Wake-up Filters" on page 8-35.

MAC Wakeup Frame Filter CRC0/1 Register (EMAC_WKUP_FFCRC0)

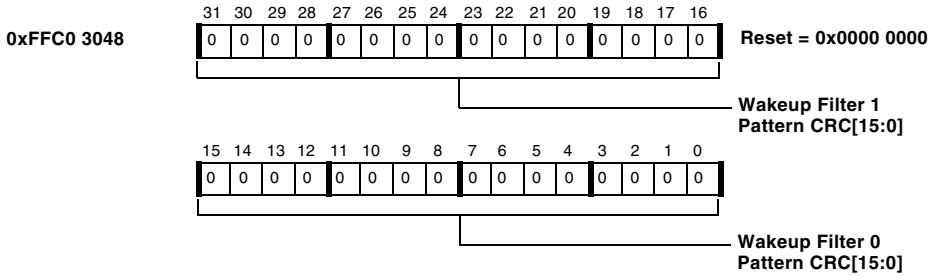


Figure 8-30. MAC Wakeup Frame Filter CRC0/1 Register

MAC Wakeup Frame Filter CRC2/3 Register (EMAC_WKUP_FFCRC1)

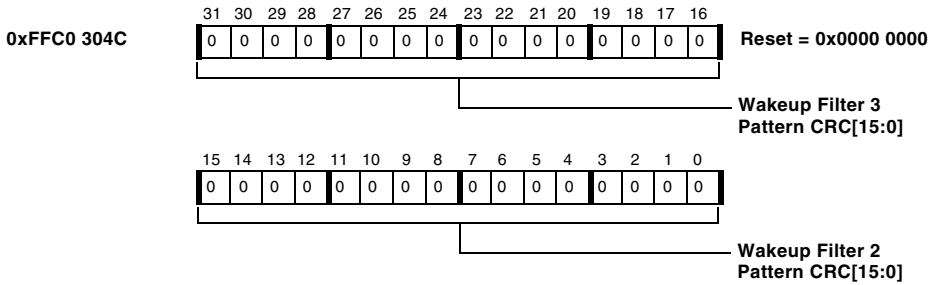


Figure 8-31. MAC Wakeup Frame Filter CRC2/3 Register

System Interface Register Group

The SIF block registers control and monitor the MAC's interactions with the Blackfin processor peripheral subsystem and the external PHY. The SIF block has several frame status registers whose bit descriptions can be found in "Ethernet MAC Frame Status Registers" on page 8-97.

EMAC_SYSCTL Register

The `EMAC_SYSCTL` register, shown in Figure 8-32, is used to set up MAC controls.

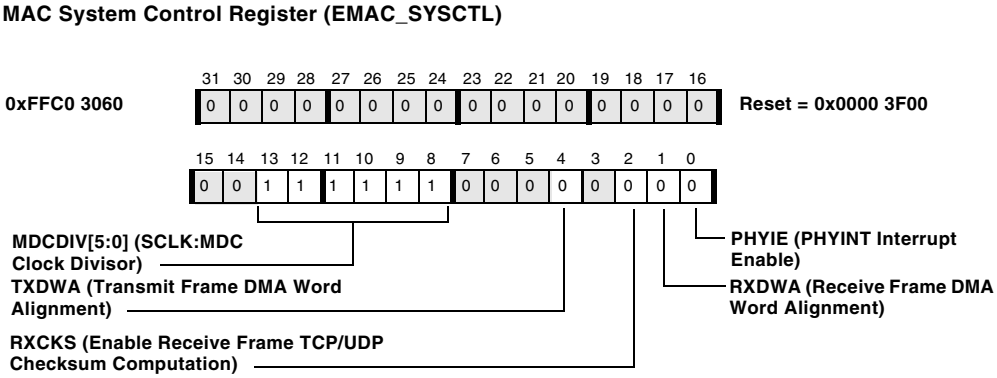


Figure 8-32. MAC System Control Register

Additional information for the EMAC_SYSCTL register bits includes:

- **SCLK:MDC clock divisor** (MDCDIV[5:0])

  This field contains the clock divisor that determines the ratio between the Blackfin system clock (SCLK) and the MAC data clock (MDC). The 6-bit ratio N determines the MDC rate as:

  MDC = SCLK/(2 x (N + 1)).

- **Transmit frame DMA word alignment** (TXDWA)

  This bit determines whether outgoing frame data is aligned on odd or even 16-bit boundaries in memory.

  [1] Even word alignment.

  [0] Odd word alignment.

- **Enable receive frame TCP/UDP checksum computation** (RXCKS)

  [1] TCP/UDP checksum computation on received frames enabled.

  [0] Receive frame TCP/UDP checksum computation not enabled.

- **Receive frame DMA word alignment** (RXDWA)

  This bit determines whether incoming frames are aligned on odd or even 16-bit boundaries in memory.

  [1] Odd word alignment.

  [0] Even word alignment.

- **PHYINT interrupt enable** (PHYIE)

  [1] PHYINT interrupt enabled.

  [0] PHYINT interrupt not enabled.

EMAC_SYSTAT Register

The EMAC_SYSTAT register, shown in Figure 8-33, contains a range of interrupt status bits that signal the occurrence of significant Ethernet events to the application. Detailed descriptions of the functionality can be found in the section entitled "Ethernet Event Interrupts" on page 8-38.

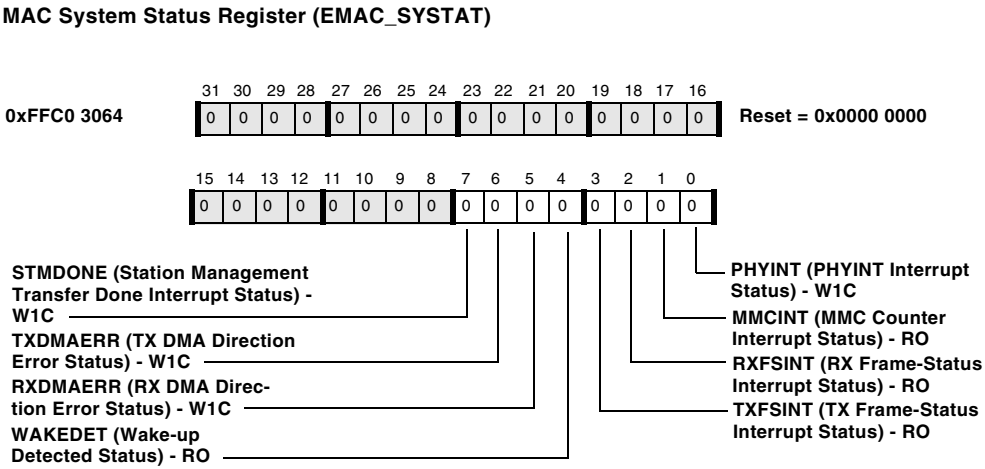


Figure 8-33. MAC System Status Register

Additional information for the EMAC_SYSTAT register bits includes:

- **Station management transfer done interrupt status** (STMDONE)

  This bit is set when a station management transfer on MDC/MDIO has completed, provided the STAIE interrupt enable control bit is set in the EMAC_STAADD register.

- **TX DMA direction error status** (TXDMAERR)

  This bit is set if a TX data or status DMA request is granted by the DMA channel with transfer in the wrong direction. Data should be memory-read, status should be memory-write. This interrupt is non-maskable in the Ethernet MAC.

- **RX DMA direction error status** (RXDMAERR)

  This bit it set if an RX data or status DMA request is granted by the DMA channel with transfer in the wrong (memory-read) direction. This interrupt is non-maskable in the Ethernet MAC.

- **Wakeup detected status** (WAKEDET)

  To clear this bit, write 1 to the wakeup control/status register.

  [1] Wakeup detected.

  [0] Wakeup not detected.

- **TX frame-status interrupt status** (TXFSINT)

  To clear this bit, write 1s to the EMAC_RX_STKY register bits.

  [1] TX frame-status interrupt has occurred.

  [0] TX frame-status interrupt has not occurred.

- **RX frame-status interrupt status** (RXFSINT)

  To clear this bit, write 1s to the EMAC_RX_STKY register bits.

  [1] RX frame-status interrupt has occurred.

  [0] RX frame-status interrupt has not occurred.

- **MMC counter interrupt status** (MMCINT)

  To clear this bit, write 1 to the EMAC_MMC_RIRQS or EMAC_MMC_TIRQS register.

  [1] MMC counter interrupt has occurred.

  [0] MMC counter interrupt has not occurred.

- **PHYINT interrupt status** (PHYINT)

  [1] PHYINT interrupt has occurred.

  [0] PHYINT interrupt has not occurred.

Ethernet MAC Frame Status Registers

The Ethernet MAC frame status registers keep track of the status of each frame received or transmitted, as well as the status of MMC interrupts.

EMAC_RX_STAT Register

The EMAC_RX_STAT register, shown in Figure 8-34, tells the status of the most recently completed receive frame, including type of error for cases where an error occurs. When the receive complete bit is set, exactly one of bits 13 through 20 is 1. Bits 13 through 20 indicate the receive status as defined in IEEE 802.3, section 4.3.2. In case of multiple errors, errors are prioritized in the order listed in Table 8-4 on page 8-20. Bits 18 and 19 identify frames which are not considered received by the station and also are not considered errors. (See section 4.1.2.1.2 and section 4.2.4.2.2 of IEEE 802.3.) Bit 20 identifies frames damaged within the MAC sublayer.

Note if the PB (pass bad frames) bit is 0, then delivery via DMA of frames with status bits 14 through 18 or 20 is cancelled. The DMA buffer is reused for the next frame. If the PR (promiscuous) bit is 0, then frames with bit 19 set are not delivered (the DMA is never initiated).

Ethernet MAC RX Current Frame Status Register (EMAC_RX_STAT)

All bits in this register are RO.

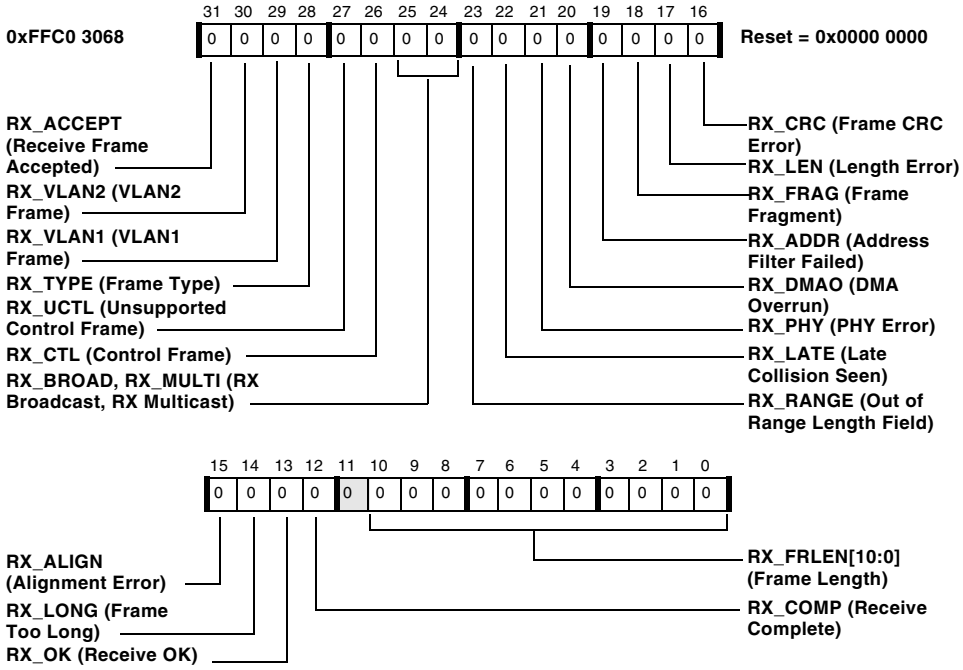


Figure 8-34. Ethernet MAC RX Current Frame Status Register

Additional information for the EMAC_RX_STAT register bits includes:

- **Receive frame accepted** (RX_ACCEPT)

  [1] The receive frame was accepted, based on the address filter result and the frame filtering modes in the EMAC_OPMODE register. Note that this does not imply a status of receiveOK. If the RA

(receive all) control bit is 0, then the only frames delivered by DMA are the frames whose receive frame accepted status bit is 1.

[0] Receive frame not accepted.

- **VLAN2 frame** (RX_VLAN2)

  [1] The frame is a valid tagged frame with a length/type field matching the VLAN2 tag register, and with status of receiveOK.

  [0] The frame does not meet those conditions.

- **VLAN1 frame** (RX_VLAN1)

  [1] The frame is a valid tagged frame with a length/type field matching the VLAN1 tag register, and with status of receiveOK.

  [0] The frame does not meet those conditions.

- **Frame type** (RX_TYPE)

  [1] The frame is a valid typed frame, with status of receiveOK and with a length/type field greater than or equal to 0x600.

  [0] The frame is not of that type.

- **Unsupported control frame** (RX_UCTL)

  [1] The frame is a valid MAC control frame (with status of receiveOK and with a length/type field equal to 802.3_MAC_Control, 88-08), but does not contain the pause opcode, or is not 64 bits in length, or is received in half-duplex mode.

  [0] The frame does not meet those conditions.

- **Control frame** (RX_CTL)

  [1] The frame is a valid MAC control frame in full duplex mode with status of receiveOK, with a length/type field equal to MAC_Control, 88-08, with length of 64 bytes, and with a MAC control opcode field equal to the pause opcode (00-01).

  [0] The frame does not meet those conditions.

- **RX broadcast, RX multicast** (RX_BROAD, RX_MULTI)

  [1 1] Illegal
  [1 0] Broadcast address
  [0 1] Group address
  [0 0] Unicast address

- **Out of range length field** (RX_RANGE)

  [1] The frame's length/type field was consistent with the length interpretation (<1536 = 0x600) but was greater than the maximum allowable frame size in bytes, as indicated by the frame too long bit).

  [0] The frame's length was not out of range.

- **Late collision seen** (RX_LATE)

  [1] A collision was detected after the first 64 bytes of the packet.

  [0] Late collision not detected.

- **PHY error** (RX_PHY)

  [1] RX_ER was asserted at some time during the frame. This condition always causes the FCS check to fail.

  [0] No PHY error.

- **DMA overrun** (RX_DMAO)

  [1] The received frame was truncated due to failure of the FIFO/DMA channel to continuously store data during DMA transfer to memory.

  [0] No DMA overrun.

- **Address filter failed** (RX_ADDR)

  [1] The destination address did not pass the address filters specified by the station MAC address, the multicast hash registers, and the filter modes in the operating modes register.

  [0] Address did not fail.

- **Frame fragment** (RX_FRAG)

  [1] Frame length was less than the minimum frame size (64 bytes).

  [0] Frame length was at least 64 bytes.

- **Length error** (RX_LEN)

  [1] The frame's length/type field does not match the length of received data and is consistent with the length interpretation (< 0x600), although the frame had no "frame too long" errors and had a valid FCS.

  [0] No frame length error.

- **Frame CRC error** (RX_CRC)

  [1] The frame failed FCS validation, but had neither a "frame too long" error nor a partial number of octets. Note if RX_ER is asserted by the PHY during frame reception, the FCS validation will fail.

  [0] No frame CRC error.

- **Alignment error** (RX_ALIGN)

  [1] The frame ended with a partial octet and failed RCS validation, but had no frame too long error.

  [0] No alignment error.

- **Frame too long** (RX_LONG)

  [1] The number of octets received is greater than the maximum Ethernet frame size. Maximum frame size is 1522 bytes for a frame whose length/type field matches the VLAN1 tag register, 1538 bytes for a frame whose length/type field matches the VLAN2 tag register, or 1518 for all other frames. The frame data delivered by DMA is truncated to 1556 (0x614) bytes in all cases.

  [0] Frame is not too long.

- **Receive OK** (RX_OK)

  [1] There was no receive error.

  [0] A receive error occurred.

- **Receive complete** (RX_COMP)

  This bit is cleared on reset and when the MAC RX is enabled (RE changes from 0 to 1). Frames that fail the address filter or the frame filter are not delivered by DMA, unless overridden by the RA (receive all) control bit. Note that in the RX frame status buffer written to memory by DMA, the receive complete bit is always 1. This bit acts as a semaphore, indicating that DMA of the frame has completed.

  [1] The first RX frame is complete.

  [0] The first RX frame is not yet complete.

- **Frame length** (RX_FRLEN)

  The number of bytes in the frame. If the ASTP bit is set, the pad and FCS are not included in the length.

EMAC_RX_STKY Register

The EMAC_RX_STKY register, shown in Figure 8-35, accumulates state across multiple frames, unless software clears it after every frame.

Ethernet MAC RX Sticky Frame Status Register (EMAC_RX_STKY)
All bits in this register are W1C.

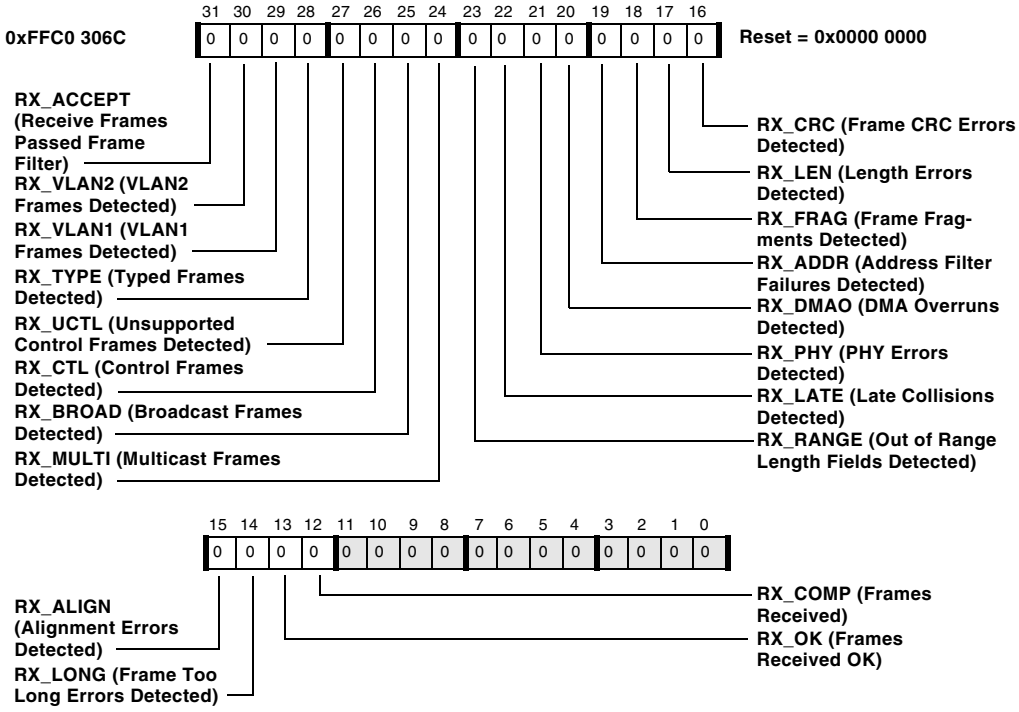


Figure 8-35. Ethernet MAC RX Sticky Frame Status Register

Additional information for the EMAC_RX_STKY register bits includes:

- **Receive frames passed frame filter** (RX_ACCEPT)

  [1] At least one receive frame passed the frame filter.
  [0] No receive frames passed the frame filter.

- **VLAN2 frames detected** (RX_VLAN2)

  [1] At least one VLAN2 frame was detected.
  [0] No VLAN2 frames were detected.

- **VLAN1 frames detected** (RX_VLAN1)

  [1] At least one VLAN1 frame was detected.
  [0] No VLAN1 frames were detected.

- **Typed frames detected** (RX_TYPE)

  [1] At least one typed frame was detected.
  [0] No typed frames were detected.

- **Unsupported control frames detected** (RX_UCTL)

  [1] At least one unsupported control frame was detected.
  [0] No unsupported control frames were detected.

- **Control frames detected** (RX_CTL)

  [1] At least one control frame was detected.
  [0] No control frames were detected.

- **Broadcast frames detected** (RX_BROAD)

  [1] At least one broadcast frame was detected.
  [0] No broadcast frames were detected.

- **Multicast frames detected** (RX_MULTI)

  [1] At least one multicast frame was detected.
  [0] No multicast frames were detected.

- **Out of range length fields detected** (RX_RANGE)

  [1] At least one out of range length field was detected.
  [0] No out of range length fields were detected.

- **Late collisions detected** (RX_LATE)

  [1] At least one collision was detected after the first 64 bytes of the packet.
  [0] No late collisions were detected.

- **PHY errors detected** (RX_PHY)

  [1] At least one PHY error was detected.
  [0] No PHY errors were detected.

- **DMA overruns detected** (RX_DMAO)

  [1] At least one DMA overrun was detected.
  [0] No DMA overruns were detected.

- **Address filter failures detected** (RX_ADDR)

  [1] At least one address filter failure was detected.
  [0] No address filter failures were detected.

- **Frame fragments detected** (RX_FRAG)

  [1] At least one frame fragment was detected.
  [0] No frame fragments were detected.

- **Length errors detected** (RX_LEN)

  [1] At least one length error was detected.
  [0] No length errors were detected.

- **Frame CRC errors detected** (RX_CRC)

  [1] At least one CRC error was detected.
  [0] No frame CRC errors were detected.

- **Alignment errors detected** (RX_ALIGN)

  [1] At least one alignment error was detected.
  [0] No alignment errors were detected.

- **Frame too long errors detected** (RX_LONG)

  [1] At least one frame too long error was detected.
  [0] No frame too long errors were detected.

- **Frames received OK** (RX_OK)

  This bit can be used to generate an interrupt on the next RX frame.

  [1] At least one frame has been received OK.
  [0] No good frames have been received.

- **Frames received** (RX_COMP)

  [1] At least one frame (good or bad) was received.
  [0] No frames were received.

EMAC_RX_IRQE Register

The `EMAC_RX_IRQE` register, shown in Figure 8-36, enables the frame status interrupts.

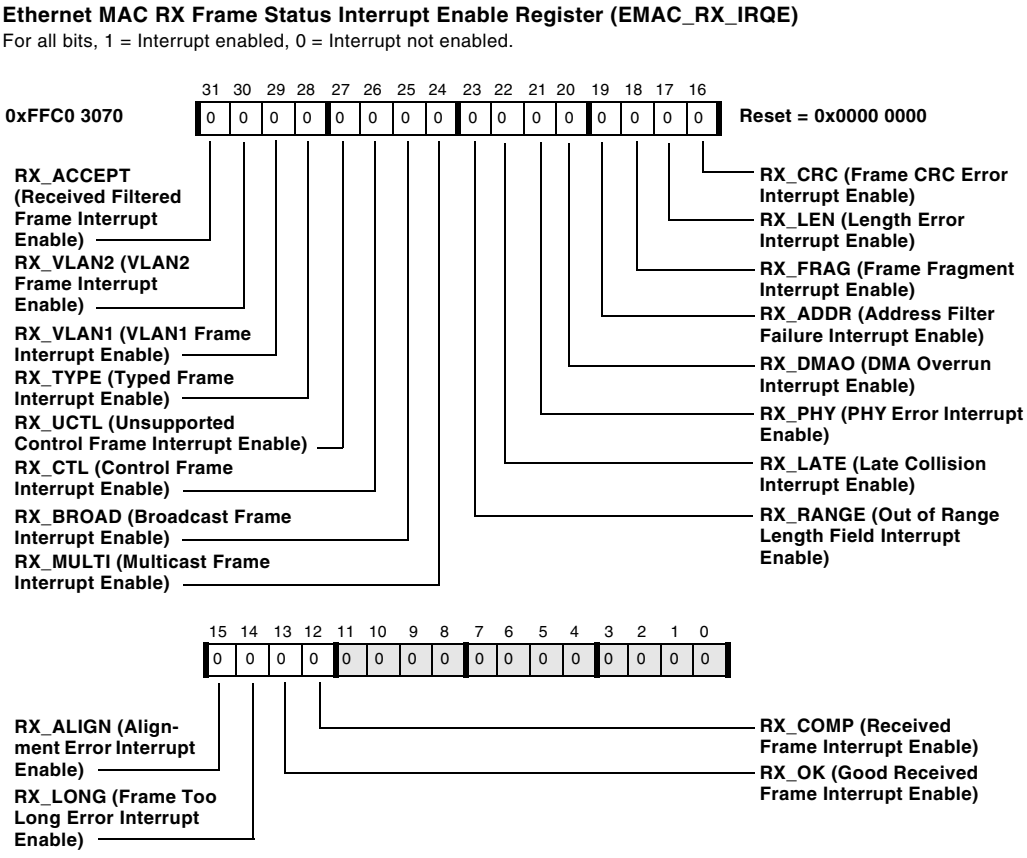


Figure 8-36. Ethernet MAC RX Frame Status Interrupt Enable Register

EMAC_TX_STAT Register

The EMAC_TX_STAT register, shown in Figure 8-37, tells the status of the most recently completed transmit frame, including type of error for cases where an error occurred. When the transmit complete bit is set, exactly one of bits 2, 3, 4, 13, or 14 is 1. Bits 1 through 3 indicate the transmit status as defined in IEEE 802.3, section 4.3.2.

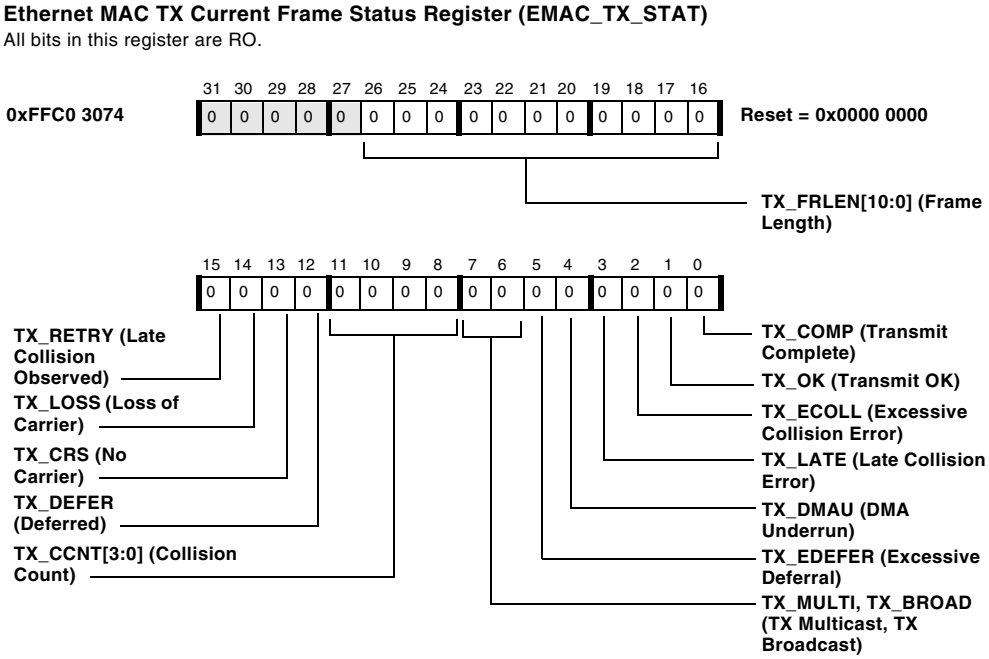


Figure 8-37. Ethernet MAC TX Current Frame Status Register

Additional information for the EMAC_TX_STAT register bits includes:

- **TX frame length** (TX_FRLEN)

  This field contains the length of the transmit frame in bytes.

- **Late collision observed** (TX_RETRY)

  [1] A late collision occurred, but the frame transmission was successful after retry.

  [0] No late collision occurred.

- **Loss of carrier** (TX_LOSS)

  [1] The carrier sense transitioned from asserted to deasserted at some time during the frame transmission. Half-duplex only.

  [0] No loss of carrier occurred.

- **No carrier** (TX_CRS)

  [1] Carrier sense (CRS) was not asserted at any time during frame transmission. Half-duplex only.

  [0] CRS was asserted.

- **Deferred** (TX_DEFER)

  [1] The transmission was deferred in half-duplex mode because the medium was initially occupied (CRS was asserted) at the time the frame was ready to transmit (after the initial frame data was transferred by DMA to the MAC). Note the deferred status bit should be expected to be 1 on frames that have been retried after early collisions, since the MAC can restart the frame immediately after a collision using data available in its local FIFO. Since the MAC does not need to wait for DMA, the frame data is typically ready for retransmission before TXEN and CRS have deasserted from the prior attempt. Half-duplex only.

  [0] Transmission not deferred.

- **Collision count** (TX_CCNT)

  This field contains the number of collisions that occurred during frame transmission.

- **TX broadcast, TX multicast** (TX_BROAD, TX_MULTI)

  [1 1] Illegal
  [1 0] Group address
  [0 1] Broadcast address
  [0 0] Unicast address

- **Excessive deferral** (TX_EDEFER)

  [1] The frame transmission was deferred for more than 24,288 bit times or 6072 TX clocks:

  ```
  MaxDeferTime = 2 x (MaxUntaggedFrameSize x 8) bits
  ```

  If the deferral check (DC) bit in the EMAC_OPMODE register is 1, frame transmission is aborted upon excessive deferral, and both the excessive deferral and excessive collision error status bits are set.

  [0] Excessive deferral did not occur.

- **DMA underrun** (TX_DMAU)

  [1] The frame transmission was interrupted by a failure of the FIFO/DMA channel to continuously supply frame data after the start of transmission on the MII/RMII.

  [0] No DMA underrun.

- **Late collision error** (TX_LATE)

  [1] Frame transmission failed because a collision occurred after the end of the collision window (512 bit times) and the LCRTE bit was clear, disabling frame transmission retry.

  [0] No late collision error.

- **Excessive collision error** (TX_ECOLL)

  [1] Frame transmission failed because too many (16) attempts were interrupted by collisions, or because the frame was deferred for more than the maximum deferral time while the deferral check (DC) control bit was set.

  [0] No excessive collision error.

- **Transmit OK** (TX_OK)

  [1] There was no transmit error.

  [0] A transmit error occurred.

- **Transmit complete** (TX_COMP)

  This bit is cleared on reset and when the MAC TX is enabled (TE changes from 0 to 1). In the TX DMA status buffer, this bit is always set to 1 on every status word written via DMA. This bit thus acts as a semaphore, indicating to software that processing of this descriptor pair has been completed.

  [1] The first TX frame is complete.

  [0] The first TX frame is not yet complete.

EMAC_TX_STKY Register

The EMAC_TX_STKY register, shown in Figure 8-38, accumulates state across multiple frames, unless software clears it after every frame.

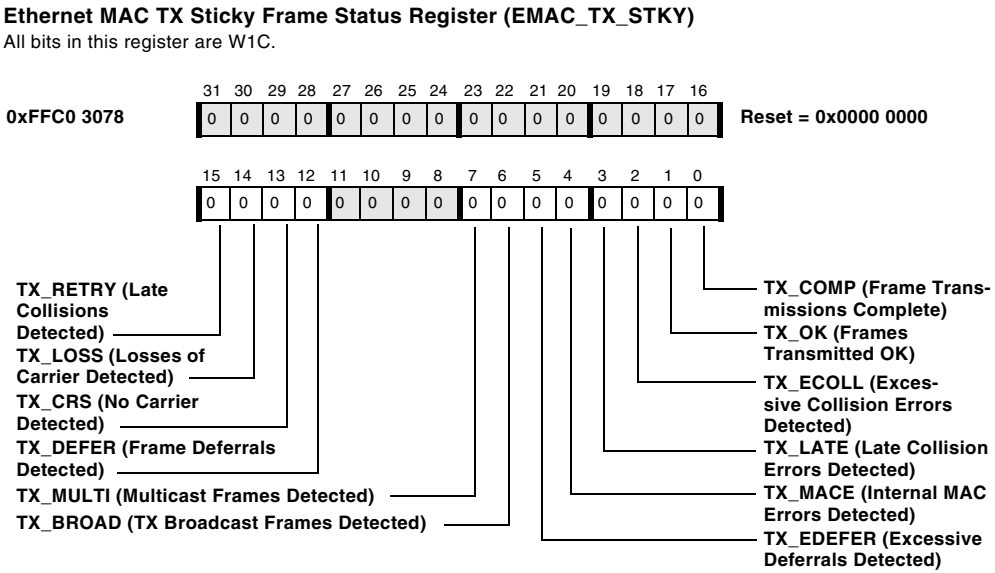


Figure 8-38. Ethernet MAC TX Sticky Frame Status Register

Additional information for the EMAC_TX_STKY register bits includes:

- **Late collisions detected** (TX_RETRY)

  [1] At least one late collision was detected on frames successfully transmitted after retry.

  [0] No late collisions were detected.

- **Losses of carrier detected** (TX_LOSS)

  [1] At least one loss of carrier was detected.

  [0] No losses of carrier were detected.

- **No carrier detected** (TX_CRS)

  [1] At least one occasion of no carrier was detected.
  [0] No instances of no carrier were detected.

- **Frame deferrals detected** (TX_DEFER)

  [1] At least one frame deferral was detected.
  [0] No frame deferrals were detected.

- **TX multicast frames detected** (TX_MULTI)

  [1] At least one multicast frame was detected.
  [0] No multicast frames were detected.

- **TX broadcast frames detected** (TX_BROAD)

  [1] At least one broadcast frame was detected.
  [0] No broadcast frames were detected.

- **Excessive deferrals detected** (TX_EDEFER)

  [1] At least one excessive deferral was detected.
  [0] No excessive deferrals were detected.

- **Internal MAC errors detected** (TX_MACE)

  [1] At least one internal MAC error was detected.
  [0] No internal MAC errors were detected.

- **Late collision errors detected** (TX_LATE)

  [1] At least one late collision error was detected.
  [0] No late collision errors were detected.

- **Excessive collision errors detected** (TX_ECOLL)

  [1] At least one excessive collision error detected.
  [0] No excessive collision errors were detected.

- **Frames transmitted OK** (TX_OK)

  This bit can be used to generate an interrupt at the completion of each TX frame.

  [1] At least one frame has been transmitted OK.
  [0] No good frames have been transmitted.

- **Frame transmissions complete** (TX_COMP)

  [1] At least one frame was transmitted.
  [0] No frames have been transmitted.

EMAC_TX_IRQE Register

The EMAC_TX_IRQE register, shown in Figure 8-39, is used to enable TX frame status interrupts.

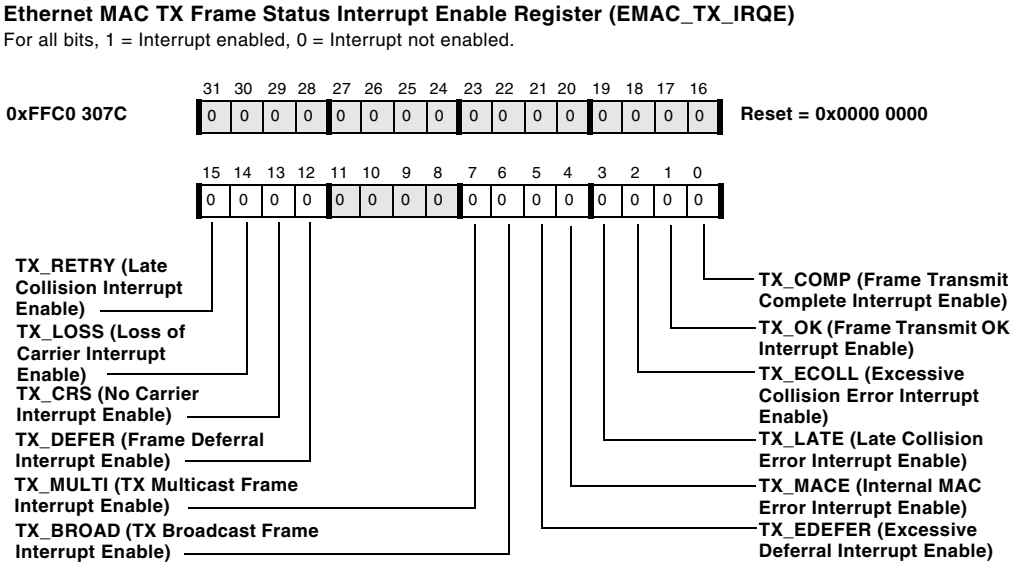


Figure 8-39. Ethernet MAC TX Frame Status Interrupt Enable Register

EMAC_MMC_RIRQS Register

The EMAC_MMC_RIRQS register, shown in Figure 8-40, indicates which of the receive MAC management counters have incremented past one-half of maximum range. Each bit is set from 0 to 1 when the corresponding counter increments from a value less than 0x8000 0000 to a value greater than or equal to 0x8000 0000 (regardless of the state of the EMAC_MMC_RIRQE interrupt enable register). Bits in this register are cleared by writing a 1; writing zero has no effect. For more information, see "MAC Management Counters" on page 8-43.

Ethernet MAC MMC RX Interrupt Status Register (EMAC_MMC_RIRQS)

All bits are W1C. For all bits, 1 = Interrupt occurred, 0 = Interrupt did not occur.

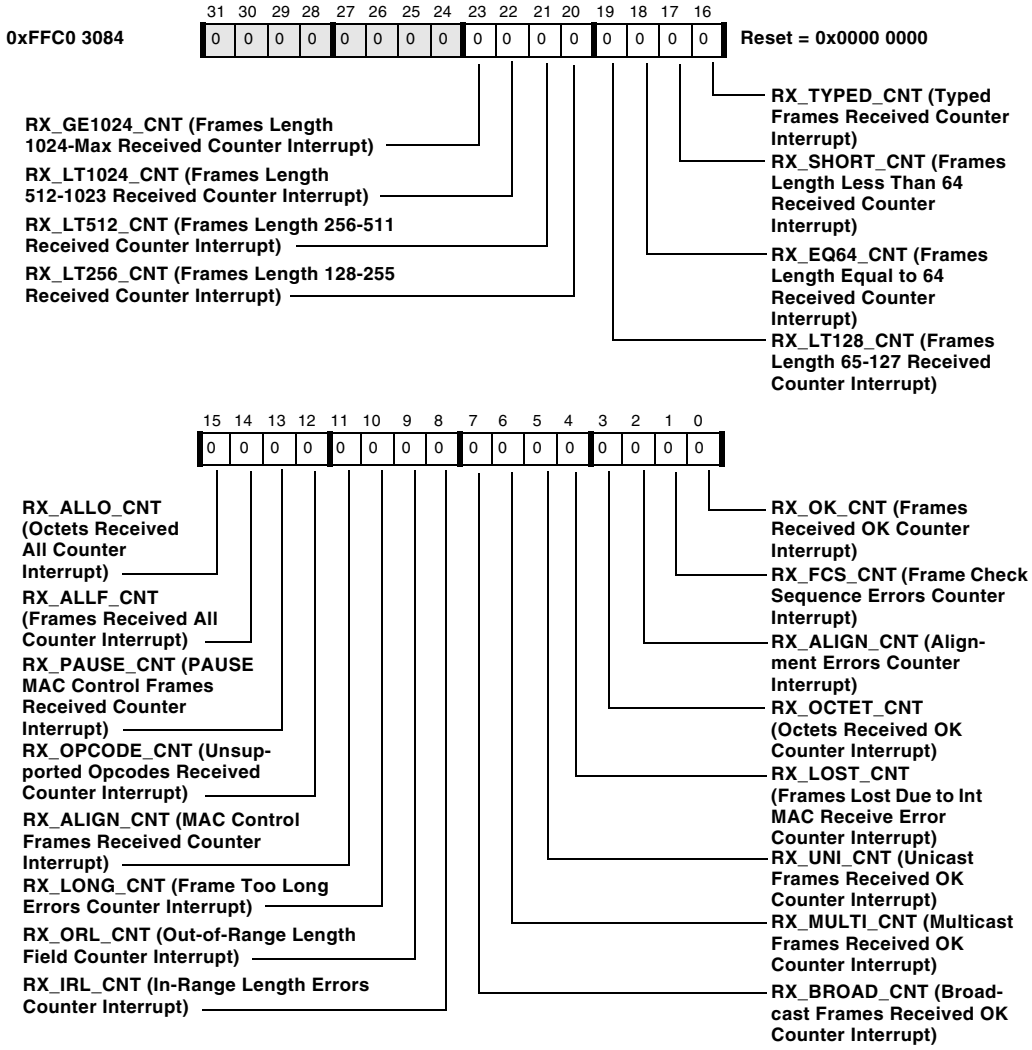


Figure 8-40. Ethernet MAC MMC RX Interrupt Status Register

EMAC_MMC_RIRQE Register

The EMAC_MMC_RIRQE register, shown in Figure 8-41, indicates which of the receive MAC management counters are enabled to signal an MMCINT interrupt when they increment past one-half of maximum range.

If a given counter's interrupt is not enabled, and that counter passes 0x8000 0000, then the counter's interrupt status bit is set to 1 but this does not cause the MMCINT interrupt to be signalled. If the corresponding interrupt enable bit is later written to 1, the MMCINT Ethernet event interrupt is signalled immediately.

Ethernet MAC MMC RX Interrupt Enable Register (EMAC_MMC_RIRQE)

For all bits, 1 = Interrupt enabled, 0 = Interrupt not enabled.

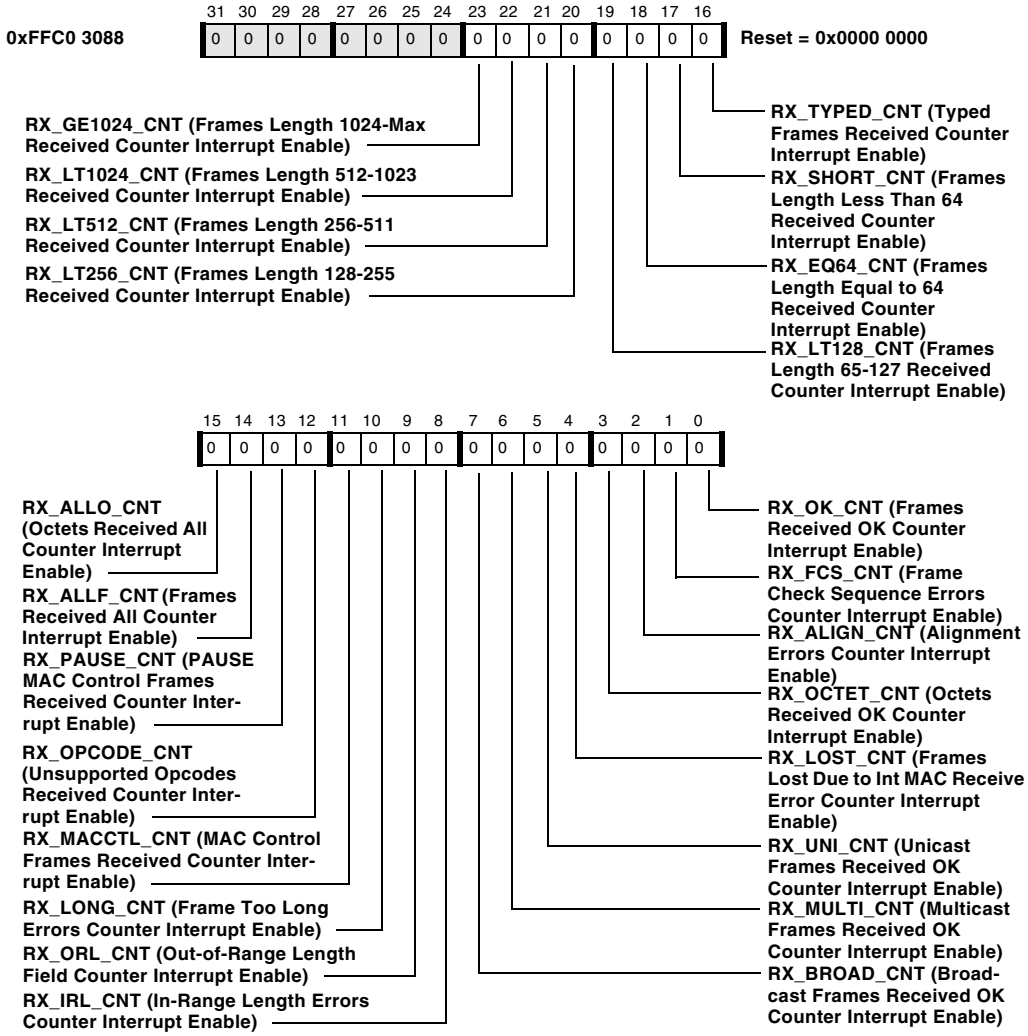


Figure 8-41. Ethernet MAC MMC RX Interrupt Enable Register

EMAC_MMC_TIRQS Register

The EMAC_MMC_TIRQS register, shown in Figure 8-42, indicates which of the transmit MAC management counters have incremented past one-half of maximum range. Each bit is set from 0 to 1 when the corresponding counter increments from a value less than 0x8000 0000 to a value greater than or equal to 0x8000 0000 (regardless of the state of the EMAC_MMC_TIRQE interrupt enable register). Bits in this register are cleared by writing a 1; writing zero has no effect. For more information, see "MAC Management Counters" on page 8-43.

Ethernet MAC MMC TX Interrupt Status Register (EMAC_MMC_TIRQS)

All bits are W1C. For all bits, 1 = Interrupt occurred, 0 = Interrupt did not occur.

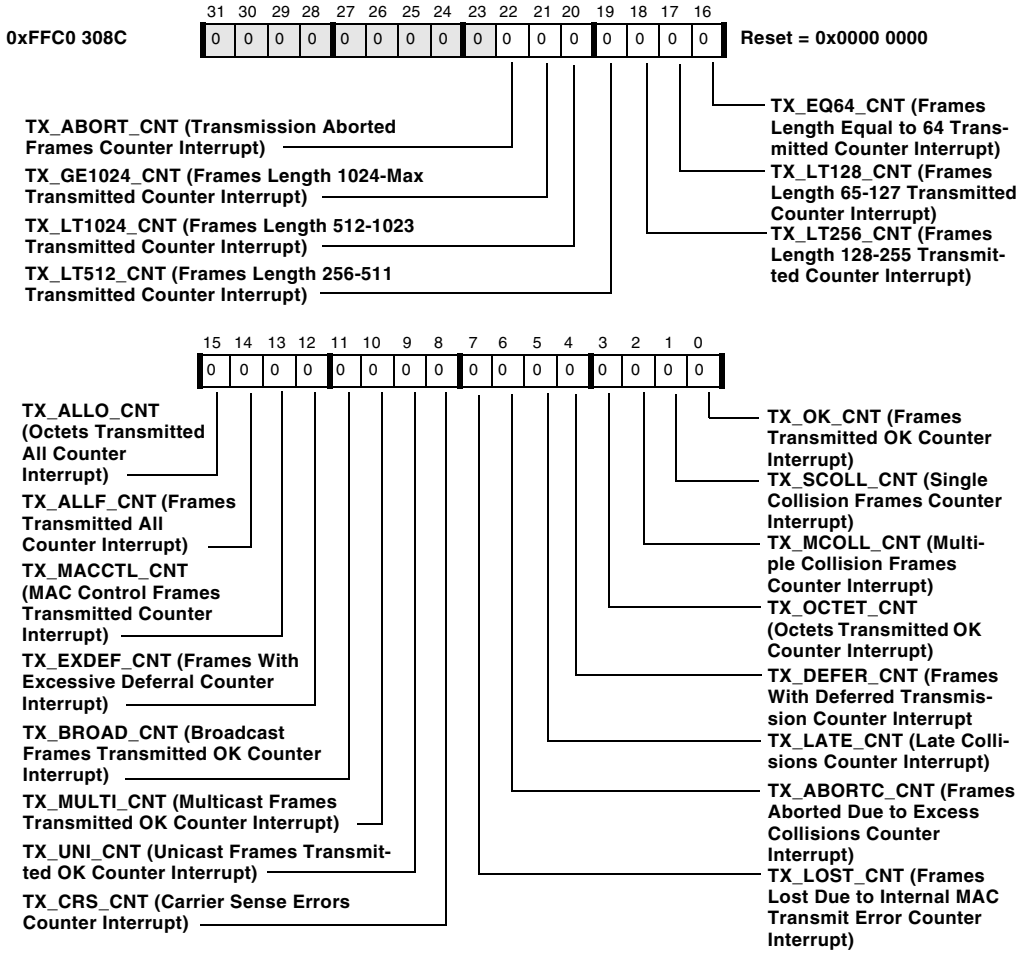


Figure 8-42. Ethernet MAC MMC TX Interrupt Status Register

EMAC_MMC_TIRQE Register

The EMAC_MMC_TIRQE register, shown in Figure 8-43, indicates which of the transmit MAC management counters are enabled to signal an MMCINT interrupt when they increment past one-half of maximum range.

If a given counter's interrupt is not enabled, and that counter passes 0x8000 0000, then the counter's interrupt status bit is set to 1 but this does not cause the MMCINT interrupt to be signalled. If the corresponding interrupt enable bit is later written to 1, the MMCINT Ethernet event interrupt is signalled immediately.

Ethernet MAC MMC TX Interrupt Enable Register (EMAC_MMC_TIRQE)

For all bits, 1 = Interrupt enabled, 0 = Interrupt not enabled.

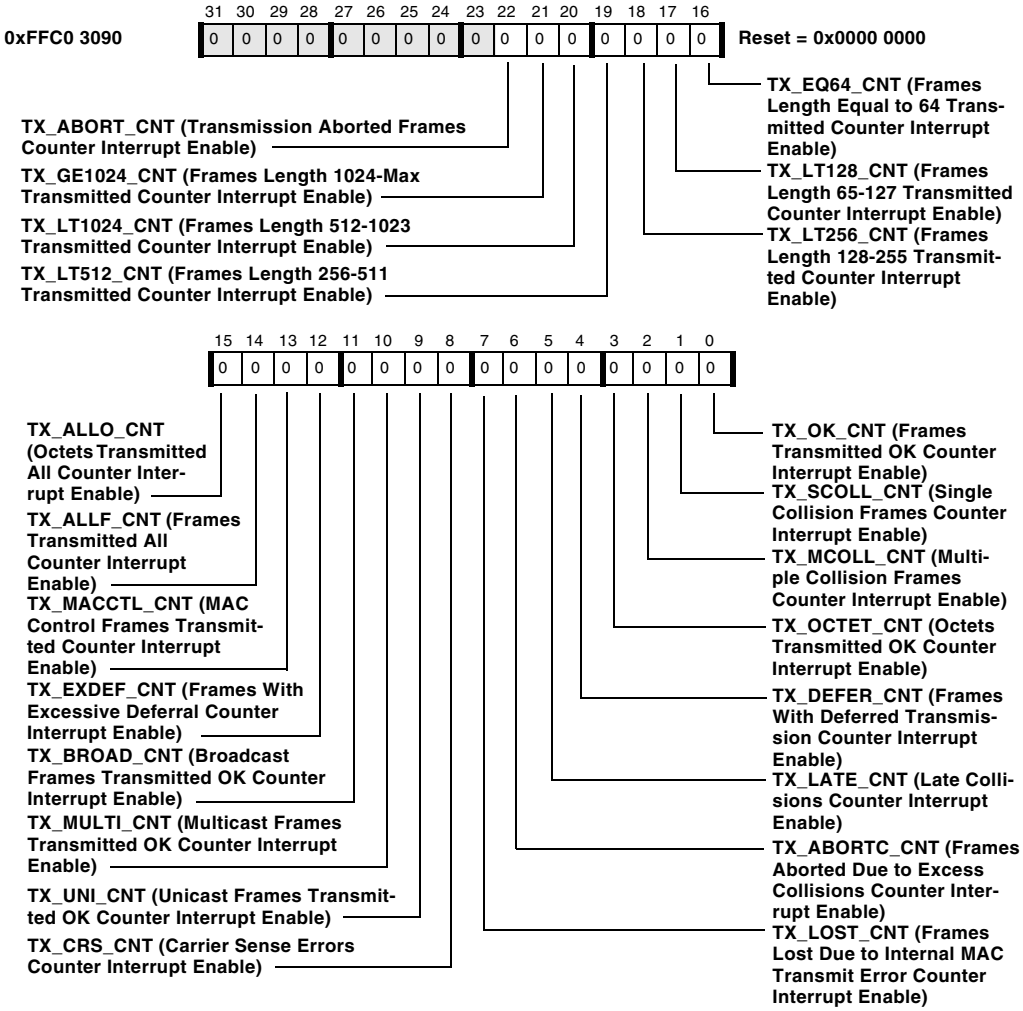


Figure 8-43. Ethernet MAC MMC TX Interrupt Enable Register

MAC Management Counter Registers

The MAC Management Counter (MMC) block register group consists of a number of 32-bit unsigned counter registers that gather statistical data regarding the operation of the MAC. The MAC management counter registers update automatically at the completion of frame transmit and receive, whenever the `MMCE` bit in the MMC control register is set. Counters contain a 32-bit unsigned value, and may be configured to saturate at 0xFFFF FFFF (`CROLL = 0`) or to wrap around to zero (`CROLL = 1`). Counters cannot be written directly, but can be collectively reset to zero by writing 1 to the `RSTC` bit, or they can be programmed for clear-on-read behavior by setting `CCOR` to 1.The reset value for all MMC registers is 0x0000 0000. See Table 8-10 on page 8-54 for more information.

Each of these counters can be set up to generate interrupts when they reach half of the maximum unsigned 32-bit value. This functionality is described in detail in the section entitled "Ethernet Event Interrupts" on page 8-38.

EMAC_MMC_CTL Register

The EMAC_MMC_CTL register, shown in Figure 8-44, is used to globally configure all MMC counter registers.

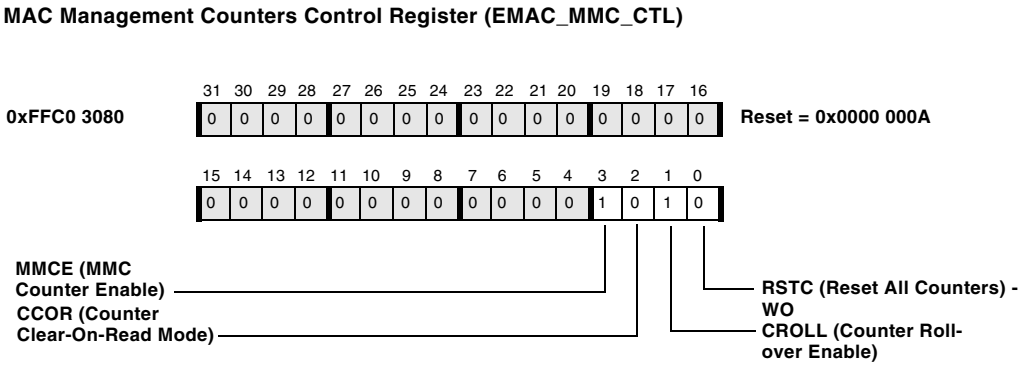


Figure 8-44. MAC Management Counters Control Register

Additional information for the EMAC_MMC_CTL register bits includes:

- **MMC counter enable** (MMCE)

  Setting this bit turns on all the MMC counters, which update on every frame transmission or reception.

  [1] MMC counters are enabled.

  [0] MMC counters are not enabled. Counters retain their values but are not updated.

- **Counter clear-on-read mode** (CCOR)

  [1] Counters are in clear-on-read mode. The contents of each counter is reset each time it is read by the application.

  [0] Counters are not in clear-on-read mode. Reads do not affect counter contents.

- **Counter rollover enable** (CROLL)

  [1] Counter rollover is enabled. This causes all MMC counters to wrap around to zero when the count exceeds the maximum 32-bit value of 0xFFFF FFFF.

  [0] Counter rollover is not enabled. All MMC registers saturate upon reaching 0xFFFF FFFF.

- **Reset all counters** (RSTC)

  Writing a 1 to this bit at any time globally resets all MMC counters.

  [1] Globally clear all MMC counters.

  [0] Do not reset all counters.

Programming Examples

This section gives a general overview of the functionality of an Ethernet MAC driver. All necessary steps for reproducing and understanding this interface are explained with code listings and accompanying text. These code listings are similar to the driver model supported by VisualDSP++ and are mainly written in C. Data transfers over the MAC with DMA are explained in Figure 8-5 on page 8-11 and Figure 8-7 on page 8-24, which show receive and transmit DMA operations. Please examine these figures carefully—the code listings reproduce this kind of “linked list” in the form

of C structures. Also provided are code listings that describe accessing an external PHY via the station management (MIM) block. All macros which are not explained in this section can be found in the `cdefBF537.h` and `defBF537.h` header files of VisualDSP++.

The code examples in this section (Listing 8-1 through Listing 8-9) show basic functions and structures. The management counter register and the interrupt settings are advanced functions and are not covered here. There are many counter registers which are accessible by polling of the appropriate register or using interrupt service routines. The `EMAC_SYSCTL` and `EMAC_SYSTAT` register should be used to configure the Ethernet MAC interrupts capabilities. See Figure 8-12 on page 8-39 for a detailed description of the MAC interrupts.

Ethernet Structures

Listing 8-1. Type Definition

```
// type definitions
typedef unsigned long  int    u32;
typedef unsigned short int    u16;
typedef unsigned        char  u8;
typedef volatile u32          reg32;
typedef volatile u16          reg16;
```

The type definitions are placed here to help with reading of the following code.

Listing 8-2. DMA Configuration

```
typedef struct ADI_DMA_CONFIG_REG {
  u16 b_DMAEN:1;     /* 0     Enabled */
  u16 b_WNR:1;        /* 1     Direction */
  u16 b_WDSIZE:2;     /* 2:3   Transfer word size */
```

```
  u16 b_DMA2D:1;       /* 4     DMA mode */
  u16 b_RESTART:1;     /* 5     Retain FIFO */
  u16 b_DI_SEL:1;      /* 6     Data interrupt timing select */
  u16 b_DI_EN:1;       /* 7     Data interrupt enabled */
  u16 b_NDSIZE:4;      /* 8:11  Flex descriptor size */
  u16 b_FLOW:3;        /* 12:14 Flow */
} ADI_DMA_CONFIG_REG;
```

A convenient way to handle the DMA properties in a “linked list” is to use structures, because each set should be assigned to the appropriate DMA descriptor. Listing 8-3 shows a structure used to manage DMA descriptors. Before jumping to the next descriptor, like 1A-1B-2A-2B-1C in Figure 8-5 on page 8-11 and Figure 8-7 on page 8-24, the structure ADI_DMA_CONFIG_REG immediately loads to the DMA register before starting its DMA transfer.

Listing 8-3. DMA Descriptor

```
typedef struct dma_descriptor {
  struct dma_descriptor*  NEXT_DESC_PTR;
  u32                     START_ADDR;
  ADI_DMA_CONFIG_REG      CONFIG;
} DMA_DESCRIPTOR;
```

The structure shown in Listing 8-3 shows how it is possible to create a “linked list” of DMAs. The START_ADDR points to the data and the ADI_DMA_CONFIG_REG structure (shown in Listing 8-2) holds all the necessary settings.

Structures like these are convenient for handling Ethernet streams, because they allow the programmer to simply call members of the structure instead of extracting meaningful items through array offsets. This structure, shown in Listing 8-4, is mirrored in the Ethernet MAC header with additional NoBytes.

Listing 8-4. Ethernet Frame Buffer

```
typedef struct adi_ether_frame_buffer {
  u16   NoBytes;  /* the no. of following bytes */
  u8    Dest[6];  /* destination MAC address */
  u8    Srce[6];  /* source MAC address */
  u16   LTfield;  /* length/type field */
  u8    Data[0];  /* payload bytes */
} ADI_ETHER_FRAME_BUFFER;
```

The ADI_ETHER_BUFFER structure in Listing 8-5, Top Level Structure, covers all the above structures and shows the general framework as described in Figure 8-5 on page 8-11 and Figure 8-7 on page 8-24. The two Dma[2] structures are needed for descriptors 1A,1B and 2A,2B. The pointer *frmData represents the payload of the frame, which has a specific number of bytes (as dictated by the NoBytes structure member). This is relevant only in transmit mode—in receive mode the driver will not touch this NoBytes variable. To ease programming by keeping the transmit and receive structures the same, the MAC can pad the first 16-bit word (that is, the data corresponding to the NoBytes structure member) with zeros if the RXDWA bit in EMAC_SYSCTL is 1. The *pNext and *pPrev pointers are necessary for creating a "linked list." The IPHdrChksum and IPPayloadChksum are available in case the Ethernet MAC is set to calculate this. See the RXCKS bit in the EMAC_SYSCTL register (shown in Figure 8-32 on page 8-93). These two variables are relevant only in receive mode of the Ethernet MAC. The StatusWord variable holds the EMAC_RX_STAT register value in receive mode and holds the EMAC_TX_STAT register value in transmit mode.

Listing 8-5. Top Level Structure

```
typedef struct adi_ether_buffer {
  DMA_DESCRIPTOR  Dma[2];  /* first for the frame, second for the
status */
  ADI_ETHER_FRAME_BUFFER  *FrmData;  /* pointer to data */
```

```
  struct adi_ether_buffer  *pNext;  /* next buffer */
  struct adi_ether_buffer  *pPrev;  /* prev buffer */
  u16  IPHdrChksum;  /* the IP header checksum */
  u16  IPPayloadChksum;  /* the IP header and payload checksum */
  u32  StatusWord;  /* the frame status word */
} ADI_ETHER_BUFFER;
```

MAC Address Setup

Write EMAC_ADDRLO and EMAC_ADDRHI in the initialization routine of the Ethernet MAC, as shown in Listing 8-6. The Ethernet MAC address is a unique number and may not be used twice. See the IEEE Std. 802.3-2002 specification for further information.

Listing 8-6. MAC Address Setup

```
// MAC address
u8  SrcAddr[6] = {0x5A,0xD4,0x9A,0x48,0xDE,0xAC};

//function
void SetupMacAddr(u8 *MACaddr)
{
  *pEMAC_ADDRLO = *(u32 *)&MACaddr[0];
  *pEMAC_ADDRHI = *(u16 *)&MACaddr[4];
}

//function call
SetupMacAddr(SrcAddr);
```

PHY Control Routines

The EMAC_STAAD register provides the option of either polling the STABUSY bit or getting an interrupt during each MIM block access. The function in Listing 8-7 polls the STABUSY bit and should be placed after each read or write command to the PHY register.

Listing 8-7. Poll MIM Block

```
//
/* Wait until the previous MDC/MDIO transaction has completed */
//
void PollMdcDone(void)
{
  /* poll the STABUSY bit */
  while(*pEMAC_STAADD & STABUSY)
}
```

Shown in Listing 8-8, the SET_PHYAD and SET_REGAD macros shift the PHY-Addr and RegAddr values to the appropriate field within the EMAC_STAADD register. The other macros STAOP, STAIE, and STABUSY, also set bits in the STAADD register. Use of the STAOP macro controls the read and write transfer of the MIM block.

Listing 8-8. Write Access to the PHY

```
//
/* Write an off-chip register in a PHY through the MDC/MDIO port
*/
//
void WrPHYReg(u16 PHYAddr, u16 RegAddr, u16 Data)
{
  PollMdcDone();
  *pEMAC_STADAT = Data;
  *pEMAC_STAADD = SET_PHYAD(PHYAddr) | \
```

```
                    SET_REGAD(RegAddr) |\
                    STAOP | STABUSY;
}
```

The data in the STADAT register is immediately shifted out after a write to the STAADD register. See Figure 8-4 on page 8-9.

The function in Listing 8-9 shows how PHY data is read over the MIM function block of the MAC. First, the STABUSY bit of the EMAC_STAADD will be polled until no other function is using the MIM block. The PHY address and register address is sent over the MIM block. Then, the STABUSY bit is polled again, before the data is finally read through the EMAC_STADAT register.

Listing 8-9. Read Access to the PHY

```
//
/* Read an off-chip register in a PHY through the MDC/MDIO port
*/
//
u16  RdPHYReg(u16 PHYAddr, u16 RegAddr)
{
  u16 Data;
  PollMdcDone();

  *pEMAC_STAADD =  SET_PHYAD(PHYAddr) |\
                   SET_REGAD(RegAddr) |\
                   STABUSY;

  PollMdcDone();
  Data = (u16)*pEMAC_STADAT;

  return Data;
}
```

A complete PHY initialization also requires the initialization of the station management clock, which is described in detail in the section "MII Station Management" on page 8-48. The three PHY functions included in this section (write, read, and poll) and the initialization routine of the station management clock are the minimum requirements for setup and control of any PHYs.

9 CAN MODULE

This chapter describes the Controller Area Network (CAN) module. Following an overview and a list of key features is a description of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples. Familiarity with the CAN standard is assumed. Refer to Version 2.0 of *CAN Specification* from Robert Bosch GmbH.

Overview

Key features of the CAN module are:

- Conforms to the CAN 2.0B (active) standard
- Supports both standard (11-bit) and extended (29-bit) identifiers
- Supports data rates of up to 1Mbit/s
- 32 mailboxes (8 transmit, 8 receive, 16 configurable)
- Dedicated acceptance mask for each mailbox
- Data filtering (first 2 bytes) can be used for acceptance filtering (DeviceNet™ mode)
- Error status and warning registers
- Universal counter module
- Readable receive and transmit pin values

The CAN module is a low bit rate serial interface intended for use in applications where bit rates are typically up to 1Mbit/s. The CAN protocol incorporates a data CRC check, message error tracking and fault node confinement as means to improve network reliability to the level required for control applications.

Interface Overview

The interface to the CAN bus is a simple two-wire line. See Figure 9-1 for a symbolic representation of the CAN transceiver interconnection, and Figure 9-2 for a block diagram. The Blackfin processor's CANTX output and CANRX input pins are connected to an external CAN transceiver's TX and RX pins (respectively). The CANTX and CANRX pins operate with TTL levels and are appropriate for operation with CAN bus transceivers according to ISO/DIS 11898.

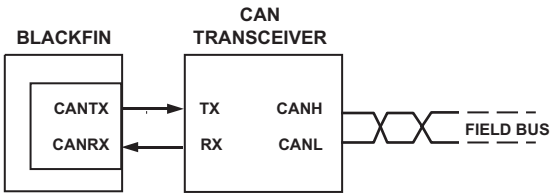


Figure 9-1. Representation of CAN Transceiver Interconnection

The CANRX and CANTX signals are multiplexed with the secondary data signals of SPORT0. To enable CAN functionality on the PJ4 and PJ5 pins, the PJCE bit field in the PORT_MUX register must be set to 01. CAN data is defined to be either *dominant* (logic 0) or *recessive* (logic 1). The default state of the CANTX output is recessive. Because the CANTX pin is multiplexed with a SPORT transmit signal pin, the output state may be low if the SPORT is selected instead of the CAN, as is the default case after reset.

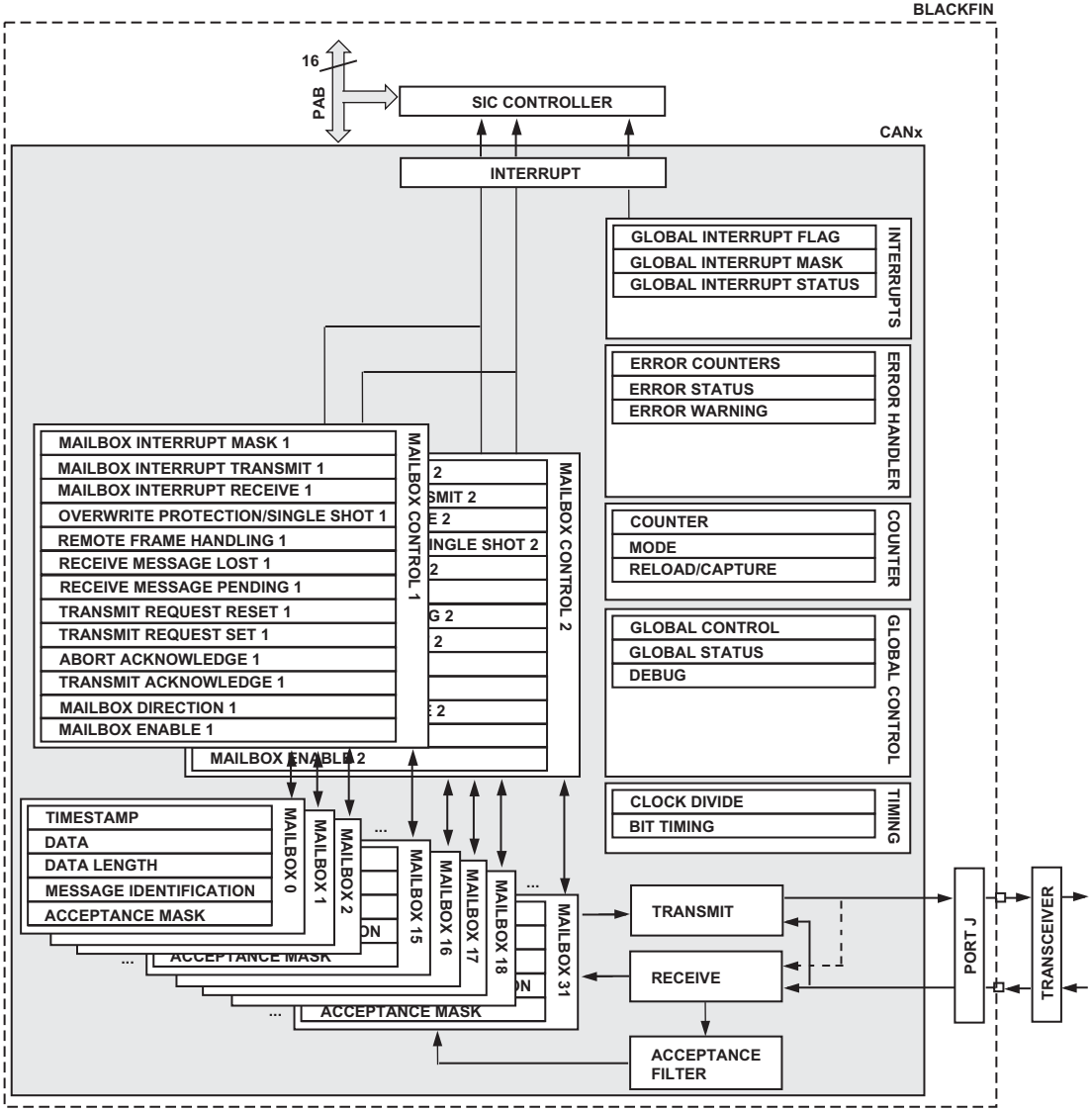


Figure 9-2. CAN Block Diagram

CAN Mailbox Area

The full-CAN controller features 32 message buffers, which are called mailboxes. Eight mailboxes are dedicated for message transmission, eight are for reception, and 16 are programmable in direction. Accordingly, the CAN module architecture is based around a 32-entry mailbox RAM. The mailbox is accessed sequentially by the CAN serial interface or the Blackfin core. Each mailbox consists of eight 16-bit control and data registers and two optional 16-bit acceptance mask registers, all of which must be configured before the mailbox itself is enabled. Since the mailbox area is implemented as RAM, the reset values of these registers are undefined. The data is divided into fields, which includes a message identifier, a time stamp, a byte count, up to 8 bytes of data, and several control bits. See Figure 9-3.

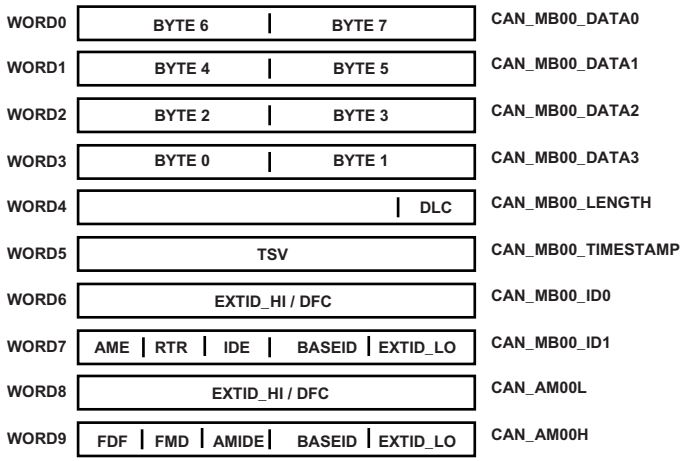


Figure 9-3. CAN Mailbox Area

The CAN mailbox identification (CAN_MBxx_ID0/1) register pair includes:

- The 29 bit identifier (base part BASEID plus extended part EXTID_LO/HI)
- The acceptance mask enable bit (AME)
- The remote transmission request bit (RTR)
- The identifier extension bit (IDE)

Do not write to the identifier of a message object while the mailbox is enabled for the CAN module (the corresponding bit in CAN_MCx is set).

The other mailbox area registers are:

- The data length code (DLC) in CAN_MBxx_LENGTH. The upper 12 bits of CAN_MBxx_LENGTH of each mailbox are marked as reserved. These 12 bits should always be set to 0. If DLC is programmed to a value greater than eight, the internal logic will set it to eight.
- Up to eight bytes for the data field, sent MSB first from the CAN_MBxx_DATA3/2/1/0 registers, respectively, based on the number of bytes defined in the DLC. For example, if only one byte is transmitted or received (DLC = 1), then it is stored in the most significant byte of the CAN_MBxx_DATA3 register.
- Two bytes for the time stamp value (TSV) in the CAN_MBxx_TIMESTAMP register

The final registers in the mailbox area are the acceptance mask registers (CAN_AMxxH and CAN_AMxxL). The acceptance mask is enabled when the AME bit is set in the CAN_MBxx_ID1 register. If the “filtering on data field” option is enabled (DNM = 1 in the CAN_CONTROL register and FDF = 1 in the corresponding acceptance mask), the EXTID_HI[15:0] bits of

CAN_MBxx_IDO are reused as acceptance code (DFC) for the data field filtering. For more details, see "Receive Operation" on page 9-15 of this chapter.

CAN Mailbox Control

Mailbox control MMRs function as control and status registers for the 32 mailboxes. Each bit in these registers represents one specific mailbox. Since CAN MMRs are all 16 bits wide, pairs of registers are required to manage certain functionality for all 32 individual mailboxes. Mailboxes 0-15 are configured/monitored in registers with a suffix of 1. Similarly, mailboxes 16-31 use the same named register with a suffix of 2. For example, the CAN mailbox direction registers (CAN_MDx) would control mailboxes as shown in Figure 9-4.

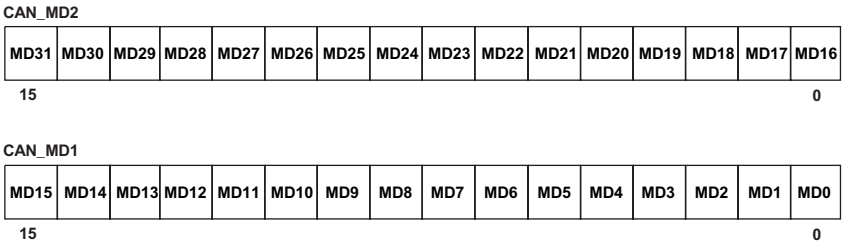


Figure 9-4. CAN Register Pairs

The mailbox control register area consists of these register pairs:

- CAN_MC1 and CAN_MC2 (mailbox enable registers)
- CAN_MD1 and CAN_MD2 (mailbox direction registers)
- CAN_TA1 and CAN_TA2 (transmit acknowledge registers)
- CAN_AA1 and CAN_AA2 (abort acknowledge registers)
- CAN_TRS1 and CAN_TRS2 (transmit request set registers)

- CAN_TRR1 and CAN_TRR2 (transmit request reset registers)
- CAN_RMP1 and CAN_RMP2 (receive message pending registers)
- CAN_RML1 and CAN_RML2 (receive message lost registers)
- CAN_RFH1 and CAN_RFH2 (remote frame handling registers)
- CAN_OPSS1 and CAN_OPSS2 (overwrite protection/single shot transmission registers)
- CAN_MBIM1 and CAN_MBIM2 (mailbox interrupt mask registers)
- CAN_MBTIF1 and CAN_MBTIF2 (mailbox transmit interrupt flag registers)
- CAN_MBRIF1 and CAN_MBRIF2 (mailbox receive interrupt flag registers)

Since mailboxes 24–31 support transmit operation only and mailboxes 0–7 are receive-only mailboxes, the lower eight bits in the "1" registers and the upper eight bits in the "2" registers are sometimes reserved or are restricted in their usage.

CAN Protocol Basics

Although the CANRX and CANTX pins are TTL-compliant signals, the CAN signals beyond the transceiver (see Figure 9-1 on page 9-2) have asymmetric drivers. A low state on the CANTX pin activates strong drivers while a high state is driven weakly. Consequently, active low is called the "dominant" state and active high is called "recessive." If the CAN module is passive, the CANTX pin is always high. If two CAN nodes transmit at the same time, dominant bits overwrite recessive bits.

The CAN protocol defines that all nodes trying to send a message on the CAN bus attempt to send a frame once the CAN bus becomes available. The start of frame indicator (SOF) signals the beginning of a new frame.

Each CAN node then begins transmitting its message starting with the message ID. While transmitting, the CAN controller samples the CANRX pin to verify that the logic level being driven is the value it just placed on the CANTX pin. This is where the names for the logic levels apply. If a transmitting node places a recessive ‘1’ on CANTX and detects a dominant ‘0’ on the CANRX pin, it knows that another node has placed a dominant bit on the bus, which means another node has higher priority. So, if the value sensed on CANRX is the value driven on CANTX, transmission continues, otherwise the CAN controller senses that it has lost arbitration and configuration determines what the next course of action is once arbitration is lost. See Figure 9-5 for more details regarding CAN frame structure.

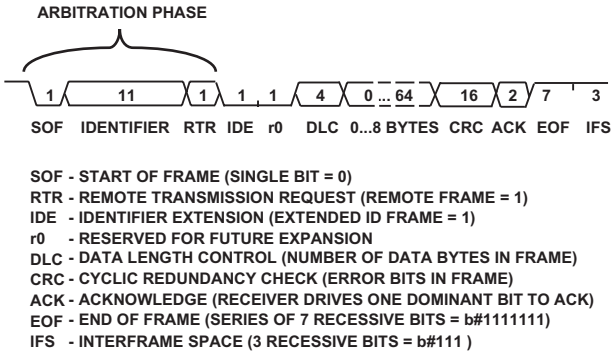


Figure 9-5. Standard CAN Frame

Figure 9-5 is a basic 11-bit identifier frame. After the SOF and identifier is the RTR bit, which indicates whether the frame contains data (data frame) or is a request for data associated with the message identifier in the frame being sent (remote frame).

Due to the inherent nature of the CAN protocol, a dominant bit in the RTR field wins arbitration against a remote frame request (RTR=1) for the same message ID, thereby defining a remote request to be lower priority than a data frame.

The next field of interest is the IDE. When set, it indicates that the message is an extended frame with a 29-bit identifier instead of an 11-bit identifier. In an extended frame, the first part of the message resembles Figure 9-6.

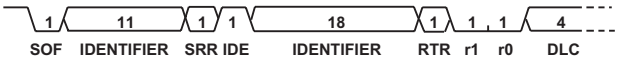


Figure 9-6. Extended CAN Frame

As could be concluded with regards to the RTR field, a dominant bit in the IDE field wins arbitration against an extended frame with the same lower 11-bits, therefore, standard frames are higher priority than extended frames. The substitute remote request bit (SRR, always sent as recessive), the reserved bits r0 and r1 (always sent as dominant), and the checksum (CRC) are generated automatically by the internal logic.

CAN Operation

The CAN controller is in configuration mode when coming out of processor reset or hibernate. It is only when the CAN is in configuration mode that hardware behavior can be altered. Before initializing the mailboxes themselves, the CAN bit timing must be set up to work on the CAN bus that the controller is expected to connect to.

Bit Timing

The CAN controller does not have a dedicated clock. Instead, the CAN clock is derived from the system clock (SCLK) based on a configurable number of time quanta. The Time Quantum (TQ) is derived from the formula TQ = (BRP+1)/SCLK, where BRP is the 10-bit BRP field in the

CAN_CLOCK register. Although the BRP field can be set to any value, it is recommended that the value be greater than or equal to 4, as restrictions apply to the bit timing configuration when BRP is less than 4.

The CAN_CLOCK register defines the TQ value, and multiple time quanta make up the duration of a CAN bit on the bus. The CAN_TIMING register controls the nominal bit time and the sample point of the individual bits in the CAN protocol. Figure 9-7 shows the three phases of a CAN bit—the synchronization segment, the segment before the sample point, and the segment after the sample point.

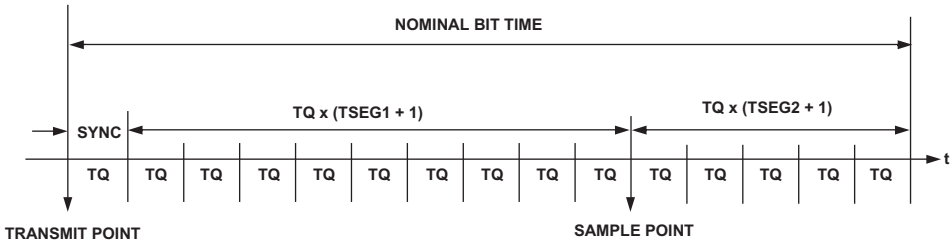


Figure 9-7. Three Phases of a CAN Bit

The synchronization segment is fixed to one TQ. It is required to synchronize the nodes on the bus. All signal edges are expected to occur within this segment.

The TSEG1 and TSEG2 fields of CAN_TIMING control how many TQs the CAN bits consist of, resulting in the CAN bit rate. The nominal bit time is given by the formula tBIT = TQ x (1 + (1 + TSEG1) + (1 + TSEG2)). For safe receive operation on given physical networks, the sample point is programmable by the TSEG1 field. The TSEG2 field holds the number of TQs needed to complete the bit time. Often, best sample reliability is achieved with sample points in the high 80% range of the bit time. Never use sample points lower than 50%. Thus, TSEG1 should always be greater than or equal to TSEG2.

The Blackfin CAN module does not distinguish between the propagation segment and the phase segment 1 as defined by the standard. The TSEG1 value is intended to cover both of them. The TSEG2 value represents the phase segment 2.

If the CAN module detects a signal edge outside the synchronization segment, it can automatically move the sampling point such that the CAN bit is still handled properly. The synchronization jump width (SJW) field specifies the maximum number of TQs, ranging from 1 to 4 (SJW + 1), allowed for such a resynchronization attempt. The SJW value should not exceed TSEG2 or TSEG1. Therefore, the fundamental rule for writing CAN_TIMING is:

```
SJW <= TSEG2 <= TSEG1
```

In addition to this fundamental rule, phase segment 2 must also be greater than or equal to the Information Processing Time (IPT). This is the time required by the logic to sample CANRX input. On the Blackfin CAN module, this is 3 SCLK cycles. Because of this, restrictions apply to the minimal value of TSEG2 if the clock prescaler BRP is lower than 2. If BRP is set to 0, the TSEG2 field must be greater than or equal to 2. If the prescaler is set to 1, the minimum TSEG2 is 1.

All nodes on a CAN bus should use the same nominal bit rate.

With all the timing parameters set, the final consideration is how sampling is performed. The default behavior of the CAN controller is to sample the CAN bit once at the sampling point described by the CAN_TIMING register, controlled by the SAM bit. If the SAM bit is set, however, the input signal is oversampled three times at the SCLK rate. The resulting value is generated by a majority decision of the three sample values. Always keep the SAM bit cleared if the BRP value is less than 4.

Do not modify the CAN_CLOCK or CAN_TIMING registers during normal operation. Always enter configuration mode first. Writes to these registers have no effect if not in configuration or debug mode. If not coming out or

processor reset or hibernate, enter configuration mode by setting the CCR bit in the master control (CAN_CONTROL) register and poll the global status (CAN_STATUS) register until the CCA bit is set.

If the TSEG1 field of the CAN_TIMING register is programmed to '0,' the module doesn't leave the configuration mode.

During configuration mode, the module is not active on the CAN bus line. The CANTX output pin remains recessive and the module does not receive/transmit messages or error frames. After leaving the configuration mode, all CAN core internal registers and the CAN error counters are set to their initial values.

A software reset does not change the values of CAN_CLOCK and CAN_TIMING. Thus, an ongoing transfer via the CAN bus cannot be corrupted by changing the bit timing parameter or initiating the software reset (SRS = 1 in CAN_CONTROL).

Transmit Operation

Figure 9-8 shows the CAN transmit operation. Mailboxes 24-31 are dedicated transmitters. Mailboxes 8-23 can be configured as transmitters by writing 0 to the corresponding bit in the CAN_MDx register. After writing the data and the identifier into the mailbox area, the message is sent after mailbox n is enabled (MCn = 1 in CAN_MCx) and, subsequently, the corresponding transmit request bit is set (TRSn = 1 in CAN_TRSx).

When a transmission completes, the corresponding bits in the transmit request set register and in the transmit request reset register (TRRn in CAN_TRRx) are cleared. If transmission was successful, the corresponding bit in the transmit acknowledge register (TAn in CAN_TAx) is set. If the transmission was aborted due to lost arbitration or a CAN error, the corresponding bit in the abort acknowledge register (AAn in CAN_AAx) is set. A requested transmission can also be manually aborted by setting the corresponding TRRn bit in CAN_TRRx.

Multiple CAN_TRSx bits can be set simultaneously by software, and these bits are reset after either a successful or an aborted transmission. The TRSn bits can also be set by the CAN hardware when using the auto-transmit mode of the universal counter, when a message loses arbitration and the single-shot bit is not set (OPSSn = 0 in CAN_OPSSx), or in the event of a remote frame request. The latter is only possible for receive/transmit mailboxes if the automatic remote frame handling feature is enabled (RFHn = 1 in CAN_RFHx).

Special care should be given to mailbox area management when a TRSn bit is set. Write access to the mailbox is permissible with TRSn set, but changing data in such a mailbox may lead to unexpected data during transmission.

Enabling and disabling mailboxes has an impact on transmit requests. Setting the TRSn bit associated with a disabled mailbox may result in erroneous behavior. Similarly, disabling a mailbox before the associated TRSn bit is reset by the internal logic can cause unpredictable results.

Retransmission

Normally, the current message object is sent again after arbitration is lost or an error frame is detected on the CAN bus line. If there is more than one transmit message object pending, the message object with the highest mailbox is sent first (see Figure 9-8). The currently aborted transmission is restarted after any messages with higher priority are sent.

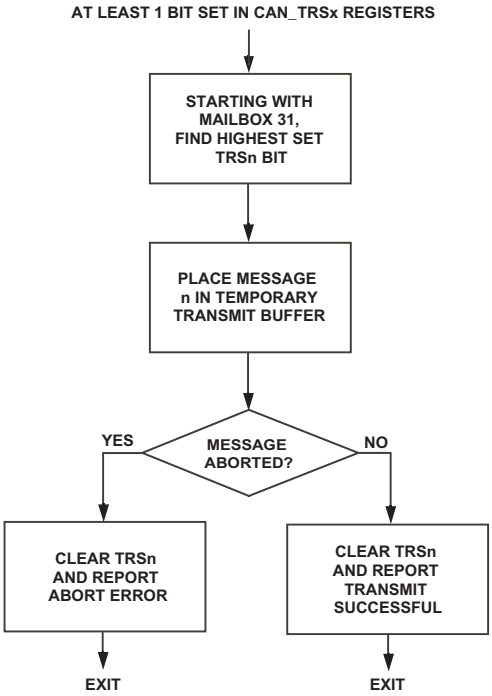


Figure 9-8. CAN Transmit Operation Flow Chart

A message which is currently under preparation is not replaced by another message which is written into the mailbox. The message under preparation is one that is copied into the temporary transmit buffer when the internal transmit request for the CAN core module is set. The message in the buffer is not replaced until it is sent successfully, the arbitration on the CAN bus line is lost, or there is an error frame on the CAN bus line.

Single Shot Transmission

If the single shot transmission feature is used (OPSSn = 1 in CAN_OPSSx), the corresponding TRSn bit is cleared after the message is successfully sent or if the transmission is aborted due to a lost arbitration or an error frame on the CAN bus line. Thus, there is no further attempt to transmit the message again if the initial try failed, and the abort error is reported (AAn = 1 in CAN_AAx)

Auto-Transmission

In auto-transmit mode, the message in mailbox 11 can be sent periodically using the universal counter. This mode is often used to broadcast heartbeats to all CAN nodes. Accordingly, messages sent this way usually have high priority.

The period value is written to the CAN_UCRC register. When enabled in this mode (set UCCNF[3:0] = 0x3 in CAN_UCCNF), the counter (CAN_UCCNT) is loaded with the value in the CAN_UCRC register. The counter decrements at the CAN bit clock rate down to 0 and is then reloaded from CAN_UCRC. Each time the counter reaches a value of 0, the TRS11 bit is automatically set by internal logic, and the corresponding message from mailbox 11 is sent.

For proper auto-transmit operation, mailbox 11 must be configured as a transmit mailbox and must contain valid data (identifier, control bits, and data) before the counter first expires after this mode is enabled.

Receive Operation

The CAN hardware autonomously receives messages and discards invalid messages. Once a valid message has been successfully received, the receive logic interrogates all enabled receive mailboxes sequentially, from mailbox 23 down to mailbox 0, whether the message is of interest to the local node or not.

Each incoming data frame is compared to all identifiers stored in active receive mailboxes (MDn = 1 and MCn = 1) and to all active transmit mailboxes with the remote frame handling feature enabled (RFHn = 1 in CAN_RFHx).

The message identifier of the received message, along with the identifier extension (IDE) and remote transmission request (RTR) bits, are compared against each mailbox's register settings. If the AME bit is not set, a match is signalled only if IDE, RTR, and all identifier bits are exact. If, however, AME is set, the acceptance mask registers determine which of the identifier, IDE, and RTR bits need to match. This way, a mailbox can accept a group of messages.

Table 9-1. Mailbox Used for Acceptance Mask Filtering

Mailbox used for Acceptance Filtering				
MCn	**MDn**	**RFHn**	**Mailbox n**	**Comment**
0	x	x	Ignored	Mailbox n disabled
1	0	0	Ignored	Mailbox n enabled Mailbox n configured for transmit Remote frame handling disabled
1	0	1	Used	Mailbox n enabled Mailbox n configured for transmit Remote frame handling enabled
1	1	x	Used	Mailbox n enabled Mailbox n configured for receive

If the acceptance filter finds a matching identifier, the content of the received data frame is stored in that mailbox. A received message is stored only once, even if multiple receive mailboxes match its identifier. If the current identifier does not match any mailbox, the message is not stored.

Figure 9-9 illustrates the decision tree of the receive logic when processing the individual mailboxes.

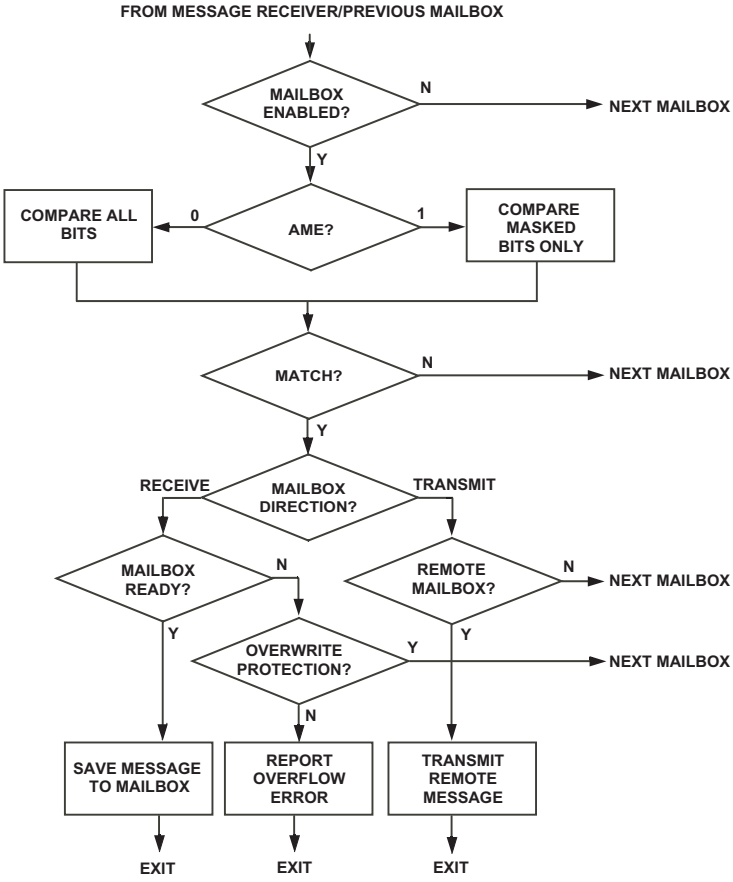


Figure 9-9. CAN Receive Operation Flow Chart

If a message is received for a mailbox and that mailbox still contains unread data (`RMPn = 1`), the user has to decide whether the old message should be overwritten or not. If `OPSSn = 0`, the receive message lost bit (`RMLn` in `CAN_RMLx`) is set and the stored message is overwritten. This results in the receive message lost interrupt being raised in the global

interrupt status register (RMLIS = 1 in CAN_GIS). If OPSSn = 1, the next mailboxes are checked for another matching identifier. If no match is found, the message is discarded and the next message is checked.

If a receive mailbox is disabled, an ongoing receive message for that mailbox is lost even if a second mailbox is configured to receive the same identifier.

Data Acceptance Filter

If DeviceNet mode is enabled (DNM = 1 in CAN_CONTROL) and the mailbox is set up for filtering on data field, the filtering is done on the standard ID of the message and data fields. The data field filtering can be programmed for either the first byte only or the first two bytes, as shown in Table 9-2.

Table 9-2. Data Field Filtering

FDF Filter On Data Field	FMD Full Mask Data Field	Description
0	0	Do not allow filtering on the data field
0	1	Not allowed. FMD must be 0 if FDF is 0.
1	0	Filter on first data byte only
1	1	Filter on first two data bytes

If the FDF bit is set in the corresponding CAN_AMxxH register, the CAN_AMxxL register holds the data field mask (DFM[15:0]). If the FDF bit is cleared in the corresponding CAN_AMxxH register, the CAN_AMxxL register holds the extended identifier mask (EXTID_HI[15:0]).

Remote Frame Handling

Automatic handling of remote frames can be enabled/disabled by setting/clearing the corresponding bit in the remote frame handling registers (CAN_RFHx).

Remote frames are data frames with no data field and the RTR bit set. The data length code of the data frame is equal to the DLC of the corresponding remote frame. A data length code can be programmed with values in the range of 0 to 15, but data length code values greater than 8 are considered as 8. A remote frame contains:

- the identifier bits
- the control field DLC
- the remote transmission request (RTR) bit

Only configurable mailboxes 8–23 can process remote frames, but all message centers can receive and transmit remote frame requests. When setup for automatic remote frame handling, the CAN_OPSSx register has no effect. All content of a mailbox is always overwritten by an incoming message.

If a remote frame is received, the DLC of the corresponding mailbox is overwritten with the received value.

Erroneous behavior may result when the remote frame handling bit (RFHn) is changed and the corresponding mailbox is currently processed.

Watchdog Mode

Watchdog mode is used to make sure messages are received periodically. It is often used to observe whether or not a certain node on the network is alive and functioning properly, and, if not, to detect and manage its failure case accordingly.

Upon programming the universal counter to watchdog mode (set UCCNF[3:0] = 0x2 in CAN_UCCNF), the counter in the CAN_UCCNT register is loaded with the predefined value contained in the CAN universal counter reload/capture register (CAN_UCRC). This counter then decrements at the CAN bit rate. If the UCCT and UCRC bits in the CAN_UCCNF register are set and a message is received in mailbox 4 before the counter counts down to 0, the counter is reloaded with the CAN_UCRC contents. If the counter has counted down to 0 without receiving a message in mailbox 4, the UCEIS bit in the global interrupt status (CAN_GIS) register is set, and the counter is automatically reloaded with the contents of the CAN_UCRC register. If an interrupt is desired, the UCEIM bit in the CAN_GIM register must also be set. With the mask bit set, when a watchdog interrupt occurs, the UCEIF bit in the CAN_GIF register is also set.

The counter can be reloaded with the contents of CAN_UCRC or disabled by writing to the CAN_UCCNF register.

The time period it takes for the watchdog interrupt to occur is controlled by the value written into the CAN_UCRC register by the user.

Time Stamps

To get an indication of the time of reception or the time of transmission for each message, program the CAN universal counter to time stamp mode (set UCCNF[3:0] = 0x1 in CAN_UCCNF). The value of the 16-bit free-running counter (CAN_UCCNT) is then written into the CAN_MBxx_TIMESTAMP register of the corresponding mailbox when a received message has been stored or a message has been transmitted.

The time stamp value is captured at the sample point of the start of frame (SOF) bit of each incoming or outgoing message. Afterwards, this time stamp value is copied to the CAN_MBxx_TIMESTAMP register of the corresponding mailbox.

If the mailbox is configured for automatic remote frame handling, the time stamp value is written for transmission of a data frame (mailbox configured as transmit) or the reception of the requested data frame (mailbox configured as receive).

The counter can be cleared (set UCRC bit to 1) or disabled (set UCE bit to 0) by writing to the CAN_UCCNF register. The counter can also be loaded with a value by writing to the counter register itself (CAN_UCCNT).

It is also possible to clear the counter (CAN_UCCNT) by reception of a message in mailbox number 4 (synchronization of all time stamp counters in the system). This is accomplished by setting the UCCT bit in the CAN_UCCNF register.

An overflow of the counter sets a bit in the global interrupt status register (UCEIS in the CAN_GIS register). A global interrupt can optionally occur by unmasking the bit in the global interrupt mask register (UCEIM in the CAN_GIM register). If the interrupt source is unmasked, a bit in the global interrupt flag register is also set (UCEIF in the CAN_GIF register).

Temporarily Disabling Mailboxes

If a mailbox is enabled and configured as “transmit,” write accesses to the data field are denied. If this mailbox is used for automatic remote frame handling, the data field must be updated without losing an incoming remote request frame and without sending inconsistent data. Therefore, the CAN controller allows for temporary mailbox disabling, which can be enabled by programming the mailbox temporary disable register (CAN_MBTD).

The pointer to the requested mailbox must be written to the TDPTR[4:0] bits of the CAN_MBTD register and the mailbox temporary disable request bit (TDR) must be set. The corresponding mailbox temporary disable flag (TDA) is subsequently set by the internal logic.

If a mailbox is configured as “transmit” (MDn = 0) and TDA is set, the content of the data field of that mailbox can be updated. If there is an incoming remote request frame while the mailbox is temporarily disabled, the corresponding transmit request set bit (TRSn) is set by the internal logic and the data length code of the incoming message is written to the corresponding mailbox. However, the message being requested is not sent until the temporary disable request is cleared (TDR = 0). Similarly, all transmit requests for temporarily disabled mailboxes are ignored until TDR is cleared. Additionally, transmission of a message is immediately aborted if the mailbox is temporarily disabled and the corresponding TRRn bit for this mailbox is set.

If a mailbox is configured as “receive” (MDn = 1), the temporary disable flag is set and the mailbox is not processed. If there is an incoming message for the mailbox n being temporarily disabled, the internal logic waits until the reception is complete or there is an error on the CAN bus to set TDA. Once TDA is set, the mailbox can then be completely disabled (MCn = 0) without the risk of losing an incoming frame. The temporary disable request (TDR) bit must then be reset as soon as possible.

When TDA is set for a given mailbox, only the data field of that mailbox can be updated. Accesses to the control bits and the identifier are denied.

Functional Operation

The following sections describe the functional operation of the CAN module, including interrupts, the event counter, warnings and errors, debug features, and low power features.

CAN Interrupts

The CAN module provides three independent interrupts: two mailbox interrupts (mailbox receive interrupt MBRIRQ and mailbox transmit interrupt MBTIRQ) and the global interrupt GIRQ. The values of these three interrupts can also be read back in the interrupt status registers.

Mailbox Interrupts

Each of the 32 mailboxes in the CAN module may generate a receive or transmit interrupt, depending on the mailbox configuration. To enable a mailbox to generate an interrupt, set the corresponding MBIMn bit in CAN_MBIMx.

If a mailbox is configured as a receive mailbox, the corresponding receive interrupt flag is set (MBRIFn = 1 in CAN_MBRIFx) after a received message is stored in mailbox n (RMPn = 1 in CAN_RMPx). If the automatic remote frame handling feature is used, the receive interrupt flag is set after the requested data frame is stored in the mailbox. If any MBRIFn bits are set in CAN_MBRIFx, the MBRIRQ interrupt output is raised in CAN_INTR. In order to clear the MBRIRQ interrupt request, all of the set MBRIFn bits must be cleared by software by writing a 1 to those set bit locations in CAN_MBRIFx.

If a mailbox is configured as a transmit mailbox, the corresponding transmit interrupt flag is set (MBTIFn = 1 in CAN_MBTIFx) after the message in mailbox n is sent correctly (TAn = 1 in CAN_TAx). The TAn bits maintain state even after the corresponding mailbox n is disabled (MCn = 0). If the automatic remote frame handling feature is used, the transmit interrupt flag is set after the requested data frame is sent from the mailbox. If any MBTIFn bits are set in CAN_MBTIFx, the MBTIRQ interrupt output is raised in CAN_INTR. In order to clear the MBTIRQ interrupt request, all of the set MBTIFn bits must be cleared by software by writing a 1 to those set bit locations in CAN_MBTIFx.

Global Interrupt

The global interrupt logic is implemented with three registers—the global interrupt mask register (CAN_GIM), where each interrupt source can be enabled or disabled separately; the global interrupt status register (CAN_GIS); and the global interrupt flag register (CAN_GIF). The interrupt mask bits only affect the content of the global interrupt flag register (CAN_GIF). If the mask bit is not set, the corresponding flag bit is not set when the event occurs. The interrupt status bits in the global interrupt status register, however, are always set if the corresponding interrupt event occurs, independent of the mask bits. Thus, the interrupt status bits can be used for polling of interrupt events.

The global interrupt output (GIRQ) bit in the global interrupt status register is only asserted if a bit in the CAN_GIF register is set. The GIRQ bit remains set as long as at least one bit in the interrupt flag register CAN_GIF is set. All bits in the interrupt status and in the interrupt flag registers remain set until cleared by software or a software reset has occurred.

There are several interrupt events that can activate this GIRQ interrupt:

- **Access denied interrupt** (ADIM, ADIS, ADIF)

  At least one access to the mailbox RAM occurred during a data update by internal logic.

- **External trigger output interrupt** (EXTIM, EXTIS, EXTIF)

  The external trigger event occurred.

- **Universal counter exceeded interrupt** (UCEIM, UCEIS, UCEIF)

  There was an overflow of the universal counter (in time stamp mode or event counter mode) or the counter has reached the value 0x0000 (in watchdog mode).

- **Receive message lost interrupt** (RMLIM, RMLIS, RMLIF)

  A message has been received for a mailbox that currently contains unread data. At least one bit in the receive message lost register (CAN_RMLx) is set. If the bit in CAN_GIS (and CAN_GIF) is reset and there is at least one bit in CAN_RMLx still set, the bit in CAN_GIS (and CAN_GIF) is not set again. The internal interrupt source signal is only active if a new bit in CAN_RMLx is set.

- **Abort acknowledge interrupt** (AAIM, AAIS, AAIF)

  At least one AAn bit in the abort acknowledge registers CAN_AAx is set. If the bit in CAN_GIS (and CAN_GIF) is reset and there is at least one bit in CAN_AAx still set, the bit in CAN_GIS (and CAN_GIF) is not set again. The internal interrupt source signal is only active if a new bit in CAN_AAx is set. The AAn bits maintain state even after the corresponding mailbox n is disabled (MCn = 0).

- **Access to unimplemented address interrupt** (UIAIM, UIAIS, UIAIF)

  There was a CPU access to an address which is not implemented in the controller module.

- **Wakeup interrupt** (WUIM, WUIS, WUIF)

  The CAN module has left the sleep mode because of detected activity on the CAN bus line.

- **Bus-Off interrupt** (BOIM, BOIS, BOIF)

  The CAN module has entered the bus-off state. This interrupt source is active if the status of the CAN core changes from normal operation mode to the bus-off mode. If the bit in CAN_GIS (and CAN_GIF) is reset and the bus-off mode is still active, this bit is not set again. If the module leaves the bus-off mode, the bit in CAN_GIS (and CAN_GIF) remains set.

- **Error-Passive interrupt** (EPIM, EPIS, EPIF)

  The CAN module has entered the error-passive state. This interrupt source is active if the status of the CAN module changes from the error-active mode to the error-passive mode. If the bit in CAN_GIS (and CAN_GIF) is reset and the error-passive mode is still active, this bit is not set again. If the module leaves the error-passive mode, the bit in CAN_GIS (and CAN_GIF) remains set.

- **Error warning receive interrupt** (EWRIM, EWRIS, EWRIF)

  The CAN receive error counter (RXECNT) has reached the warning limit. If the bit in CAN_GIS (and CAN_GIF) is reset and the error warning mode is still active, this bit is not set again. If the module leaves the error warning mode, the bit in CAN_GIS (and CAN_GIF) remains set.

- **Error warning transmit interrupt** (EWTIM, EWTIS, EWTIF)

  The CAN transmit error counter (TXECNT) has reached the warning limit. If the bit in CAN_GIS (and CAN_GIF) is reset and the error warning mode is still active, this bit is not set again. If the module leaves the error warning mode, the bit in CAN_GIS (and CAN_GIF) remains set.

Event Counter

For diagnostic functions, it is possible to use the universal counter as an event counter. The counter can be programmed in the 4-bit UCCNF[3:0] field of CAN_UCCNF to increment on one of these conditions:

- UCCNF[3:0] = 0x6 – CAN error frame. Counter is incremented if there is an error frame on the CAN bus line.
- UCCNF[3:0] = 0x7 – CAN overload frame. Counter is incremented if there is an overload frame on the CAN bus line.

- `UCCNF[3:0] = 0x8` – Lost arbitration. Counter is incremented every time arbitration on the CAN line is lost during transmission.
- `UCCNF[3:0] = 0x9` – Transmission aborted. Counter is incremented every time arbitration is lost or a transmit request is cancelled (`AAn` is set).
- `UCCNF[3:0] = 0xA` – Transmission succeeded. Counter is incremented every time a message sends without detected errors (`TAn` is set).
- `UCCNF[3:0] = 0xB` – Receive message rejected. Counter is incremented every time a message is received without detected errors but not stored in a mailbox because there is no matching identifier found.
- `UCCNF[3:0] = 0xC` – Receive message lost. Counter is incremented every time a message is received without detected errors but not stored in a mailbox because the mailbox contains unread data (`RMLn` is set).
- `UCCNF[3:0] = 0xD` – Message received. Counter is incremented every time a message is received without detected errors, whether the received message is rejected or stored in a mailbox.
- `UCCNF[3:0] = 0xE` – Message stored. Counter is incremented every time a message is received without detected errors, has an identifier that matches an enabled receive mailbox, and is stored in the receive mailbox (`RMPn` is set).
- `UCCNF[3:0] = 0xF` – Valid message. Counter is incremented every time a valid transmit or receive message is detected on the CAN bus line.

CAN Warnings and Errors

CAN warnings and errors are controlled using the CAN_CEC register, the CAN_ESR register, and the CAN_EWR register.

Programmable Warning Limits

It is possible to program the warning level for EWTIS (error warning transmit interrupt status) and EWRIS (error warning receive interrupt status) separately by writing to the error warning level error count fields for receive (EWLREC) and transmit (EWLTEC) in the CAN error counter warning level (CAN_EWR) register. After powerup reset, the CAN_EWR register is set to the default warning level of 96 for both error counters. After software reset, the content of this register remains unchanged.

CAN Error Handling

Error management is an integral part of the CAN standard. Five different kinds of bus errors may occur during transmissions:

- **Bit error**

  A bit error can be detected by the transmitting node only. Whenever a node is transmitting, it continuously monitors its receive pin (CANRX) and compares the received data with the transmitted data. During the arbitration phase, the node simply postpones the transmission if the received and transmitted data do not match. However, after the arbitration phase (that is, once the RTR bit has been sent successfully), a bit error is signaled any time the value on CANRX does not equal what is being transmitted on CANTX.

- **Form error**

  A form error occurs any time a fixed-form bit position in the CAN frame contains one or more illegal bits, that is, when a dominant bit is detected at a delimiter or end-of-frame bit position.

- **Acknowledge error**

  An acknowledge error occurs whenever a message has been sent and no receivers drive an acknowledge bit.

- **CRC error**

  A CRC error occurs whenever a receiver calculates the CRC on the data it received and finds it different than the CRC that was transmitted on the bus itself.

- **Stuff error**

  The CAN specification requires the transmitter to insert an extra stuff bit of opposite value after 5 bits have been transmitted with the same value. The receiver disregards the value of these stuff bits. However, it takes advantage of the signal edge to resynchronize itself. A stuff error occurs on receiving nodes whenever the 6th consecutive bit value is the same as the previous five bits.

Once the CAN module detects any of the above errors, it updates the error status register `CAN_ESR` as well as the error counter register `CAN_CEC`. In addition to the standard errors, the `CAN_ESR` register features a flag that signals when the `CANRX` pin sticks at dominant level, indicating that shorted wires are likely.

Error Frames

It is of central importance that all nodes on the CAN bus ignore data frames that one single node failed to receive. To accomplish this, every node sends an error frame as soon as it has detected an error. See Figure 9-10.

Once a device has detected an error, it still completes the ongoing bit and initiates an error frame by sending six dominant and eight recessive bits to the bus. This is a violation to the bit stuffing rule and informs all nodes that the ongoing frame needs to be discarded.

All receivers that did not detect the transmission error in the first instance now detect a stuff bit error. The transmitter may detect a normal bit error sooner. It aborts the transmission of the ongoing frame and tries sending it again later.

Finally, all nodes on the bus have detected an error. Consequently, all of them send 6 dominant and 8 recessive bits to the bus as well. The resulting error frame consists of two different fields. The first field is given by the superposition of error flags contributed from the different stations, which is a sequence of 6 to 12 dominant bits. The second field is the error delimiter and consists of 8 recessive bits indicating the end of frame.

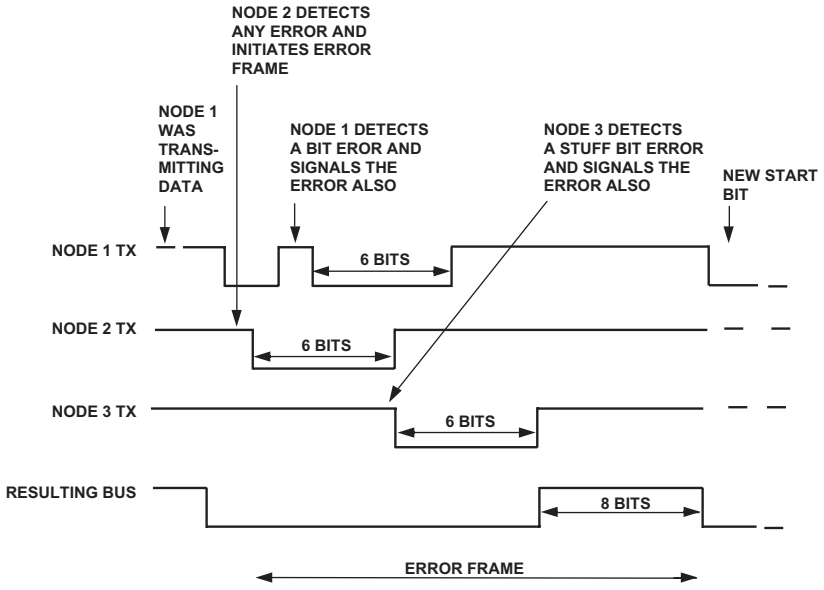


Figure 9-10. CAN Error Scenario Example

For CRC errors, the error frame is initiated at the end of the frame, rather than immediately after the failing bit.

After having received 8 recessive bits, every node knows that the error condition has been resolved and starts transmission if messages are pending. The former transmitter that had to abort its operation must win the new arbitration again, otherwise its message is delayed as determined by priority.

Because the transmission of an error frame destroys the frame under transmission, a faulty node erroneously detecting an error can block the bus. Because of this, there are two node states which determine a node's right to signal an error—error active and error passive. Error active nodes are those which have an error detection rate below a certain limit. These nodes drive an 'active error flag' of 6 dominant bits.

Nodes with a higher error detection rate are suspected of having a local problem and, therefore, have a limited right to signal errors. These error passive nodes drive a 'passive error flag' consisting of 6 recessive bits. Thus, an error passive transmitting node is still able to inform the other nodes about the abortion of a self-transmitted frame, but it is no longer able to destroy correctly received frames of other nodes.

Error Levels

The CAN specification requires each node in the system to operate in one of three levels. See Table 9-3. This prevents nodes with high error rates from blocking the entire network, as the errors might be caused by local hardware. The Blackfin CAN module provides an error counter for transmit (`TEC`) and an error counter for receive (`REC`). The CAN error count register `CAN_CEC` houses each of these 8-bit counters.

After initialization, both the `TEC` and the `REC` counters are 0. Each time a bus error occurs, one of the counters is incremented by either 1 or 8, depending on the error situation (documented in Version 2.0 of *CAN Specification*). Successful transmit and receive operations decrement the respective counter by 1.

If either of the error counters exceeds 127, the CAN module goes into a passive state and the CAN error passive mode (EP) bit in CAN_STATUS is set. Then, it is not allowed to send any more active error frames. However, it is still allowed to transmit messages and to signal passive error frames in case the transmission fails because of a bit error.

If one of the counters exceeds 255 (that is, when the 8-bit counters overflow), the CAN module is disconnected from the bus. It goes into bus off mode and the CAN error bus off mode (EBO) bit is set in CAN_STATUS. Software intervention is required to recover from this state.

Table 9-3. CAN Error Level Description

Level	Condition	Description
Error active	Transmit and receive error counters < 128	This is the initial condition level. As long as errors stay below 128, the node will drive active error flags during error frames.
Error passive	Transmit or receive error counters ≥ 128, but < 256	Errors have accumulated to a level which requires the node to drive passive error flags during error frames.
Bus off	Transmit or receive error counters ≥ 256	CAN module goes into bus off mode

In addition to these levels, the CAN module also provides a warning mechanism, which is an enhancement to the CAN specification. There are separate warnings for transmit and receive. By default, when one of the error counters exceeds 96, a warning is signaled and is represented in the CAN_STATUS register by either the CAN receive warning flag (WR) or CAN transmit warning flag (WT) bits. The error warning level can be programmed using the error warning register, CAN_EWR. More information is available on page 9-86.

Additionally, interrupts can occur for all of these levels by unmasking them in the global interrupt mask register (CAN_GIM) shown on page 9-47. The interrupts include the bus off interrupt (BOIM), the error-passive interrupt (EPIM), the error warning receive interrupt (EWRIM), and the error warning transmit interrupt (EWTIM).

During the bus off recovery sequence, the configuration mode request bit in the CAN_CONTROL register is set by the internal logic (CCR = 1), thus the CAN core module does not automatically come out of the bus off mode. The CCR bit cannot be reset until the bus off recovery sequence is finished.

This behavior can be over-ridden by setting the auto-bus on (ABO) bit in the CAN_CONTROL register. After exiting the bus off or configuration modes, the CAN error counters are reset.

Debug and Test Modes

The CAN module contains test mode features that aid in the debugging of the CAN software and system. Listing 9-1 provides an example of enabling CAN debug features.

When these features are used, the CAN module may not be compliant to the CAN specification. All test modes should be enabled or disabled only when the module is in configuration mode (CCA = 1 in the CAN_STATUS register) or in suspend mode (CSA = 1 in CAN_STATUS).

The CDE bit is used to gain access to all of the debug features. This bit must be set to enable the test mode, and must be written first before subsequent writes to the CAN_DEBUG register. When the CDE bit is cleared, all debug features are disabled.

Listing 9-1. Enabling CAN Debug Features in C

```
#include <cdefBF537.h>
/* Enable debug mode, CDE must be set before other flags can be
changed in register */
*pCAN_DEBUG |= CDE ;

/* Set debug flags */
*pCAN_DEBUG &= ~DTO ;
*pCAN_DEBUG |= MRB | MAA | DIL ;

/* Run test code */

/* Disable debug mode */
*pCAN_DEBUG &= ~CDE ;
```

When the CDE bit is set, it enables writes to the other bits of the CAN_DEBUG register. It also enables these features, which are not compliant with the CAN standard:

- Bit timing registers can be changed anytime, not only during configuration mode. This includes the CAN_CLOCK and CAN_TIMING registers.
- Allows write access to the read-only transmit/receive error counter register CAN_CEC.

The mode read back bit (MRB) is used to enable the read back mode. In this mode, a message transmitted on the CAN bus (or via an internal loop back mode) is received back directly to the internal receive buffer. After a correct transmission, the internal logic treats this as a normal receive message. This feature allows the user to test most of the CAN features without an external device.

The mode auto acknowledge bit (MAA) allows the CAN module to generate its own acknowledge during the ACK slot of the CAN frame. No external devices or connections are necessary to read back a transmit message. In this mode, the message that is sent is automatically stored in the internal receive buffer. In auto acknowledge mode, the module itself transmits the acknowledge. This acknowledge can be programmed to appear on the CANTX pin if DIL=1 and DTO=0. If the acknowledge is only going to be used internally, then these test mode bits should be set to DIL=0 and DTO=1.

The disable internal loop bit (DIL) is used to internally enable the transmit output to be routed back to the receive input.

The disable transmit output bit (DTO) is used to disable the CANTX output pin. When this bit is set, the CANTX pin continuously drives recessive bits.

The disable receive input bit (DRI) is used to disable the CANRX input. When set, the internal logic receives recessive bits or receives the internally generated transmit value in the case of the internal loop enabled (DIL=0). In either case, the value on the CANRX input pin is ignored.

The disable error counters bit (DEC) is used to disable the transmit and receive error counters in the CAN_CEC register. When this bit is set, the CAN_CEC holds its current contents and is not allowed to increment or decrement the error counters. This mode does not conform to the CAN specification.

Writes to the error counters should be in debug mode only. Write access during reception may lead to undefined values. The maximum value which can be written into the error counters is 255. Thus, the error counter value of 256 which forces the module into the bus off state can not be written into the error counters.

Table 9-4 shows several common combinations of test mode bits.

Table 9-4. CAN Test Modes

MRB	MAA	DIL	DTO	DRI	CDE	Functional Description
X	X	X	X	X	0	Normal mode, not debug mode.
0	X	X	X	X	X	No read back of transmit message.
1	0	1	0	0	1	Normal transmission on CAN bus line. Read back. External acknowledge from external device required.
1	1	1	0	0	1	Normal transmission on CAN bus line. Read back. No external acknowledge required. Transmit message and acknowledge are transmitted on CAN bus line. CANRX input is enabled.
1	1	0	0	0	1	Normal transmission on CAN bus line. Read back. No external acknowledge required. Transmit message and acknowledge are transmitted on CAN bus line. CANRX input and internal loop are enabled (internal OR of TX and RX).
1	1	0	0	1	1	Normal transmission on CAN bus line. Read back. No external acknowledge required. Transmit message and acknowledge are transmitted on CAN bus line. CANRX input is ignored. Internal loop is enabled

Table 9-4. CAN Test Modes (Cont'd)

MRB	MAA	DIL	DTO	DRI	CDE	Functional Description
1	1	0	1	1	1	No transmission on CAN bus line. Read back. No external acknowledge required. Neither transmit message nor acknowledge are transmitted on CANTX. CANRX input is ignored. Internal loop is enabled.

Low Power Features

The Blackfin processor provides a low power hibernate state, and the CAN module includes built-in sleep and suspend modes to save power. The behavior of the CAN module in these three modes is described in the following sections.

CAN Built-In Suspend Mode

The most modest of power savings modes is the suspend mode. This mode is entered by setting the suspend mode request (CSR) bit in the CAN_CONTROL register. The module enters the suspend mode after the current operation of the CAN bus is finished, at which point the internal logic sets the suspend mode acknowledge (CSA) bit in CAN_STATUS. Once this mode is entered, the module is no longer active on the CAN bus line, slightly reducing power consumption. When the CAN module is in suspend mode, the CANTX output pin remains recessive and the module does not receive/transmit messages or error frames. The content of the CAN error counters remains unchanged.

The suspend mode can subsequently be exited by clearing the CSR bit in CAN_CONTROL. The only differences between suspend mode and configuration mode are that writes to the CAN_CLOCK and CAN_TIMING registers are still locked in suspend mode and the CAN control and status registers are not reset when exiting suspend mode.

CAN Built-In Sleep Mode

The next level of power savings can be realized by using the CAN module's built-in sleep mode. This mode is entered by setting the sleep mode request (SMR) bit in the CAN_CONTROL register. The module enters the sleep mode after the current operation of the CAN bus is finished. Once this mode is entered, many of the internal CAN module clocks are shut off, reducing power consumption, and the sleep mode acknowledge (SMACK) bit is set in CAN_INTR. When the CAN module is in sleep mode, all register reads return the contents of CAN_INTR instead of the usual contents. All register writes, except to CAN_INTR, are ignored in sleep mode.

A small part of the module is clocked continuously to allow for wakeup out of sleep mode. A write to the CAN_INTR register ends sleep mode. If the WBA bit in the CAN_CONTROL register is set before entering sleep mode, a dominant bit on the CANRX pin also ends sleep mode.

CAN Wakeup From Hibernate State

For greatest power savings, the Blackfin processor provides a hibernate state, where the internal voltage regulator shuts off the internal power supply to the chip, turning off the core and system clocks in the process. In this mode, the only power drawn (roughly 50µA) is that used by the regulator circuitry awaiting any of the possible hibernate wakeup events. One such event is a wakeup due to CAN bus activity. After hibernation, the CAN module must be re-initialized.

For low power designs, the external CAN bus transceiver is typically put into standby mode via one of the Blackfin processor's general purpose I/O pins. While in standby mode, the CAN transceiver continually drives the

recessive logic '1' level onto the CANRX pin. If the transceiver then senses CAN bus activity, it will, in turn, drive the CANRX pin to the dominant logic '0' level. This signals to the Blackfin processor that CAN bus activity has been detected. If the internal voltage regulator is programmed to recognize CAN bus activity as an event to exit hibernate state, the part responds appropriately. Otherwise, the activity on the CANRX pin has no effect on the processor state.

To enable this functionality, the voltage control register (VR_CTL) must be programmed with the CAN wakeup enable bit set. The typical sequence of events to use the CAN wakeup feature is:

1. Use a general-purpose I/O pin to put the external transceiver into standby mode.
2. Program VR_CTL with the CAN wakeup enable bit (CANWE) set and the FREQ field set to 00.

Register Definitions

The following sections describe the CAN register definitions.

Table 9-5 through Table 9-9 show the functions of the CAN registers.

Table 9-5. CAN Global Register Mapping

Register Name	Function	Notes
CAN_CONTROL	Master control register	Reserved bits 15:8 and 3 must always be written as '0'
CAN_STATUS	Global status register	Write accesses have no effect
CAN_DEBUG	CAN debug register	Use of these modes is not CAN-compliant
CAN_CLOCK	CAN clock register	Accessible only in configuration mode

Table 9-5. CAN Global Register Mapping (Cont'd)

Register Name	Function	Notes
CAN_TIMING	CAN timing register	Accessible only in configuration mode
CAN_INTR	CAN interrupt register	Reserved bits 15:8 and 5:4 must always be written as '0'
CAN_GIM	Global interrupt mask register	Bits 15:11 are reserved
CAN_GIS	Global interrupt status register	Bits 15:11 are reserved
CAN_GIF	Global interrupt flag register	Bits 15:11 are reserved

Table 9-6. CAN Mailbox/Mask Register Mapping

Register Name	Function	Notes
CAN_AMxxH/L	Acceptance mask registers	Change only when mailbox MBxx is disabled
CAN_MBxx_ID1/0	Mailbox word 7/6 register	Do not write when MBxx is enabled
CAN_MBxx_ TIMESTAMP	Mailbox word 5 register	Holds timestamp information when timestamp mode is active
CAN_MBxx_LENGTH	Mailbox word 4 register	Values greater than 8 are treated as 8
CAN_MBxx_ DATA3/2/1/0	Mailbox word 3/2/1/0 register	Software controls reading correct data based on DLC

Table 9-7. CAN Mailbox Control Register Mapping

Register Name	Function	Notes
CAN_MCx	Mailbox configuration registers	Always disable before modifying mailbox area or direction
CAN_MDx	Mailbox direction registers	Never change MDn direction when mailbox n is enabled. MD[31:24] and MD[7:0] are read only
CAN_RMPx	Receive message pending registers	Clearing RMPn bits also clears corresponding RMLn bits
CAN_RMLx	Receive message lost registers	Write accesses have no effect
CAN_OPSSx	Overwrite protection or single-shot transmission register	Function depends on mailbox direction. Has no effect when RFHn = 1. Do not modify OPSSn bit if mailbox n is enabled
CAN_TRSx	Transmission request set registers	May by set by internal logic under certain circumstances. TRS[7:0] are read-only
CAN_TRRx	Transmission request reset registers	TRRn bits must not be set if mailbox n is disabled or TRSn = 0
CAN_AAx	Abort acknowledge registers	AAn bit is reset if TRSn bit is set manually, but not when TRSn is set by internal logic
CAN_TAx	Transmission acknowledge registers	TAn bit is reset if TRSn bit is set manually, but not when TRSn is set by internal logic
CAN_MBTD	Temporary mailbox disable feature register	Allows safe access to data field of an enabled mailbox
CAN_RFHx	Remote frame handling registers	Available only to configurable mailboxes 23:8. RFH[31:24] and RFH[7:0] are read-only
CAN_MBIMx	Mailbox interrupt mask registers	Mailbox interrupts are raised only if these bits are set

Table 9-7. CAN Mailbox Control Register Mapping (Cont'd)

Register Name	Function	Notes
CAN_MBTIFx	Mailbox transmit interrupt flag registers	Can be cleared if mailbox or mailbox interrupt is disabled. Changing direction while MBTIFn = 1 results in MBRIFn = 1 and MBTIFn = 0
CAN_MBRIFx	Mailbox receive interrupt flag registers	Can be cleared if mailbox or mailbox interrupt is disabled. Changing direction while MBRIFn = 1 results in MBTIFn = 1 and MBRIFn = 0

Table 9-8. CAN Universal Counter Register Mapping

Register Name	Function	Notes
CAN_UCCNF	Universal counter mode register	Bits 15:8 and bit 4 are reserved
CAN_UCCNT	Universal counter register	Counts up or down based on universal counter mode
CAN_UCRC	Universal counter reload/capture register	In timestamp mode, holds time of last successful transmit or receive

Table 9-9. CAN Error Register Mapping

Register Name	Function	Notes
CAN_CEC	CAN error counter register	Undefined while in bus off mode, not affected by software reset
CAN_ESR	Error status register	Only the first error is stored. SA0 flag is cleared by recessive bit on CAN bus
CAN_EWR	CAN error counter warning level register	Default is 96 for each counter

Global Registers

Figure 9-11 through Figure 9-19 on page 9-48 show the CAN global registers.

CAN_CONTROL Register

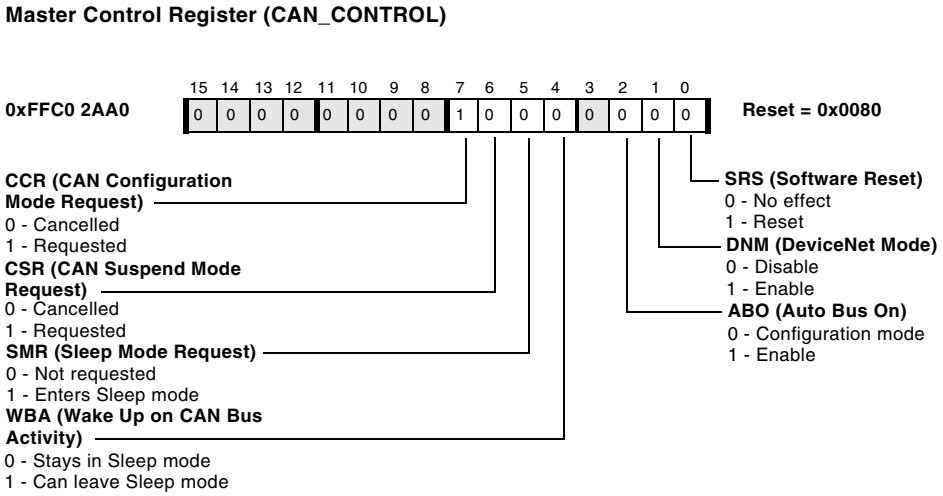


Figure 9-11. Master Control Register

CAN_STATUS Register

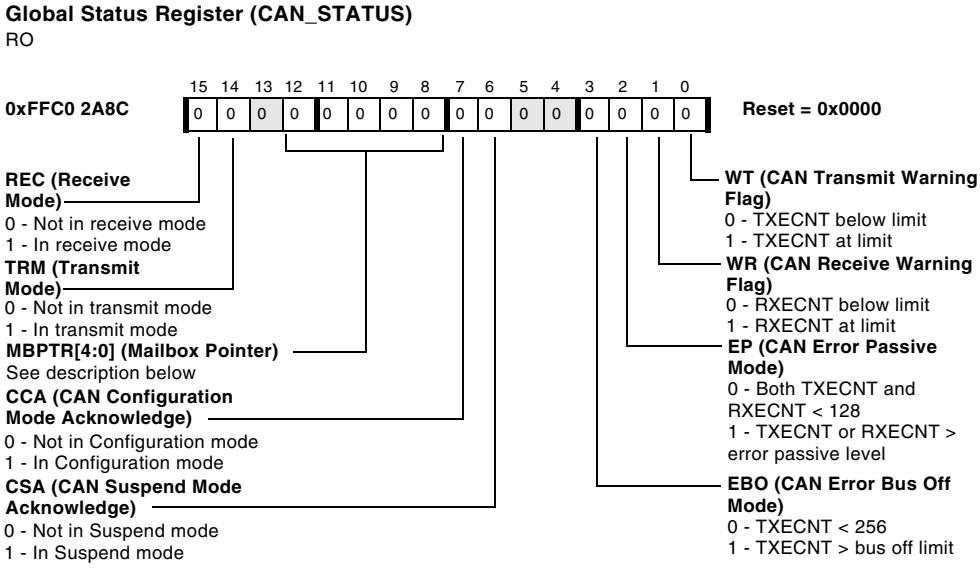


Figure 9-12. Global Status Register

- **Mail box pointer** (`MBPTR[4:0]`)

  Represents the mailbox number of the current transmit message. After a successful transmission, these bits remain unchanged.

  [11111] The message of mailbox 31 is currently being processed.

  …

  …

  …

  [00000] The message of mailbox 0 is currently being processed.

CAN_DEBUG Register

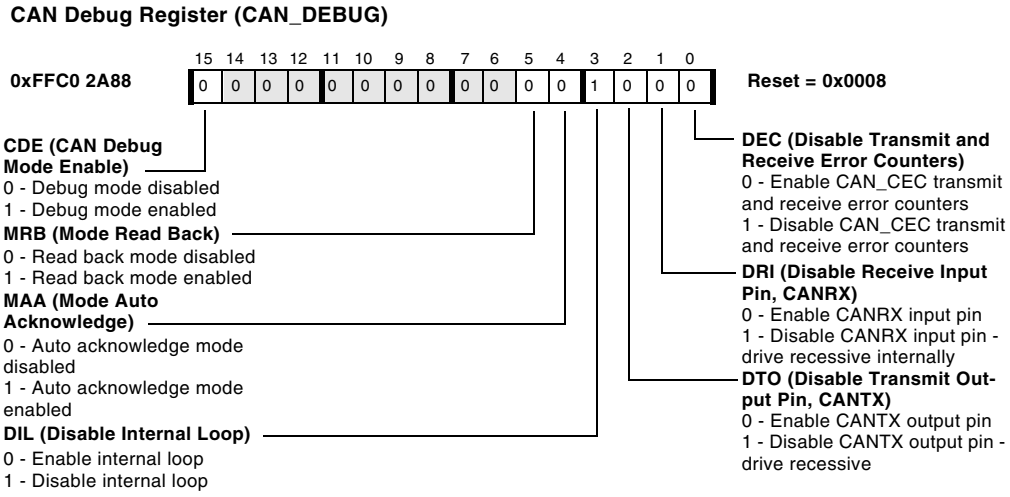


Figure 9-13. CAN Debug Register

CAN_CLOCK Register

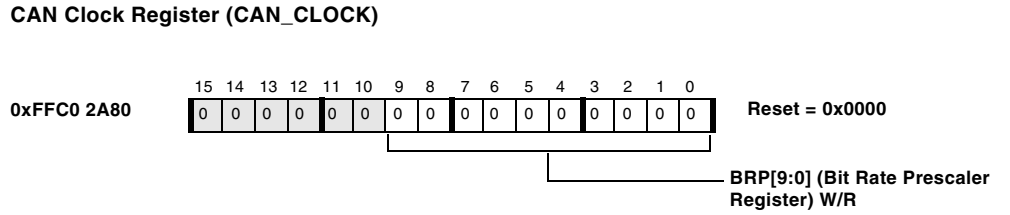


Figure 9-14. CAN Clock Register

CAN_TIMING Register

CAN Timing Register (CAN_TIMING)

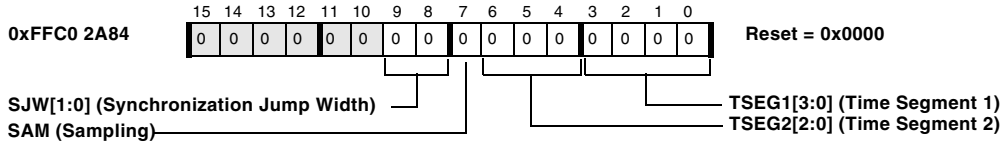


Figure 9-15. CAN Timing Register

CAN_INTR Register

CAN Interrupt Register (CAN_INTR)
RO

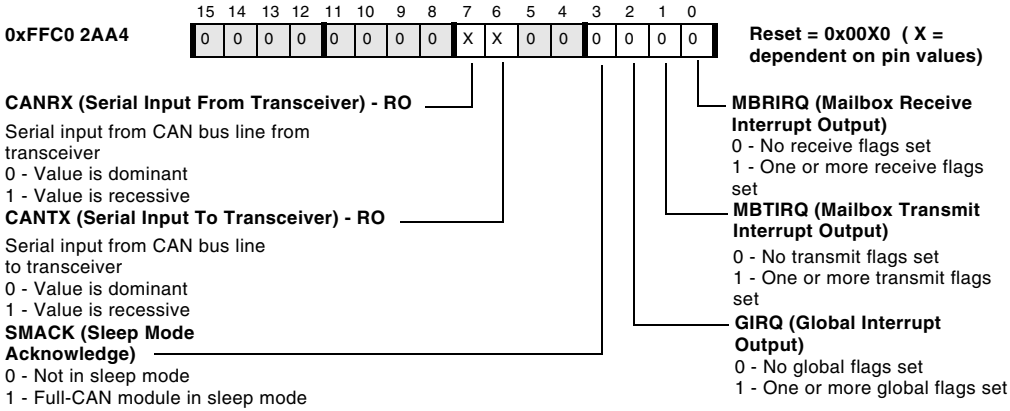


Figure 9-16. CAN Interrupt Register

CAN_GIM Register

Global Interrupt Mask Register (CAN_GIM)

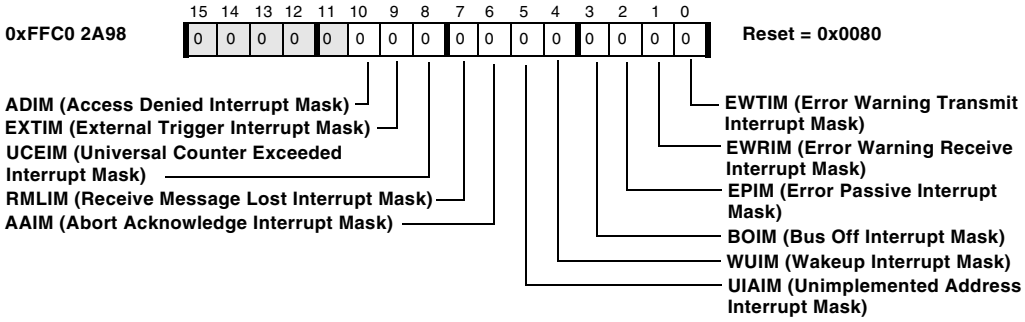


Figure 9-17. Global Interrupt Mask Register

CAN_GIS Register

Global Interrupt Status Register (CAN_GIS)
All bits are W1C

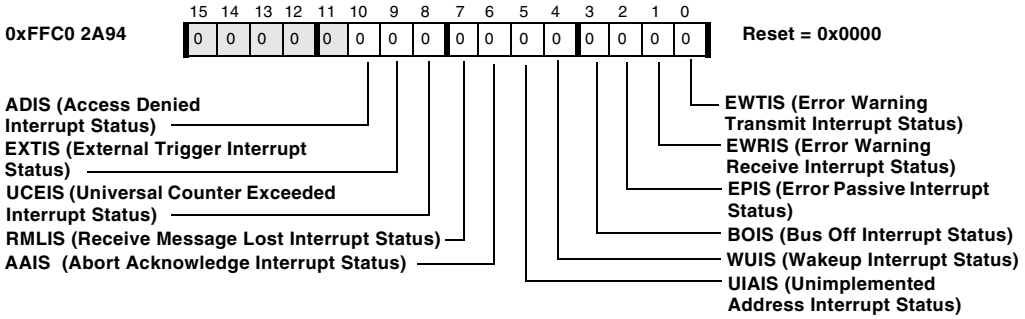


Figure 9-18. Global Interrupt Status Register

CAN_GIF Register

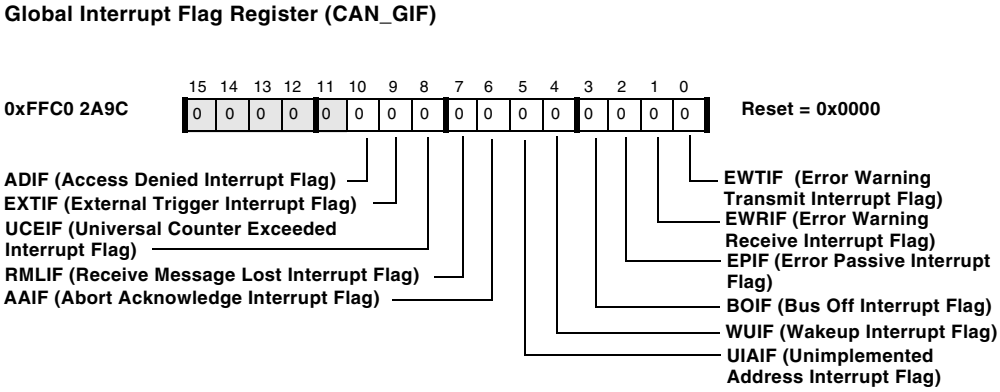


Figure 9-19. Global Interrupt Flag Register

Mailbox/Mask Registers

Figure 9-20 through Figure 9-29 on page 9-67 show the CAN mailbox and mask registers.

CAN_AMxx Registers

Acceptance Mask Register (CAN_AMxxH)

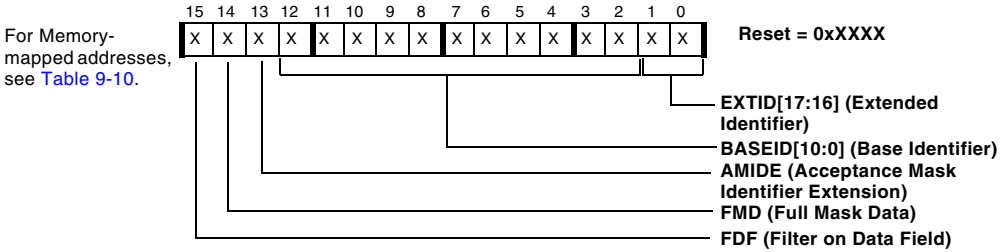


Figure 9-20. Acceptance Mask Register (H)

Table 9-10. Acceptance Mask Register (H) Memory-mapped Addresses

Register Name	Memory-mapped Address
CAN_AM00H	0xFFC0 2B04
CAN_AM01H	0xFFC0 2B0C
CAN_AM02H	0xFFC0 2B14
CAN_AM03H	0xFFC0 2B1C
CAN_AM04H	0xFFC0 2B24
CAN_AM05H	0xFFC0 2B2C
CAN_AM06H	0xFFC0 2B34
CAN_AM07H	0xFFC0 2B3C
CAN_AM08H	0xFFC0 2B44
CAN_AM09H	0xFFC0 2B4C
CAN_AM10H	0xFFC0 2B54
CAN_AM11H	0xFFC0 2B5C

Table 9-10. Acceptance Mask Register (H) Memory-mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
CAN_AM12H	0xFFC0 2B64
CAN_AM13H	0xFFC0 2B6C
CAN_AM14H	0xFFC0 2B74
CAN_AM15H	0xFFC0 2B7C
CAN_AM16H	0xFFC0 2B84
CAN_AM17H	0xFFC0 2B8C
CAN_AM18H	0xFFC0 2B94
CAN_AM19H	0xFFC0 2B9C
CAN_AM20H	0xFFC0 2BA4
CAN_AM21H	0xFFC0 2BAC
CAN_AM22H	0xFFC0 2BB4
CAN_AM23H	0xFFC0 2BBC
CAN_AM24H	0xFFC0 2BC4
CAN_AM25H	0xFFC0 2BCC
CAN_AM26H	0xFFC0 2BD4
CAN_AM27H	0xFFC0 2BDC
CAN_AM28H	0xFFC0 2BE4
CAN_AM29H	0xFFC0 2BEC
CAN_AM30H	0xFFC0 2BF4
CAN_AM31H	0xFFC0 2BFC

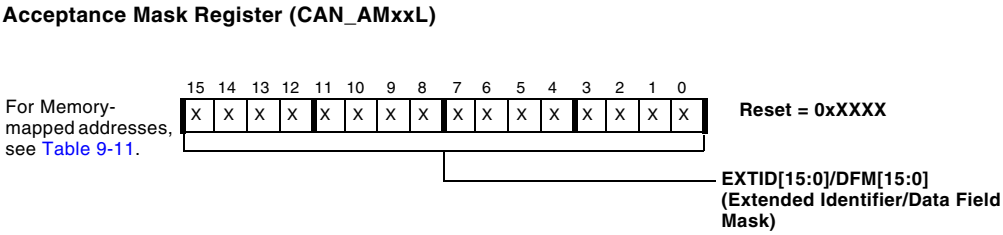


Figure 9-21. Acceptance Mask Register (L)

Table 9-11. Acceptance Mask Register (L) Memory-mapped Addresses

Register Name	Memory-mapped Address
CAN_AM00L	0xFFC0 2B00
CAN_AM01L	0xFFC0 2B08
CAN_AM02L	0xFFC0 2B10
CAN_AM03L	0xFFC0 2B18
CAN_AM04L	0xFFC0 2B20
CAN_AM05L	0xFFC0 2B28
CAN_AM06L	0xFFC0 2B30
CAN_AM07L	0xFFC0 2B38
CAN_AM08L	0xFFC0 2B40
CAN_AM09L	0xFFC0 2B48
CAN_AM10L	0xFFC0 2B50
CAN_AM11L	0xFFC0 2B58
CAN_AM12L	0xFFC0 2B60
CAN_AM13L	0xFFC0 2B68
CAN_AM14L	0xFFC0 2B70
CAN_AM15L	0xFFC0 2B78

Table 9-11. Acceptance Mask Register (L) Memory-mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
CAN_AM16L	0xFFC0 2B80
CAN_AM17L	0xFFC0 2B88
CAN_AM18L	0xFFC0 2B90
CAN_AM19L	0xFFC0 2B98
CAN_AM20L	0xFFC0 2BA0
CAN_AM21L	0xFFC0 2BA8
CAN_AM22L	0xFFC0 2BB0
CAN_AM23L	0xFFC0 2BB8
CAN_AM24L	0xFFC0 2BC0
CAN_AM25L	0xFFC0 2BC8
CAN_AM26L	0xFFC0 2BD0
CAN_AM27L	0xFFC0 2BD8
CAN_AM28L	0xFFC0 2BE0
CAN_AM29L	0xFFC0 2BE8
CAN_AM30L	0xFFC0 2BF0
CAN_AM31L	0xFFC0 2BF8

CAN_MBxx_ID1 Registers

Mailbox Word 7 Register (CAN_MBxx_ID1)

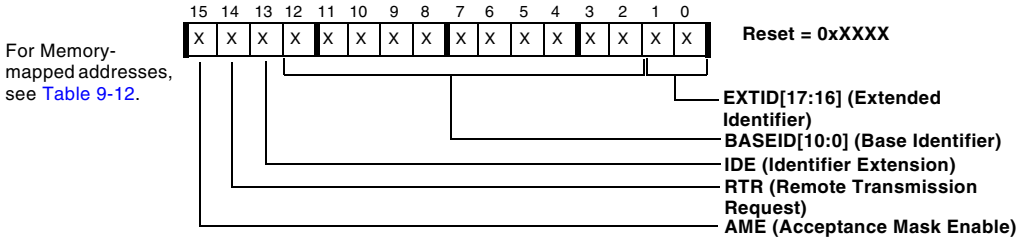


Figure 9-22. Mailbox Word 7 Register

Table 9-12. Mailbox Word 7 Register Memory-mapped Addresses

Register Name	Memory-mapped Address
CAN_MB00_ID1	0xFFC0 2C1C
CAN_MB01_ID1	0xFFC0 2C3C
CAN_MB02_ID1	0xFFC0 2C5C
CAN_MB03_ID1	0xFFC0 2C7C
CAN_MB04_ID1	0xFFC0 2C9C
CAN_MB05_ID1	0xFFC0 2CBC
CAN_MB06_ID1	0xFFC0 2CDC
CAN_MB07_ID1	0xFFC0 2CFC
CAN_MB08_ID1	0xFFC0 2D1C
CAN_MB09_ID1	0xFFC0 2D3C
CAN_MB10_ID1	0xFFC0 2D5C
CAN_MB11_ID1	0xFFC0 2D7C
CAN_MB12_ID1	0xFFC0 2D9C

Table 9-12. Mailbox Word 7 Register Memory-mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
CAN_MB13_ID1	0xFFC0 2DBC
CAN_MB14_ID1	0xFFC0 2DDC
CAN_MB15_ID1	0xFFC0 2DFC
CAN_MB16_ID1	0xFFC0 2E1C
CAN_MB17_ID1	0xFFC0 2E3C
CAN_MB18_ID1	0xFFC0 2E5C
CAN_MB19_ID1	0xFFC0 2E7C
CAN_MB20_ID1	0xFFC0 2E9C
CAN_MB21_ID1	0xFFC0 2EBC
CAN_MB22_ID1	0xFFC0 2EDC
CAN_MB23_ID1	0xFFC0 2EFC
CAN_MB24_ID1	0xFFC0 2F1C
CAN_MB25_ID1	0xFFC0 2F3C
CAN_MB26_ID1	0xFFC0 2F5C
CAN_MB27_ID1	0xFFC0 2F7C
CAN_MB28_ID1	0xFFC0 2F9C
CAN_MB29_ID1	0xFFC0 2FBC
CAN_MB30_ID1	0xFFC0 2FDC
CAN_MB31_ID1	0xFFC0 2FFC

CAN_MBxx_ID0 Registers

Mailbox Word 6 Register (CAN_MBxx_ID0)

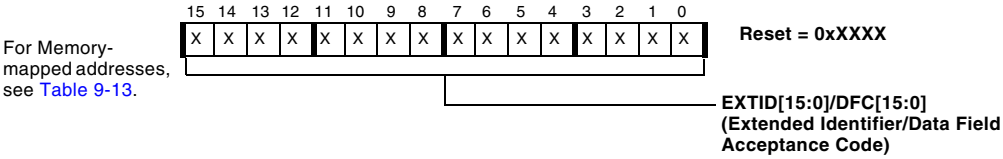


Figure 9-23. Mailbox Word 6 Register

Table 9-13. Mailbox Word 6 Register Memory-mapped Addresses

Register Name	Memory-mapped Address
CAN_MB00_ID0	0xFFC0 2C18
CAN_MB01_ID0	0xFFC0 2C38
CAN_MB02_ID0	0xFFC0 2C58
CAN_MB03_ID0	0xFFC0 2C78
CAN_MB04_ID0	0xFFC0 2C98
CAN_MB05_ID0	0xFFC0 2CB8
CAN_MB06_ID0	0xFFC0 2CD8
CAN_MB07_ID0	0xFFC0 2CF8
CAN_MB08_ID0	0xFFC0 2D18
CAN_MB09_ID0	0xFFC0 2D38
CAN_MB10_ID0	0xFFC0 2D58
CAN_MB11_ID0	0xFFC0 2D78
CAN_MB12_ID0	0xFFC0 2D98
CAN_MB13_ID0	0xFFC0 2DB8
CAN_MB14_ID0	0xFFC0 2DD8

Table 9-13. Mailbox Word 6 Register Memory-mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
CAN_MB15_ID0	0xFFC0 2DF8
CAN_MB16_ID0	0xFFC0 2E18
CAN_MB17_ID0	0xFFC0 2E38
CAN_MB18_ID0	0xFFC0 2E58
CAN_MB19_ID0	0xFFC0 2E78
CAN_MB20_ID0	0xFFC0 2E98
CAN_MB21_ID0	0xFFC0 2EB8
CAN_MB22_ID0	0xFFC0 2ED8
CAN_MB23_ID0	0xFFC0 2EF8
CAN_MB24_ID0	0xFFC0 2F18
CAN_MB25_ID0	0xFFC0 2F38
CAN_MB26_ID0	0xFFC0 2F58
CAN_MB27_ID0	0xFFC0 2F78
CAN_MB28_ID0	0xFFC0 2F98
CAN_MB29_ID0	0xFFC0 2FB8
CAN_MB30_ID0	0xFFC0 2FD8
CAN_MB31_ID0	0xFFC0 2FF8

CAN_MBxx_TIMESTAMP Registers

Mailbox Word 5 Register (CAN_MBxx_TIMESTAMP)

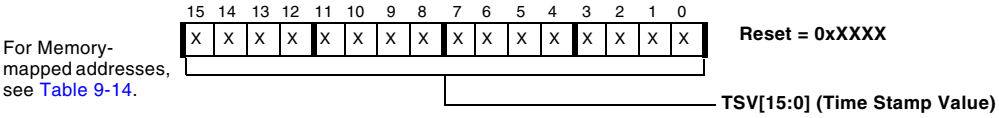


Figure 9-24. Mailbox Word 5 Register

Table 9-14. Mailbox Word 5 Register Memory-mapped Addresses

Register Name	Memory-mapped Address
CAN_MB00_TIMESTAMP	0xFFC0 2C14
CAN_MB01_TIMESTAMP	0xFFC0 2C34
CAN_MB02_TIMESTAMP	0xFFC0 2C54
CAN_MB03_TIMESTAMP	0xFFC0 2C74
CAN_MB04_TIMESTAMP	0xFFC0 2C94
CAN_MB05_TIMESTAMP	0xFFC0 2CB4
CAN_MB06_TIMESTAMP	0xFFC0 2CD4
CAN_MB07_TIMESTAMP	0xFFC0 2CF4
CAN_MB08_TIMESTAMP	0xFFC0 2D14
CAN_MB09_TIMESTAMP	0xFFC0 2D34
CAN_MB10_TIMESTAMP	0xFFC0 2D54
CAN_MB11_TIMESTAMP	0xFFC0 2D74
CAN_MB12_TIMESTAMP	0xFFC0 2D94
CAN_MB13_TIMESTAMP	0xFFC0 2DB4
CAN_MB14_TIMESTAMP	0xFFC0 2DD4

Table 9-14. Mailbox Word 5 Register Memory-mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
CAN_MB15_TIMESTAMP	0xFFC0 2DF4
CAN_MB16_TIMESTAMP	0xFFC0 2E14
CAN_MB17_TIMESTAMP	0xFFC0 2E34
CAN_MB18_TIMESTAMP	0xFFC0 2E54
CAN_MB19_TIMESTAMP	0xFFC0 2E74
CAN_MB20_TIMESTAMP	0xFFC0 2E94
CAN_MB21_TIMESTAMP	0xFFC0 2EB4
CAN_MB22_TIMESTAMP	0xFFC0 2ED4
CAN_MB23_TIMESTAMP	0xFFC0 2EF4
CAN_MB24_TIMESTAMP	0xFFC0 2F14
CAN_MB25_TIMESTAMP	0xFFC0 2F34
CAN_MB26_TIMESTAMP	0xFFC0 2F54
CAN_MB27_TIMESTAMP	0xFFC0 2F74
CAN_MB28_TIMESTAMP	0xFFC0 2F94
CAN_MB29_TIMESTAMP	0xFFC0 2FB4
CAN_MB30_TIMESTAMP	0xFFC0 2FD4
CAN_MB31_TIMESTAMP	0xFFC0 2FF4

CAN_MBxx_LENGTH Registers

Mailbox Word 4 Register (CAN_MBxx_LENGTH)

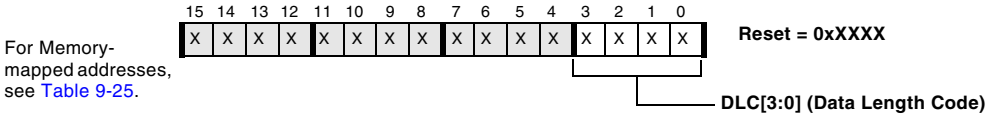


Figure 9-25. Mailbox Word 4 Register

Table 9-15. Mailbox Word 4 Register Memory-mapped Addresses

Register Name	Memory-mapped Address
CAN_MB00_LENGTH	0xFFC0 2C10
CAN_MB01_LENGTH	0xFFC0 2C30
CAN_MB02_LENGTH	0xFFC0 2C50
CAN_MB03_LENGTH	0xFFC0 2C70
CAN_MB04_LENGTH	0xFFC0 2C90
CAN_MB05_LENGTH	0xFFC0 2CB0
CAN_MB06_LENGTH	0xFFC0 2CD0
CAN_MB07_LENGTH	0xFFC0 2CF0
CAN_MB08_LENGTH	0xFFC0 2D10
CAN_MB09_LENGTH	0xFFC0 2D30
CAN_MB10_LENGTH	0xFFC0 2D50
CAN_MB11_LENGTH	0xFFC0 2D70
CAN_MB12_LENGTH	0xFFC0 2D90
CAN_MB13_LENGTH	0xFFC0 2DB0
CAN_MB14_LENGTH	0xFFC0 2DD0

Table 9-15. Mailbox Word 4 Register Memory-mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
CAN_MB15_LENGTH	0xFFC0 2DF0
CAN_MB16_LENGTH	0xFFC0 2E10
CAN_MB17_LENGTH	0xFFC0 2E30
CAN_MB18_LENGTH	0xFFC0 2E50
CAN_MB19_LENGTH	0xFFC0 2E70
CAN_MB20_LENGTH	0xFFC0 2E90
CAN_MB21_LENGTH	0xFFC0 2EB0
CAN_MB22_LENGTH	0xFFC0 2ED0
CAN_MB23_LENGTH	0xFFC0 2EF0
CAN_MB24_LENGTH	0xFFC0 2F10
CAN_MB25_LENGTH	0xFFC0 2F30
CAN_MB26_LENGTH	0xFFC0 2F50
CAN_MB27_LENGTH	0xFFC0 2F70
CAN_MB28_LENGTH	0xFFC0 2F90
CAN_MB29_LENGTH	0xFFC0 2FB0
CAN_MB30_LENGTH	0xFFC0 2FD0
CAN_MB31_LENGTH	0xFFC0 2FF0

CAN_MBxx_DATAx Registers

Mailbox Word 3 Register (CAN_MBxx_DATA3)

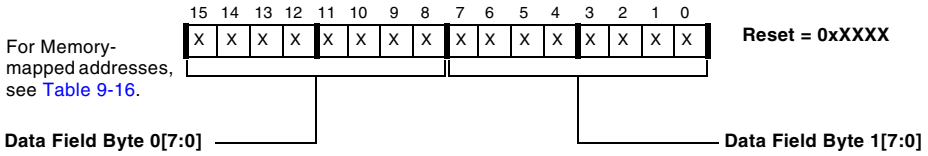


Figure 9-26. Mailbox Word 3 Register

Table 9-16. Mailbox Word 3 Register Memory-mapped Addresses

Register Name	Memory-mapped Address
CAN_MB00_DATA3	0xFFC0 2C0C
CAN_MB01_DATA3	0xFFC0 2C2C
CAN_MB02_DATA3	0xFFC0 2C4C
CAN_MB03_DATA3	0xFFC0 2C6C
CAN_MB04_DATA3	0xFFC0 2C8C
CAN_MB05_DATA3	0xFFC0 2CAC
CAN_MB06_DATA3	0xFFC0 2CCC
CAN_MB07_DATA3	0xFFC0 2CEC
CAN_MB08_DATA3	0xFFC0 2D0C
CAN_MB09_DATA3	0xFFC0 2D2C
CAN_MB10_DATA3	0xFFC0 2D4C
CAN_MB11_DATA3	0xFFC0 2D6C
CAN_MB12_DATA3	0xFFC0 2D8C
CAN_MB13_DATA3	0xFFC0 2DAC

Table 9-16. Mailbox Word 3 Register Memory-mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
CAN_MB14_DATA3	0xFFC0 2DCC
CAN_MB15_DATA3	0xFFC0 2DEC
CAN_MB16_DATA3	0xFFC0 2E0C
CAN_MB17_DATA3	0xFFC0 2E2C
CAN_MB18_DATA3	0xFFC0 2E4C
CAN_MB19_DATA3	0xFFC0 2E6C
CAN_MB20_DATA3	0xFFC0 2E8C
CAN_MB21_DATA3	0xFFC0 2EAC
CAN_MB22_DATA3	0xFFC0 2ECC
CAN_MB23_DATA3	0xFFC0 2EEC
CAN_MB24_DATA3	0xFFC0 2F0C
CAN_MB25_DATA3	0xFFC0 2F2C
CAN_MB26_DATA3	0xFFC0 2F4C
CAN_MB27_DATA3	0xFFC0 2F6C
CAN_MB28_DATA3	0xFFC0 2F8C
CAN_MB29_DATA3	0xFFC0 2FAC
CAN_MB30_DATA3	0xFFC0 2FCC
CAN_MB31_DATA3	0xFFC0 2FEC

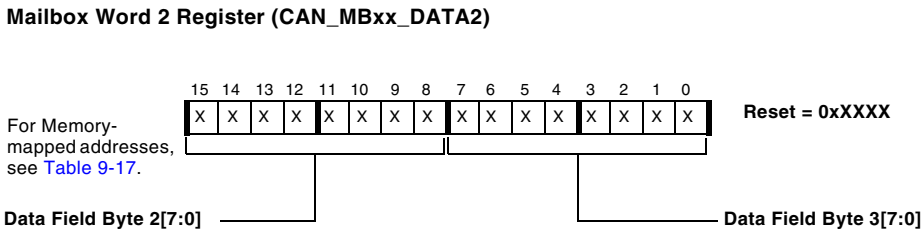


Figure 9-27. Mailbox Word 2 Register

Table 9-17. Mailbox Word 2 Register Memory-mapped Addresses

Register Name	Memory-mapped Address
CAN_MB00_DATA2	0xFFC0 2C08
CAN_MB01_DATA2	0xFFC0 2C28
CAN_MB02_DATA2	0xFFC0 2C48
CAN_MB03_DATA2	0xFFC0 2C68
CAN_MB04_DATA2	0xFFC0 2C88
CAN_MB05_DATA2	0xFFC0 2CA8
CAN_MB06_DATA2	0xFFC0 2CC8
CAN_MB07_DATA2	0xFFC0 2CE8
CAN_MB08_DATA2	0xFFC0 2D08
CAN_MB09_DATA2	0xFFC0 2D28
CAN_MB10_DATA2	0xFFC0 2D48
CAN_MB11_DATA2	0xFFC0 2D68
CAN_MB12_DATA2	0xFFC0 2D88
CAN_MB13_DATA2	0xFFC0 2DA8
CAN_MB14_DATA2	0xFFC0 2DC8
CAN_MB15_DATA2	0xFFC0 2DE8

Table 9-17. Mailbox Word 2 Register Memory-mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
CAN_MB16_DATA2	0xFFC0 2E08
CAN_MB17_DATA2	0xFFC0 2E28
CAN_MB18_DATA2	0xFFC0 2E48
CAN_MB19_DATA2	0xFFC0 2E68
CAN_MB20_DATA2	0xFFC0 2E88
CAN_MB21_DATA2	0xFFC0 2EA8
CAN_MB22_DATA2	0xFFC0 2EC8
CAN_MB23_DATA2	0xFFC0 2EE8
CAN_MB24_DATA2	0xFFC0 2F08
CAN_MB25_DATA2	0xFFC0 2F28
CAN_MB26_DATA2	0xFFC0 2F48
CAN_MB27_DATA2	0xFFC0 2F68
CAN_MB28_DATA2	0xFFC0 2F88
CAN_MB29_DATA2	0xFFC0 2FA8
CAN_MB30_DATA2	0xFFC0 2FC8
CAN_MB31_DATA2	0xFFC0 2FE8

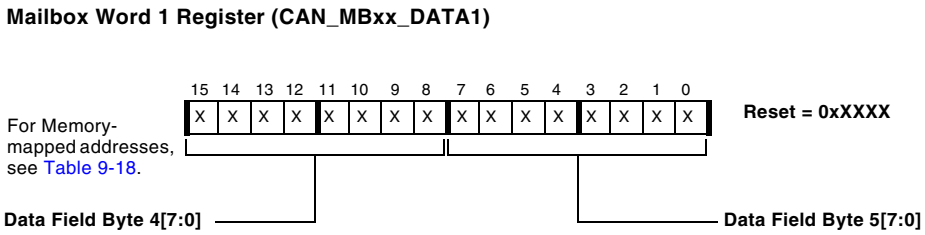


Figure 9-28. Mailbox Word 1 Register

Table 9-18. Mailbox Word 1 Register Memory-mapped Addresses

Register Name	Memory-mapped Address
CAN_MB00_DATA1	0xFFC0 2C04
CAN_MB01_DATA1	0xFFC0 2C24
CAN_MB02_DATA1	0xFFC0 2C44
CAN_MB03_DATA1	0xFFC0 2C64
CAN_MB04_DATA1	0xFFC0 2C84
CAN_MB05_DATA1	0xFFC0 2CA4
CAN_MB06_DATA1	0xFFC0 2CC4
CAN_MB07_DATA1	0xFFC0 2CE4
CAN_MB08_DATA1	0xFFC0 2D04
CAN_MB09_DATA1	0xFFC0 2D24
CAN_MB10_DATA1	0xFFC0 2D44
CAN_MB11_DATA1	0xFFC0 2D64
CAN_MB12_DATA1	0xFFC0 2D84
CAN_MB13_DATA1	0xFFC0 2DA4
CAN_MB14_DATA1	0xFFC0 2DC4
CAN_MB15_DATA1	0xFFC0 2DE4

Table 9-18. Mailbox Word 1 Register Memory-mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
CAN_MB16_DATA1	0xFFC0 2E04
CAN_MB17_DATA1	0xFFC0 2E24
CAN_MB18_DATA1	0xFFC0 2E44
CAN_MB19_DATA1	0xFFC0 2E64
CAN_MB20_DATA1	0xFFC0 2E84
CAN_MB21_DATA1	0xFFC0 2EA4
CAN_MB22_DATA1	0xFFC0 2EC4
CAN_MB23_DATA1	0xFFC0 2EE4
CAN_MB24_DATA1	0xFFC0 2F04
CAN_MB25_DATA1	0xFFC0 2F24
CAN_MB26_DATA1	0xFFC0 2F44
CAN_MB27_DATA1	0xFFC0 2F64
CAN_MB28_DATA1	0xFFC0 2F84
CAN_MB29_DATA1	0xFFC0 2FA4
CAN_MB30_DATA1	0xFFC0 2FC4
CAN_MB31_DATA1	0xFFC0 2FE4

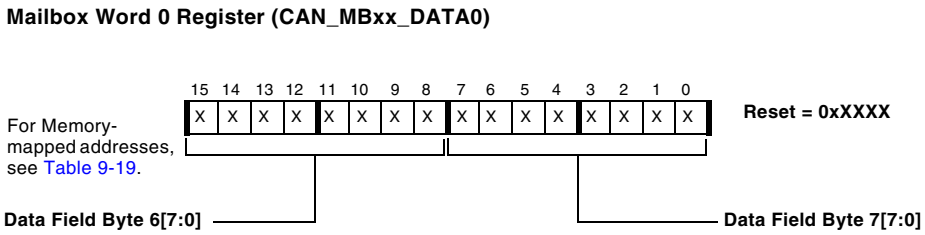


Figure 9-29. Mailbox Word 0 Register

Table 9-19. Mailbox Word 0 Register Memory-mapped Addresses

Register Name	Memory-mapped Address
CAN_MB00_DATA0	0xFFC0 2C00
CAN_MB01_DATA0	0xFFC0 2C20
CAN_MB02_DATA0	0xFFC0 2C40
CAN_MB03_DATA0	0xFFC0 2C60
CAN_MB04_DATA0	0xFFC0 2C80
CAN_MB05_DATA0	0xFFC0 2CA0
CAN_MB06_DATA0	0xFFC0 2CC0
CAN_MB07_DATA0	0xFFC0 2CE0
CAN_MB08_DATA0	0xFFC0 2D00
CAN_MB09_DATA0	0xFFC0 2D20
CAN_MB10_DATA0	0xFFC0 2D40
CAN_MB11_DATA0	0xFFC0 2D60
CAN_MB12_DATA0	0xFFC0 2D80
CAN_MB13_DATA0	0xFFC0 2DA0
CAN_MB14_DATA0	0xFFC0 2DC0
CAN_MB15_DATA0	0xFFC0 2DE0

Table 9-19. Mailbox Word 0 Register Memory-mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
CAN_MB16_DATA0	0xFFC0 2E00
CAN_MB17_DATA0	0xFFC0 2E20
CAN_MB18_DATA0	0xFFC0 2E40
CAN_MB19_DATA0	0xFFC0 2E60
CAN_MB20_DATA0	0xFFC0 2E80
CAN_MB21_DATA0	0xFFC0 2EA0
CAN_MB22_DATA0	0xFFC0 2EC0
CAN_MB23_DATA0	0xFFC0 2EE0
CAN_MB24_DATA0	0xFFC0 2F00
CAN_MB25_DATA0	0xFFC0 2F20
CAN_MB26_DATA0	0xFFC0 2F40
CAN_MB27_DATA0	0xFFC0 2F60
CAN_MB28_DATA0	0xFFC0 2F80
CAN_MB29_DATA0	0xFFC0 2FA0
CAN_MB30_DATA0	0xFFC0 2FC0
CAN_MB31_DATA0	0xFFC0 2FE0

Mailbox Control Registers

Figure 9-30 through Figure 9-56 on page 9-82 show the mailbox control registers.

CAN_MCx Registers

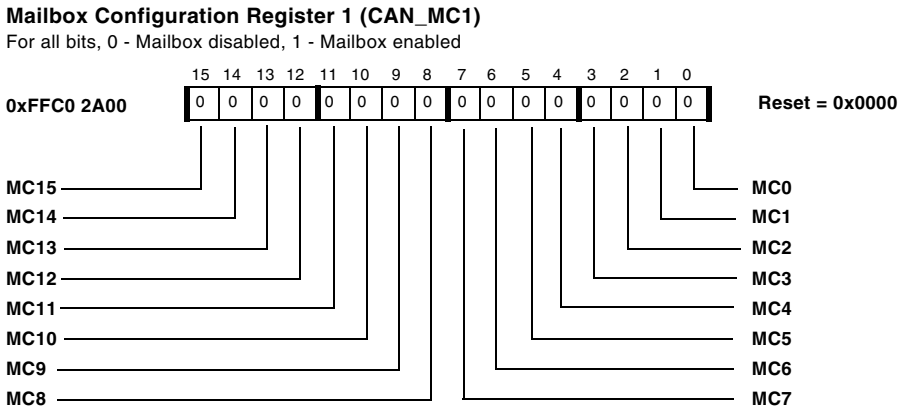


Figure 9-30. Mailbox Configuration Register 1

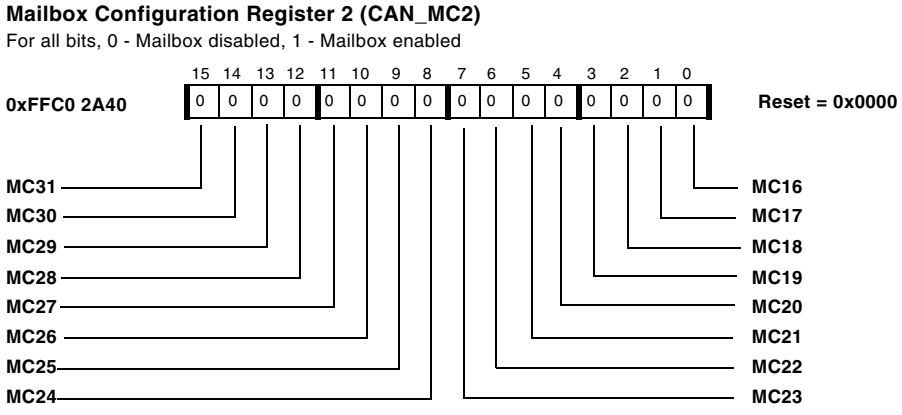


Figure 9-31. Mailbox Configuration Register 2

CAN_MDx Registers

Mailbox Direction Register 1 (CAN_MD1)

For all bits, 0 - Mailbox configured as transmit mode, 1 - Mailbox configured as receive mode

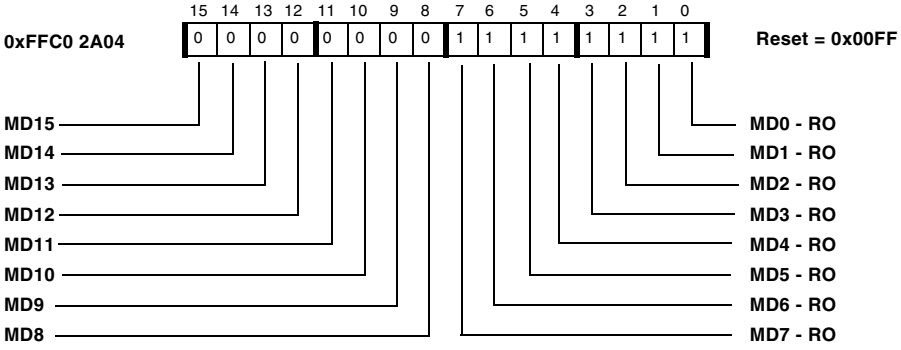


Figure 9-32. Mailbox Direction Register 1

Mailbox Direction Register 2 (CAN_MD2)

For all bits, 0 - Mailbox configured as transmit mode, 1 - Mailbox configured as receive mode

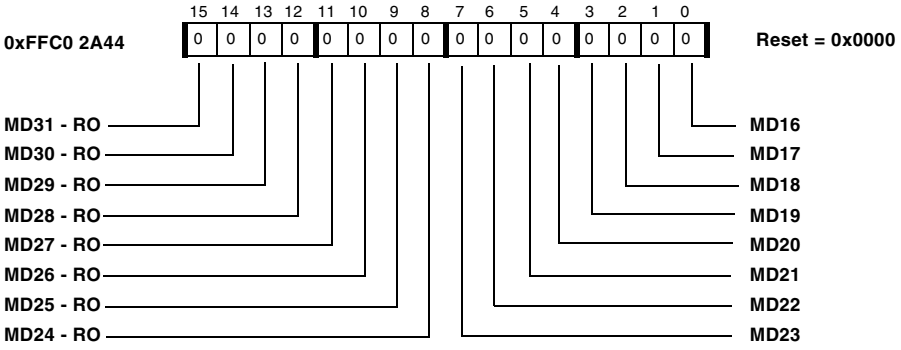


Figure 9-33. Mailbox Direction Register 2

CAN_RMPx Register

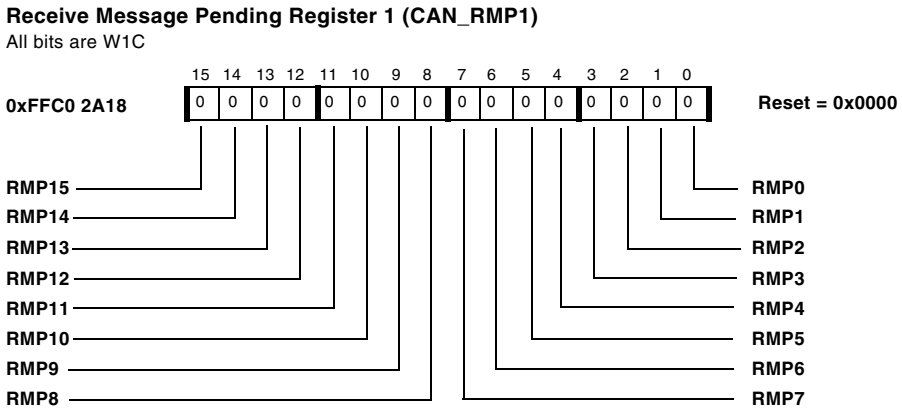


Figure 9-34. Receive Message Pending Register 1

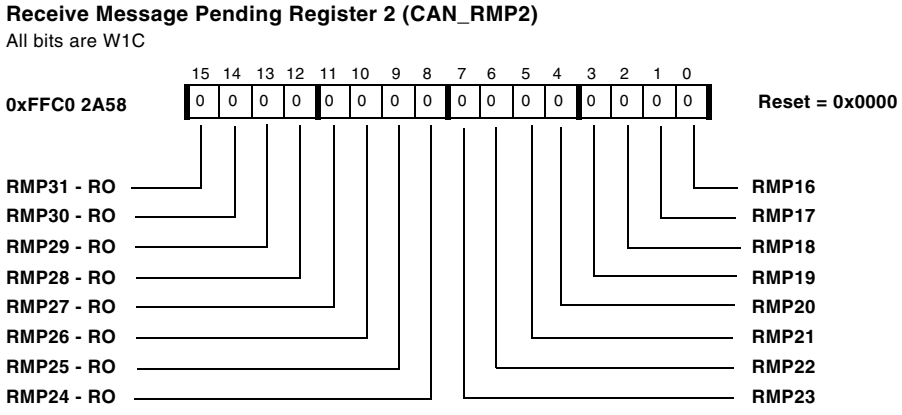


Figure 9-35. Receive Message Pending Register 2

CAN_RMLx Register

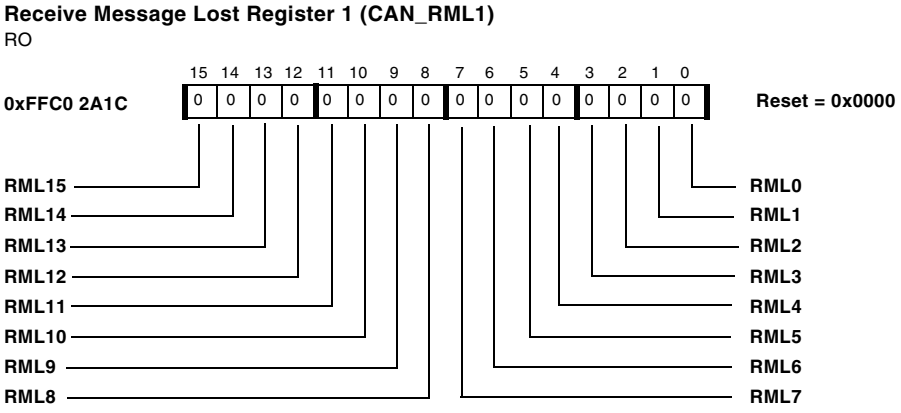


Figure 9-36. Receive Message Lost Register 1

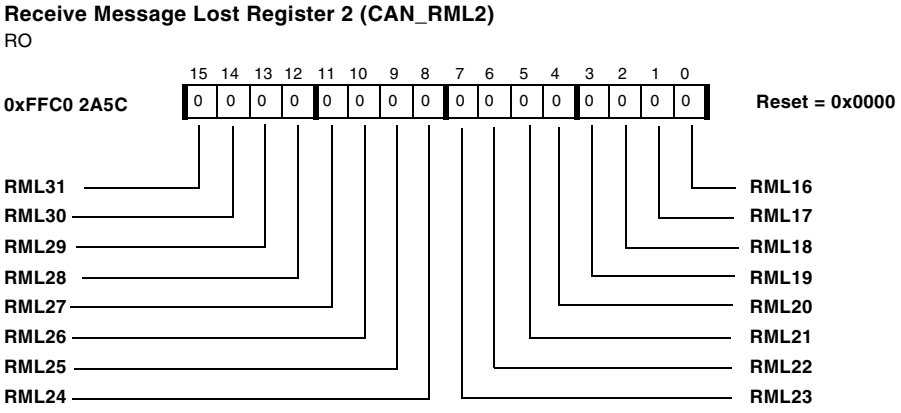


Figure 9-37. Receive Message Lost Register 2

CAN_OPSSx Register

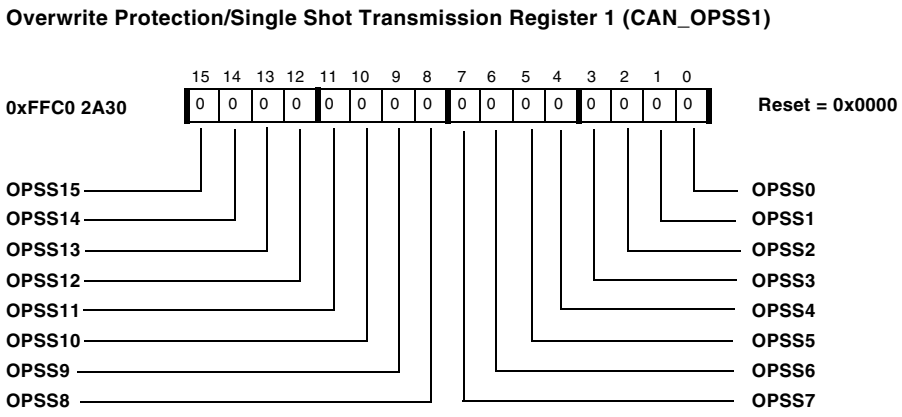


Figure 9-38. Overwrite Protection/Single Shot Transmission Register 1

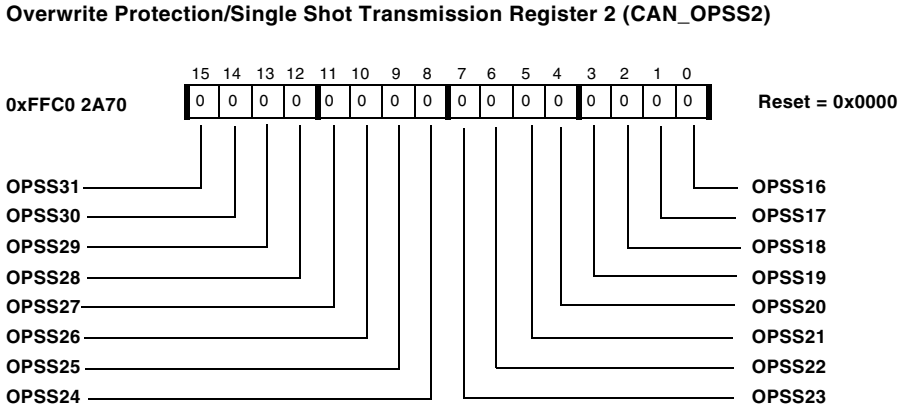


Figure 9-39. Overwrite Protection/Single Shot Transmission Register 2

CAN_TRSx Registers

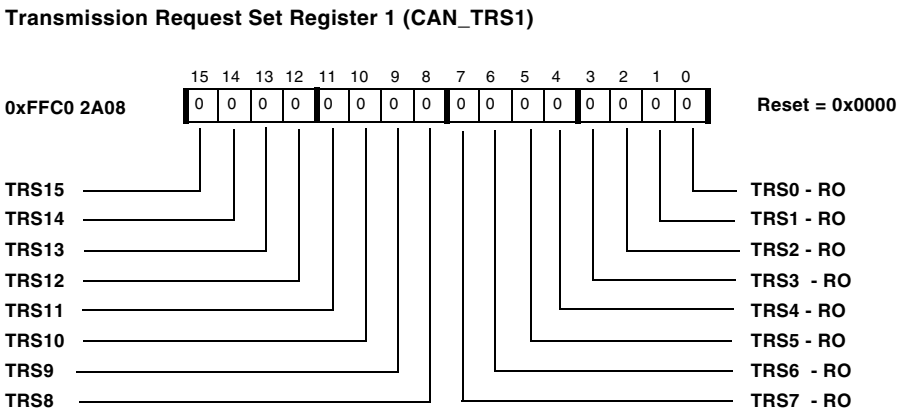


Figure 9-40. Transmission Request Set Register 1

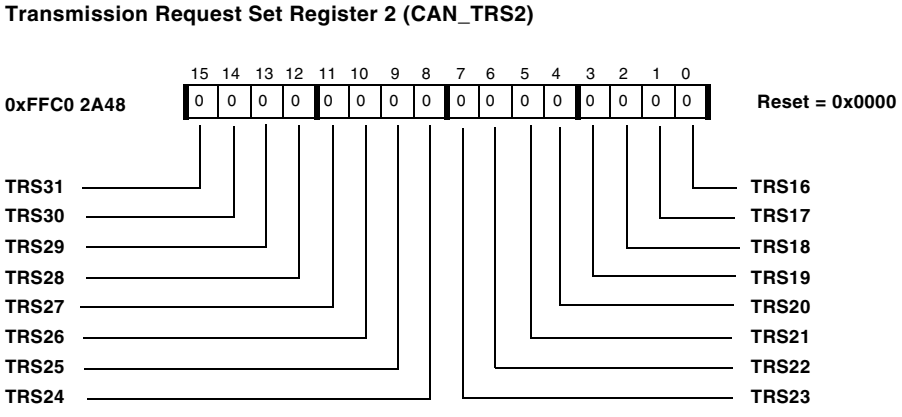


Figure 9-41. Transmission Request Set Register 2

CAN_TRRx Registers

Transmission Request Reset Register 1 (CAN_TRR1)

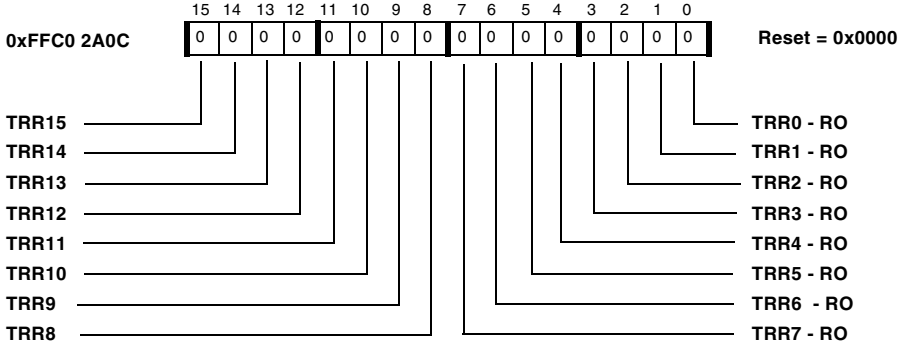


Figure 9-42. Transmission Request Reset Register 1

Transmission Request Reset Register 2 (CAN_TRR2)

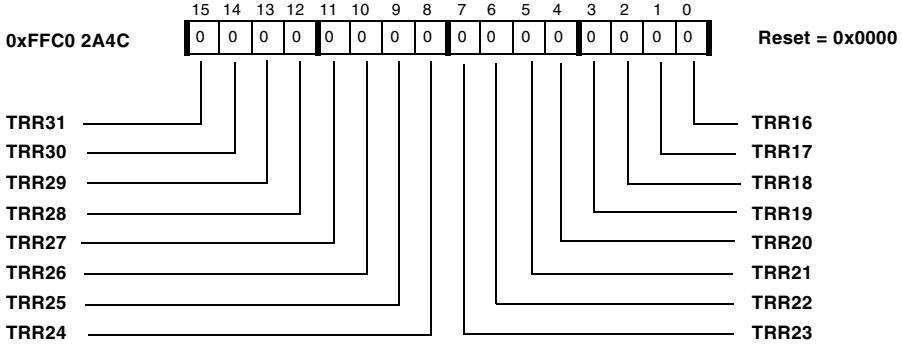


Figure 9-43. Transmission Request Reset Register 2

CAN_AAx Register

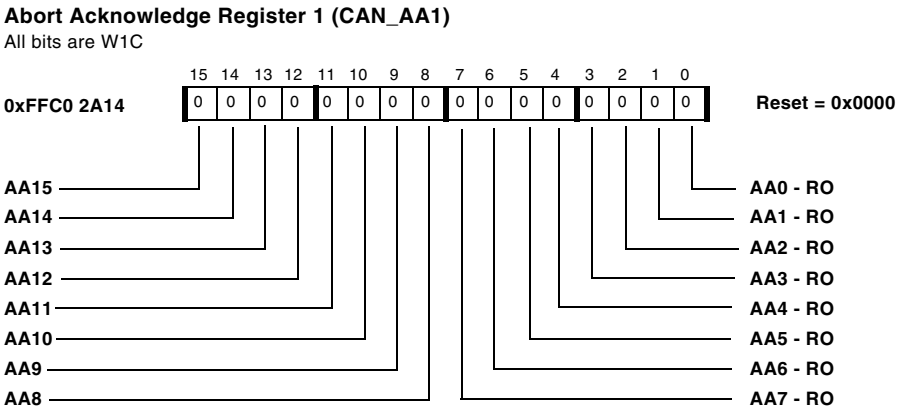


Figure 9-44. Abort Acknowledge Register 1

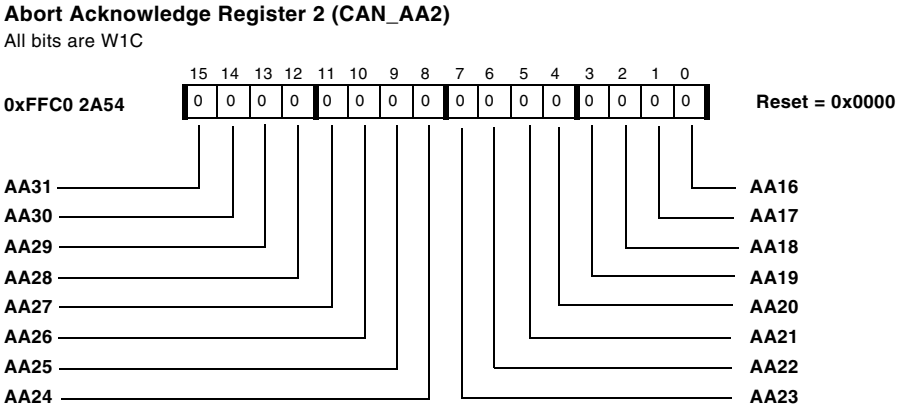


Figure 9-45. Abort Acknowledge Register 2

CAN_TAx Register

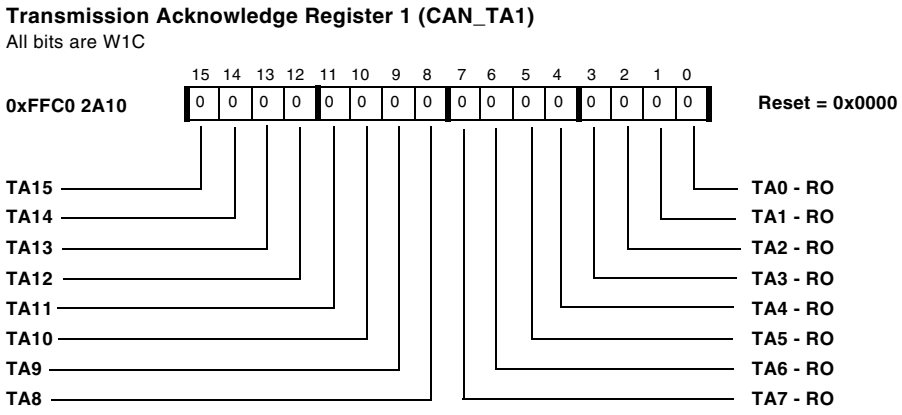


Figure 9-46. Transmission Acknowledge Register 1

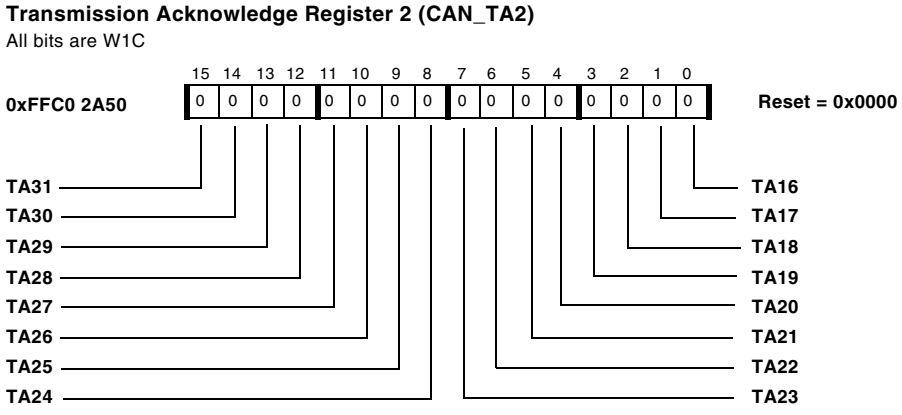


Figure 9-47. Transmission Acknowledge Register 2

CAN_MBTD Register

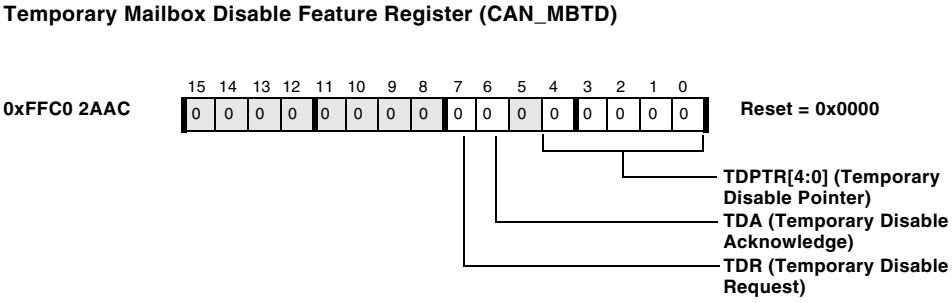


Figure 9-48. Temporary Mailbox Disable Register

CAN_RFHx Registers

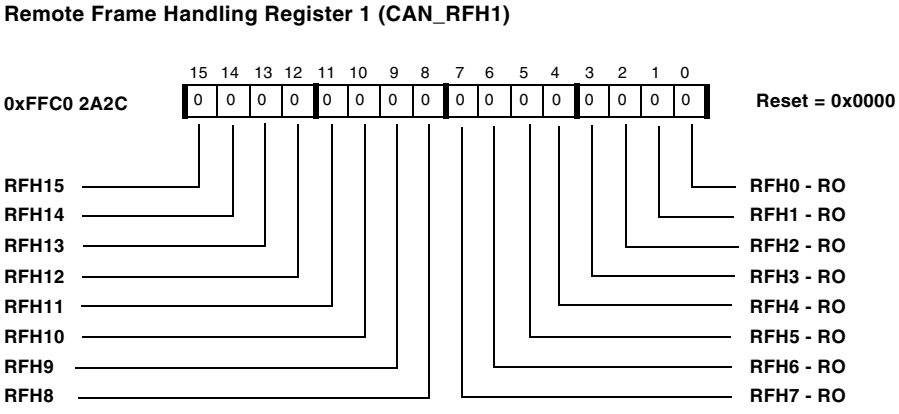


Figure 9-49. Remote Frame Handling Register 1

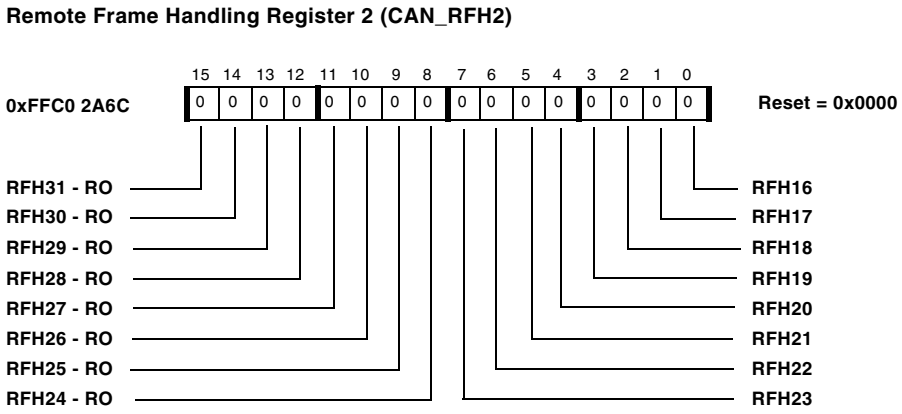


Figure 9-50. Remote Frame Handling Register 2

CAN_MBIMx Registers

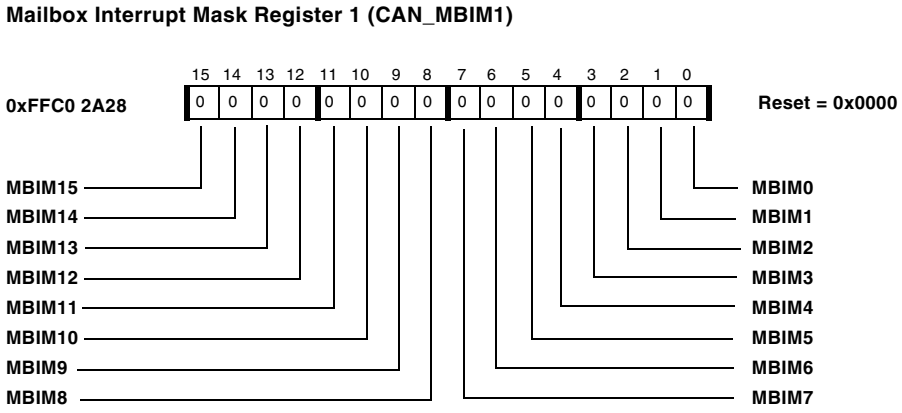


Figure 9-51. Mailbox Interrupt Mask Register 1

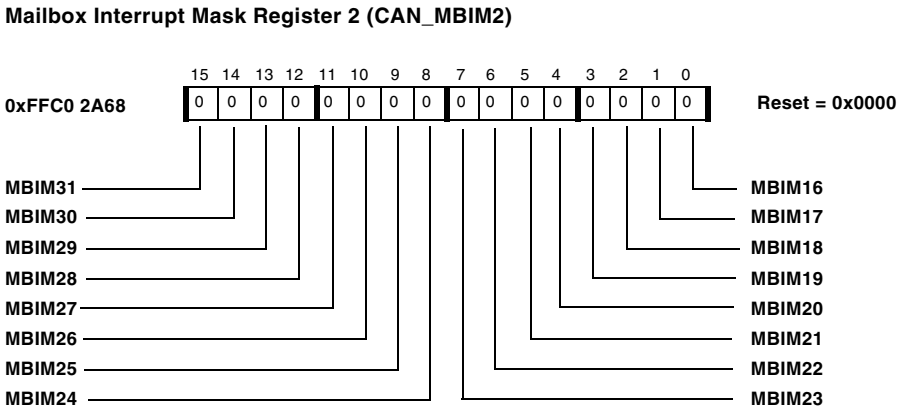


Figure 9-52. Mailbox Interrupt Mask Register 2

CAN_MBTIFx Registers

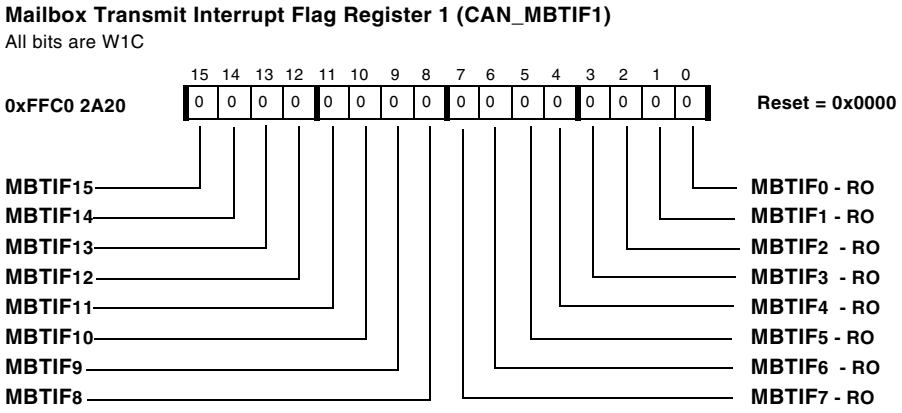


Figure 9-53. Mailbox Transmit Interrupt Flag Register 1

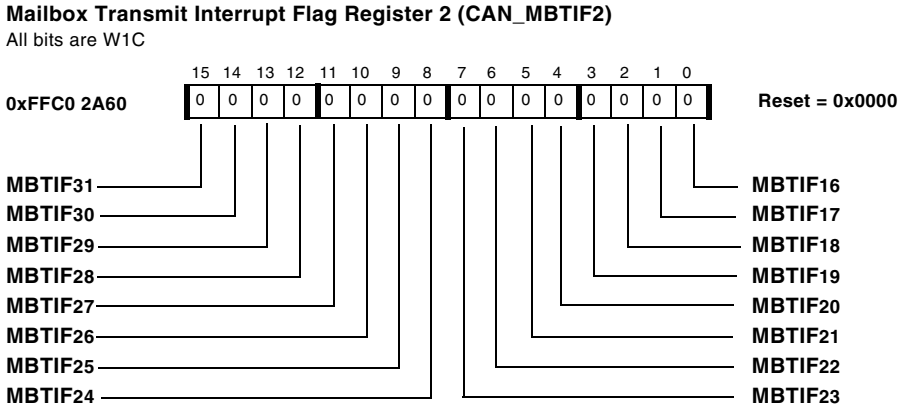


Figure 9-54. Mailbox Transmit Interrupt Flag Register 2

CAN_MBRIFx Registers

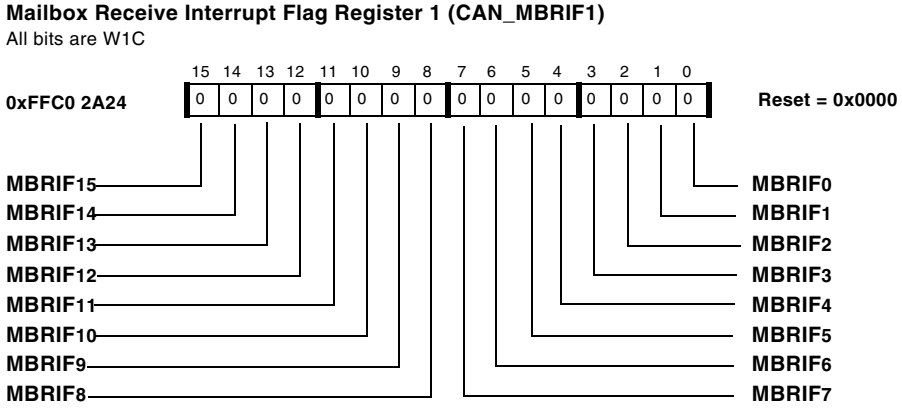


Figure 9-55. Mailbox Receive Interrupt Flag Register 1

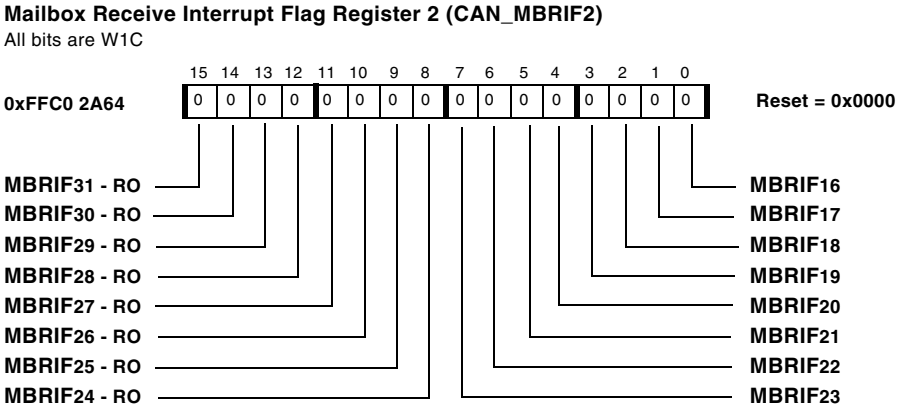


Figure 9-56. Mailbox Receive Interrupt Flag Register 2

Universal Counter Registers

Figure 9-57 through Figure 9-59 show the universal counter registers.

CAN_UCCNF Register

Universal Counter Configuration Mode Register (CAN_UCCNF)

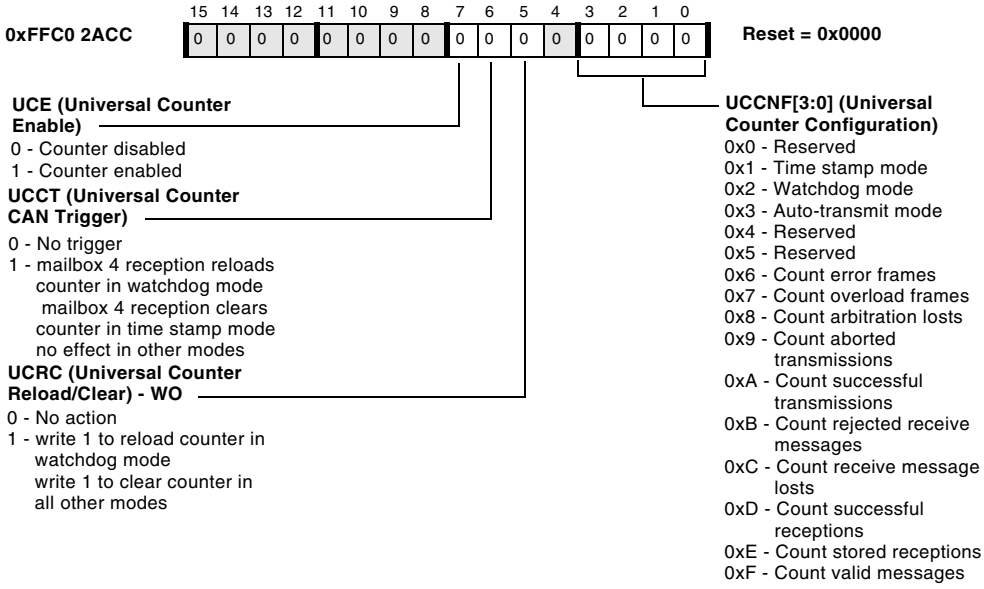


Figure 9-57. Universal Counter Configuration Mode Register

CAN_UCCNT Register

Universal Counter Register (CAN_UCCNT)

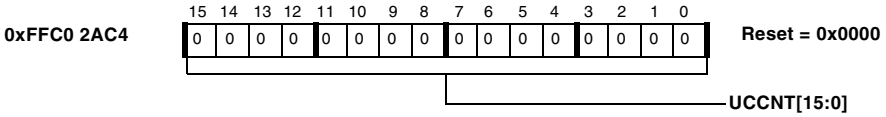


Figure 9-58. Universal Counter Register

CAN_UCRC Register

Universal Counter Reload/Capture Register (CAN_UCRC)

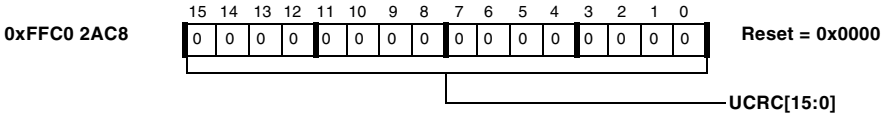


Figure 9-59. Universal Counter Reload/Capture Register

Error Registers

Figure 9-60 through Figure 9-62 show the CAN error registers.

CAN_CEC Register

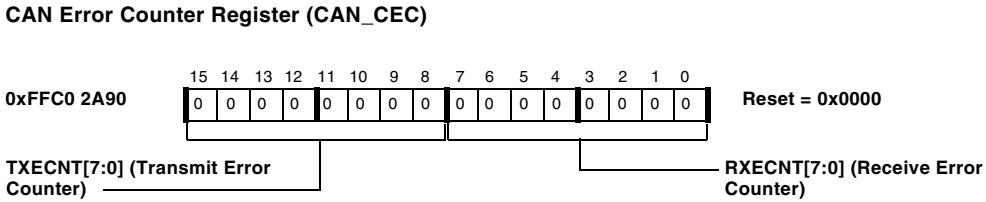


Figure 9-60. CAN Error Counter Register

CAN_ESR Register

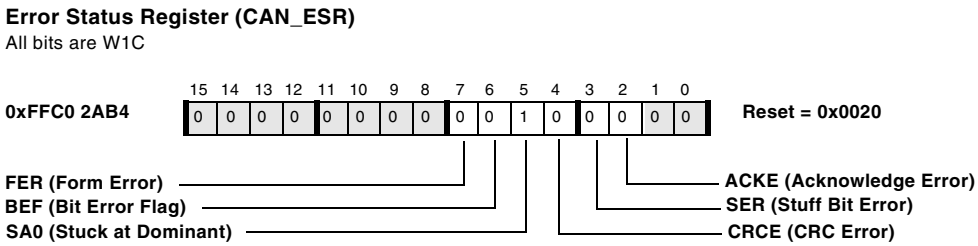


Figure 9-61. Error Status Register

CAN_EWR Register

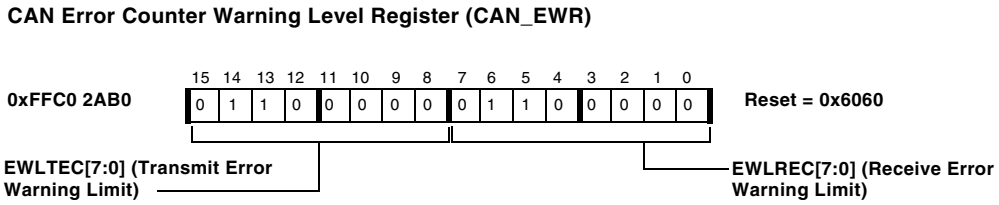


Figure 9-62. CAN Error Counter Warning Level Register

Programming Examples

The following CAN code examples (Listing 9-2 through Listing 9-4 on page 9-91) show how to program the CAN hardware and timing, initialize mailboxes, perform transfers, and service interrupts. Each of these code examples assumes that the appropriate header file is included in the source code (that is, `#include <defBF537.h>` for ADSP-BF537 projects).

CAN Setup Code

The following code initializes the port pins to connect to the CAN controller and configures the CAN timing parameters.

Listing 9-2. Initializing CAN

```
Initialize_CAN:
  P0.H = HI(PORT_MUX);  /* CAN pins muxed on Port J */
  P0.L = LO(PORT_MUX);
  R0 = PJCE_CAN(Z);  /* Enable CAN TX/RX pins */
  W[P0] = R0;
  SSYNC;

  /* ====================================================
  ** Set CAN Bit Timing
  **
  ** CAN_TIMING - SJW, TSEG2, and TSEG1 governed by:
  ** SJW <= TSEG2 <= TSEG1
  **
  ** ====================================================
  */
  P0.H = HI(CAN_TIMING);
  P0.L = LO(CAN_TIMING);
```

```
R0 = 0x0334(Z);  /* SJW = 3, TSEG2 = 3, TSEG1 = 4 */
W[P0] = R0;
SSYNC;

/* ==================================================
** CAN_CLOCK  - Calculate Prescaler (BRP)
**
** Assume a 500kbps CAN rate is desired, which means
** the duration of the bit on the CAN bus (tBIT) is
** 2us. Using the tBIT formula from the HRM, solve for
** TQ:
**
** tBIT = TQ x (1 + (TSEG1 + 1) + (TSEG2 + 1))
** 2us  = TQ x (1 + (4 + 1) + (3 + 1))
** 2e-6 = TQ x (1 + 5 + 4)
** TQ = 2e-6 / 10
** TQ = 2e-7
**
** Once time quantum (TQ) is known, BRP can be derived
** from the TQ formula in the HRM.  Assume the default
** PLL settings are used for the ADSP-BF537 EZ-KIT,
** which implies that System Clock (SCLK) is 50MHz:
**
** TQ = (BRP+1) / SCLK
** 2e-7 = (BRP+1) / 50e6
** (BRP+1) = 10
** BRP = 9
*/
P0.L = LO(CAN_CLOCK);
R0   = 9(Z);
W[P0] = R0;
SSYNC;

RTS;
```

Initializing and Enabling CAN Mailboxes

Before the CAN can transfer data, the mailbox area must be properly set up and the controller must be initialized properly.

Listing 9-3. Initializing and Enabling Mailboxes

```
CAN_Initialize_Mailboxes:
  P0.H = HI(CAN_MD1);  /* Configure Mailbox Direction */
  P0.L = LO(CAN_MD1);
  R0 = W[P0](Z);
  BITCLR(R0, BITPOS(MD8));  /* Set MB08 for Transmit */
  BITSET(R0, BITPOS(MD9));  /* Set MB09 for Receive */
  W[P0] = R0;
  SSYNC;

  /* ==================================================
  ** Populate CAN Mailbox Area
  **
  ** Mailbox 8 transmits ID 0x411 with 4 bytes of data
  ** Bytes 0 and 1 are a data pattern 0xAABB.  Bytes 2
  ** and 3 will be a count value for the number of times
  ** that message is properly sent.
  **
  ** Mailbox 9 will receive message ID 0x007
  **
  ** ==================================================
  */

  /* Initialize Mailbox 8 For Transmit */
  R0 = 0x411 << 2;  /* Put Message ID in correct slot */
  P0.L = LO(CAN_MB_ID1(8));  /* Access MB08 ID1 Register */
  W[P0] = R0;  /* Remote frame disabled, 11 bit ID */

  R0 = 0;
  P0.L = LO(CAN_MB_ID0(8));
  W[P0] = R0;  /* Zero Out Lower ID Register */

  R0 = 4;
  P0.L = LO(CAN_MB_LENGTH(8));
```

```
W[P0] = R0;  /* Set DLC to 4 Bytes */

R0 = 0xAABB(Z);
P0.L = LO(CAN_MB_DATA3(8));
W[P0] = R0;  /* Byte0 = 0xAA, Byte1 = 0xBB */

R0 = 1;
P0.L = LO(CAN_MB_DATA2(8));
W[P0] = R0;  /* Initialize Count to 1 */

/* Initialize Mailbox 9 For Receive */
R0 = 0x007 << 2;  /* Put Message ID in correct slot */
P0.L = LO(CAN_MB_ID1(9));  /* Access MB08 ID1 Register */
W[P0] = R0;  /* Remote frame disabled, 11 bit ID */

R0 = 0;
P0.L = LO(CAN_MB_ID0(9));
W[P0] = R0;  /* Zero Out Lower ID Register */
SSYNC;

/* Enable the Configured Mailboxes */
P0.L = LO(CAN_MC1);
R0 = W[P0](Z);
BITSET(R0, BITPOS(MC8));  /* Enable MB08 */
BITSET(R0, BITPOS(MC9));  /* Enable MB09 */
W[P0] = R0;
SSYNC;
RTS;
```

Initiating CAN Transfers and Processing Interrupts

After the mailboxes are properly set up, transfers can be requested in the CAN controller. This code example initializes the CAN-level interrupts, takes the CAN controller out of configuration mode, requests a transfer,

and then waits for and processes CAN TX and RX interrupts. This example assumes that the CAN_RX_HANDLER and CAN_TX_HANDLER have been properly registered in the system interrupt controller and that the interrupts are enabled properly in the SIC_IMASK register.

Listing 9-4. CAN Transfers and Interrupts

```
CAN_SetupIRQs_and_Transfer:
  P0.H = HI(CAN_MBIM1);
  P0.L = LO(CAN_MBIM1);
  R0 = 0;
  BITSET(R0, BITPOS(MBIM8));  /* Enable Mailbox Interrupts */
  BITSET(R0, BITPOS(MBIM9));  /* for Mailboxes 8 and 9 */
  W[P0] = R0;
  SSYNC;

  /* Leave CAN Configuration Mode (Clear CCR) */
  P0.L = LO(CAN_CONTROL);
  R0 = W[P0](Z);
  BITCLR(R0, BITPOS(CCR));
  W[P0] = R0;

  P0.L = LO(CAN_STATUS);
/* Wait for CAN Configuration Acknowledge (CCA) */
  WAIT_FOR_CCA_TO_CLEAR:
    R1 = W[P0](Z);
    CC = BITTST (R1, BITPOS(CCA));
    IF CC JUMP WAIT_FOR_CCA_TO_CLEAR;
  P0.L = LO(CAN_TRS1);
  R0 = TRS8;  /* Transmit Request MB08 */
  W[P0] = R0;  /* Issue Transmit Request */
  SSYNC;


  Wait_Here_For_IRQs:
    NOP;
    NOP;
    NOP;
```

```
    JUMP Wait_Here_For_IRQs;

/* ==================================================
** CAN_TX_HANDLER
**
** ISR clears the interrupt request from MB8, writes
** new data to be sent, and requests to send again
**
** ==================================================
*/

CAN_TX_HANDLER:
  [--SP] = (R7:6, P5:5);  /* Save Clobbered Registers */
  [--SP] = ASTAT;

  P5.H = HI(CAN_MBTIF1);
  P5.L = LO(CAN_MBTIF1);
  R7 = MBTIF8;
  W[P5] = R7;  /* Clear Interrupt Request Bit for MB08 */

  P5.L = LO(CAN_MB_DATA2(8));
  R7 = W[P5](Z);  /* Retrieve Previously Sent Data */

  R6 = 0xFF;  /* Mask Upper Byte to Check Lower */
  R6 = R6 & R7;  /* Byte for Wrap */
  R5 = 0xFF;  /* Check Wrap Condition */

  CC = R6 == R5;  /* Check if Lower Byte Wraps */

  IF CC JUMP HANDLE_COUNT_WRAP;
  R7 += 1;  /* If no wrap, Increment Count */
  JUMP PREPARE_TO_SEND;

  HANDLE_COUNT_WRAP:
    R6 = 0xFF00(Z);  /* Mask Off Lower Byte */
    R7 = R7 & R6;  /* Sets Lower Byte to 0 */
    R6 = 0x0100(Z);  /* Increment Value for Upper Byte */
    R7 = R7 + R6;   /* Increment Upper Byte */

  PREPARE_TO_SEND:
    W[P5] = R7;  /* Set New TX Data */

    P5.L = LO(CAN_TRS1);
    R7 = TRS8;
```

```
    W[P5] = R7;  /* Issue New Transmit Request */

  ASTAT = [SP++];  /* Restore Clobbered Registers */
  (R7:6, P5:5) = [SP++];
  SSYNC;
  RTI;

/* ====================================================
** CAN_RX_HANDLER
**
** ISR clears the interrupt request from MB9, writes
** new data to be sent, and requests to send again
**
** ====================================================
*/
CAN_RX_HANDLER:
  [--SP] = (R7:7, P5:4);  /* Save Clobbered Registers */
  [--SP] = ASTAT;

  P4.H = CAN_RX_WORD;  /* Set Pointer to Storage Element */
  P4.L = CAN_RX_WORD;

  P5.H = HI(CAN_MBRIF1);
  P5.L = LO(CAN_MBRIF1);
  R7 = MBRIF9;
  W[P5] = R7;  /* Clear Interrupt Request Bit for MB09 */

  P5.L = LO(CAN_MB_DATA3(9));
  R7 = W[P5](Z);  /* Read data from mailbox */
  W[P4] = R7;  /* Store data to SDRAM */

  ASTAT = [SP++];  /* Restore Clobbered Registers */
  (R7:7, P5:4) = [SP++];
  SSYNC;
  RTI;
```

10 SPI COMPATIBLE PORT CONTROLLERS

This chapter describes the Serial Peripheral Interface (SPI) port. Following an overview and a list of key features is a description of operation and functional modes of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

Overview

The processor has an SPI port that provides an I/O interface to a wide variety of SPI compatible peripheral devices.

With a range of configurable options, the SPI port provides a glueless hardware interface with other SPI compatible devices. SPI is a four-wire interface consisting of two data signals, a device select signal, and a clock signal. SPI is a full-duplex synchronous serial interface, supporting master modes, slave modes, and multimaster environments. The SPI compatible peripheral implementation also supports programmable bit rate and clock phase/polarities. The SPI features the use of open drain drivers to support the multimaster scenario and to avoid data contention.

Features

The SPI includes these features:

- Full duplex, synchronous serial interface
- Supports 8- or 16-bit word sizes

- Programmable baud rate, clock phase, and polarity
- Supports multimaster environments
- Integrated DMA controller
- Double-buffered transmitter and receiver
- 7 SPI chip select outputs, 1 SPI device select input
- Programmable shift direction of MSB or LSB first
- Interrupt generation on mode fault, overflow, and underflow
- Shadow register to aid debugging

Typical SPI compatible peripheral devices that can be used to interface to the SPI compatible interface include:

- Other CPUs or microcontrollers
- Codecs
- A/D converters
- D/A converters
- Sample rate converters
- SP/DIF or AES/EBU digital audio transmitters and receivers
- LCD displays
- Shift registers
- FPGAs with SPI emulation

Interface Overview

Figure 10-1 provides a block diagram of the SPI. The interface is essentially a shift register that serially transmits and receives data bits, one bit at a time at the SCK rate, to and from other SPI devices. SPI data is transmitted and received at the same time through the use of a shift register. When an SPI transfer occurs, data is simultaneously transmitted (shifted serially out of the shift register) as new data is received (shifted serially into the other end of the same shift register). The SCK synchronizes the shifting and sampling of the data on the two serial data pins.

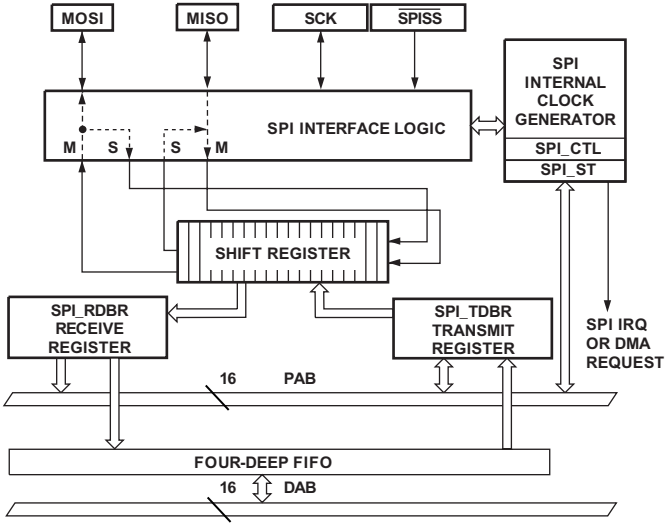


Figure 10-1. SPI Block Diagram

External Interface

Most of the SPI signals are accessible through Port F. The five most important signals (SCK, MISO, MOSI, $\overline{\text{SPISS}}$, and $\overline{\text{SPISSEL1}}$) are not multiplexed with other peripherals. However, by default they function as GPIOs and are individually enabled by the respective bits in the PORTF_FER register.

Port F features three additional slave select signals that are multiplexed with the timer signals. They can be enabled on an individual basis using the PFS4E, PFS5E, and PFS6E bits in the PORT_MUX register. See Figure 14-1 on page 14-5 for details.

Port J also provides three slave select signals. These signals cannot function as normal GPIOs and therefore do not need to be enabled by any function enable bits. However, these outputs are multiplexed with CAN and SPORT0 signals and require PJSE = 1, or PJCE = 10.

Serial Peripheral Interface Clock Signal (SCK)

The SCK signal is the serial clock signal. This control signal is driven by the master and controls the rate at which data is transferred. The master may transmit data at a variety of bit rates. The SCK signal cycles once for each bit transmitted. It is an output signal if the device is configured as a master, and an input signal if the device is configured as a slave.

The SCK is a gated clock that is active during data transfers only for the length of the transferred word. The number of active clock edges is equal to the number of bits driven on the data lines. Slave devices ignore the serial clock if the $\overline{\text{SPISS}}$ input is driven inactive (high).

The SCK is used to shift out and shift in the data driven on the MISO and MOSI lines. The data is always shifted out on active edges of the clock and sampled on inactive edges of the clock. Clock polarity and clock phase relative to data are programmable in the SPI_CTL register and define the transfer format.

The SCK signal can connect to the PF13 pin which functions as a GPIO by default. To enable this pin for use as the SPI clock signal, be sure to first configure the PORTF_FER register to enable the PF13 pin for peripheral use (see "Function Enable Registers" on page 14-23), and then set the PORT_MUX register to specifically enable the SPI clock functionality (see "Port Multiplexer Control Register" on page 14-22).

Master Out Slave In (MOSI)

The MOSI signal is the master out slave in pin, one of the bidirectional I/O data pins. If the processor is configured as a master, the MOSI pin becomes a data transmit (output) pin, transmitting output data. If the processor is configured as a slave, the MOSI pin becomes a data receive (input) pin, receiving input data. In an SPI interconnection, the data is shifted out from the MOSI output pin of the master and shifted into the MOSI input(s) of the slave(s).

The SPI MOSI signal can connect to the PF11 pin which functions as a GPIO by default. To enable this pin for use as the SPI MOSI signal, be sure to first configure the PORTF_FER register to enable the PF11 pin for peripheral use (see "Function Enable Registers" on page 14-23), and then set the PORT_MUX register to specifically enable the SPI master out slave in functionality (see "Port Multiplexer Control Register" on page 14-22).

Master In Slave Out (MISO)

The MISO signal is the master in slave out pin, one of the bidirectional I/O data pins. If the processor is configured as a master, the MISO pin becomes a data receive (input) pin, receiving input data. If the processor is configured as a slave, the MISO pin becomes a data transmit (output) pin, transmitting output data. In an SPI interconnection, the data is shifted out from the MISO output pin of the slave and shifted into the MISO input pin of the master.

The SPI MISO signal can connect to the PF12 pin which functions as a GPIO by default. To enable this pin for use as the SPI MISO signal, be sure to first configure the PORTF_FER register to enable the PF12 pin for peripheral use (see "Function Enable Registers" on page 14-23), and then set the PORT_MUX register to specifically enable the SPI master out slave in functionality (see "Port Multiplexer Control Register" on page 14-22).

Only one slave is allowed to transmit data at any given time.

The SPI configuration example in Figure 10-2 illustrates how the processor can be used as the slave SPI device. The 8-bit host microcontroller is the SPI master.

The processor can be booted via its SPI interface to allow user application code and data to be downloaded before runtime.

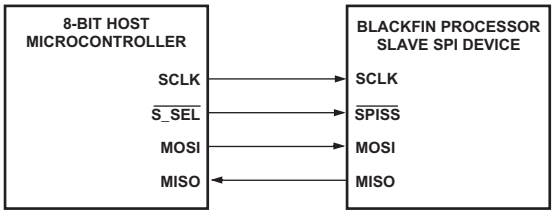


Figure 10-2. ADSP-BF537 as Slave SPI Device

Serial Peripheral Interface Slave Select Input Signal

The SPISS signal is the SPI serial peripheral slave select input signal. This is an active-low signal used to enable a processor when it is configured as a slave device. This input-only pin behaves like a chip select and is provided by the master device for the slave devices. For a master device, it can act as an error signal input in case of the multimaster environment. In multimaster mode, if the SPISS input signal of a master is asserted (driven low), and the PSSE bit in the SPI_CTL register is enabled, an error has occurred. This means that another device is also trying to be the master device.

The $\overline{\text{SPISS}}$ signal can connect to the PF14 pin which functions as a GPIO by default. To enable this pin for use as the SPI slave-select input signal, be sure to first configure the PORTF_FER register to enable the PF14 pin for peripheral use (see "Function Enable Registers" on page 14-23), and then set the PORT_MUX register to specifically enable the SPI slave select input functionality (see "Port Multiplexer Control Register" on page 14-22).

The enable lead time (T1), the enable lag time (T2), and the sequential transfer delay time (T3) each must always be greater than or equal to one-half the SCK period. See Figure 10-3. The minimum time between successive word transfers (T4) is two SCK periods. This is measured from the last active edge of SCK of one word to the first active edge of SCK of the next word. This is independent of the configuration of the SPI (CPHA, MSTR, and so on).

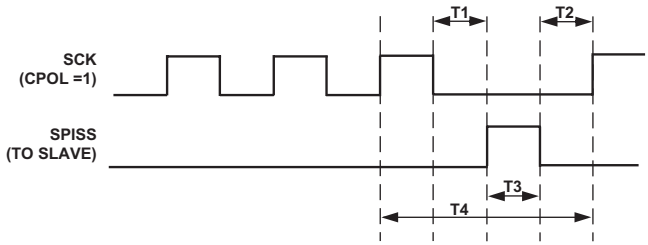


Figure 10-3. SPI Timing

For a master device with CPHA = 0, the slave select output is inactive (high) for at least one-half the SCK period. In this case, T1 and T2 will each always be equal to one-half the SCK period.

Serial Peripheral Interface Slave Select Enable Output Signals

When operating in master mode, Blackfin processors may use any GPIO pin to enable individual SPI slave devices by software. In addition, the SPI module provides hardware support to generate up to seven slave select enable signals automatically. See "SPI Flag Register" on page 10-42 for details.

These signals are always active low in the SPI protocol. Since the respective pins are not driven during reset, it is recommended to pull them up by a resistor.

Table 10-1 summarizes how to setup the port control logic in order to enable the individual slave select enable outputs.

Table 10-1. SPI Slave Select Enable Setup

Signal Name	Pin Name	Port Control To Enable Signal
$\overline{\text{SPISSEL1}}$	PF10	Set bit 10 in PORTF_FER = 1
$\overline{\text{SPISSEL2}}$	PJ11	Set PJSE in PORT_MUX = 1
$\overline{\text{SPISSEL3}}$	PJ10	Set PJSE in PORT_MUX = 1
$\overline{\text{SPISSEL4}}$	PF6	Set bit 6 in PORTF_FER = 1 and Set PFS4E in PORT_MUX = 1
$\overline{\text{SPISSEL5}}$	PF5	Set bit 5 in PORTF_FER = 1 and Set PFS5E in PORT_MUX = 1
$\overline{\text{SPISSEL6}}$	PF4	Set bit 4 in PORTF_FER = 1 and Set PFS6E in PORT_MUX = 1
$\overline{\text{SPISSEL7}}$	PJ5	Set PJCE in PORT_MUX = 10

If enabled as a master, the SPI uses the SPI_FLG register to enable up to seven general-purpose port pins to be used as individual slave select lines. Before manipulating this register, the PFx and PJx port pins that are to be used as SPI slave-select outputs must first be configured as such. To work as SPI output pins, the PFx and PJx pins must be enabled for use by SPI in the PORT_MUX register (see "Port Multiplexer Control Register" on page 14-22). For PFx pins only, the PORTF_FER register (see "Function Enable Registers" on page 14-23) must also be modified to enable those PFx pins for peripheral use. Refer to Table 10-2 for more details regarding which port pins must be configured prior to being modified via the SPI_FLG register.

In slave mode, the SPI_FLG bits have no effect, and each SPI uses the $\overline{\text{SPISS}}$ input as a slave select. Just as in the master mode case, the PF14 pin must first be configured as a peripheral pin in the PORT_MUX register, and then as an SPI pin in the PORTF_FER register. Figure 10-13 on page 10-42 shows the SPI_FLG register diagram.

Table 10-2. SPI_FLG Bit Mapping to Port Pins

Bit	Name	Function	Port Pin	Default
0		Reserved		0
1	FLS1	$\overline{\text{SPISSEL1}}$ Enable	PF10	0
2	FLS2	$\overline{\text{SPISSEL2}}$ Enable	PJ11	0
3	FLS3	$\overline{\text{SPISSEL3}}$ Enable	PJ10	0
4	FLS4	$\overline{\text{SPISSEL4}}$ Enable	PF6	0
5	FLS5	$\overline{\text{SPISSEL5}}$ Enable	PF5	0
6	FLS6	$\overline{\text{SPISSEL6}}$ Enable	PF4	0
7	FLS7	$\overline{\text{SPISSEL7}}$ Enable	PJ5	0
8		Reserved		1
9	FLG1	$\overline{\text{SPISSEL1}}$ Value	PF10	1
10	FLG2	$\overline{\text{SPISSEL2}}$ Value	PJ11	1
11	FLG3	$\overline{\text{SPISSEL3}}$ Value	PJ10	1
12	FLG4	$\overline{\text{SPISSEL4}}$ Value	PF6	1
13	FLG5	$\overline{\text{SPISSEL5}}$ Value	PF5	1
14	FLG6	$\overline{\text{SPISSEL6}}$ Value	PF4	1
15	FLG7	$\overline{\text{SPISSEL7}}$ Value	PJ5	1

Slave Select Inputs

If the SPI is in slave mode, $\overline{\text{SPISS}}$ acts as the slave select input. When enabled as a master, $\overline{\text{SPISS}}$ can serve as an error detection input for the SPI in a multimaster environment. The PSSE bit in SPI_CTL enables this feature. When PSSE = 1, the $\overline{\text{SPISS}}$ input is the master mode error input. Otherwise, $\overline{\text{SPISS}}$ is ignored.

Use of FLS Bits in SPI_FLG for Multiple Slave SPI Systems

The FLSx bits in the SPI_FLG register are used in a multiple slave SPI environment. For example, if there are eight SPI devices in the system including a processor master, the master processor can support the SPI mode transactions across the other seven devices. This configuration requires only one master processor in this multislave environment. For example, assume that the SPI is the master. The seven port pins that can be configured as SPI master mode slave-select output pins can be connected to each of the slave SPI device's $\overline{\text{SPISS}}$ pins. In this configuration, the FLSx bits in SPI_FLG can be used in three cases.

In cases 1 and 2, the processor is the master and the seven microcontrollers/peripherals with SPI interfaces are slaves. The processor can:

1. Transmit to all seven SPI devices at the same time in a broadcast mode. Here, all FLSx bits are set.

2. Receive and transmit from one SPI device by enabling only one slave SPI device at a time.

   In case 3, all eight devices connected via SPI ports can be other processors.

3. If all the slaves are also processors, then the requester can receive data from only one processor (enabled by clearing the EMISO bit in the six other slave processors) at a time and transmit broadcast data to all seven at the same time. This EMISO feature may be available in

some other microcontrollers. Therefore, it is possible to use the EMISO feature with any other SPI device that includes this functionality.

Figure 10-4 shows one processor as a master with three processors (or other SPI compatible devices) as slaves.

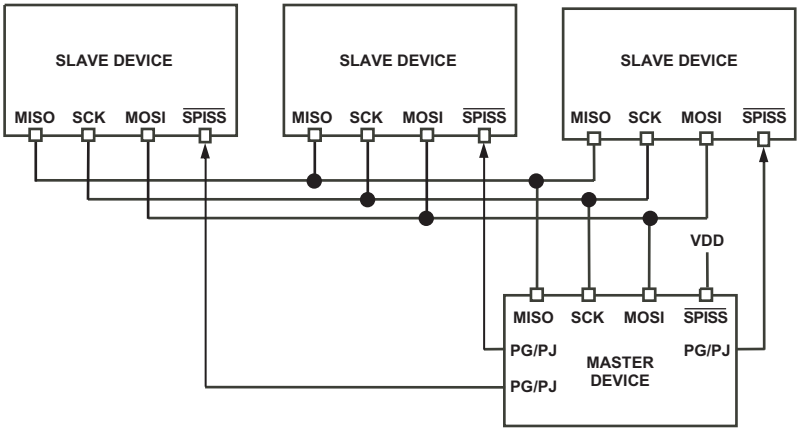


Figure 10-4. Single-Master, Multiple-Slave Configuration

The transmit buffer becomes full after it is written to. It becomes empty when a transfer begins and the transmit value is loaded into the shift register. The receive buffer becomes full at the end of a transfer when the shift register value is loaded into the receive buffer. It becomes empty when the receive buffer is read.

The SPIF bit is set when the SPI port is disabled.

Upon entering DMA mode, the transmit buffer and the receive buffer become empty. That is, the TXS bit and the RXS bit are initially cleared upon entering DMA mode.

When using DMA for SPI transmit, the DMA_DONE interrupt signifies that the DMA FIFO is empty. However, at this point there may still be data in the SPI DMA FIFO waiting to be transmitted. Therefore, software needs to poll TXS in the SPI_STAT register until it goes low for 2 successive reads, at which point the SPI DMA FIFO will be empty. When the SPIF bit subsequently goes low, the last word has been transferred and the SPI can be disabled or enabled for another mode.

Internal Interfaces

The SPI has dedicated connections to the processor's PAB and DAB.

The low-latency PAB bus is used to map the SPI resources into the system MMR space through the PAB bus. For the PAB accesses to SPI MMRs, the primary performance criteria is latency, not throughput. Transfer latencies for both read and write transfers on the PAB are 2 SCLK cycles.

The DAB bus provides a means for DMA SPI transfers to gain access to on-chip and off-chip memory with little or no degradation in core bandwidth to memory. The SPI peripheral, as a DMA master, is capable of sourcing DMA accesses. A single arbiter supports a programmable priority arbitration policy for access to the DAB. Table 2-1 on page 2-9 shows the default arbitration priority.

DMA Functionality

The SPI has a single DMA engine which can be configured to support either an SPI transmit channel or a receive channel, but not both simultaneously. Therefore, when configured as a transmit channel, the received data will essentially be ignored.

When configured as a receive channel, what is transmitted is irrelevant. A 16-bit by four-word FIFO (without burst capability) is included to improve throughput on the DAB.

When using DMA for SPI transmit, the DMA_DONE interrupt signifies that the DMA FIFO is empty. However, at this point there may still be data in the SPI DMA FIFO waiting to be transmitted. Therefore, software needs to poll TXS in the SPI_STAT register until it goes low for two successive reads, at which point the SPI DMA FIFO will be empty. When the SPIF bit subsequently goes low, the last word has been transferred and the SPI can be disabled or enabled for another mode.

The four-word FIFO is cleared when the SPI port is disabled.

SPI Transmit Data Buffer

The SPI_TDBR register is a 16-bit read-write register. Data is loaded into this register before being transmitted. Just prior to the beginning of a data transfer, the data in SPI_TDBR is loaded into the SFDR register. A read of SPI_TDBR can occur at any time and does not interfere with or initiate SPI transfers.

When the DMA is enabled for transmit operation, the DMA engine loads data into this register for transmission just prior to the beginning of a data transfer. A write to SPI_TDBR should not occur in this mode because this data will overwrite the DMA data to be transmitted.

When the DMA is enabled for receive operation, the contents of SPI_TDBR are repeatedly transmitted. A write to SPI_TDBR is permitted in this mode, and this data is transmitted.

If the SZ control bit in the SPI_CTL register is set, SPI_TDBR may be reset to 0 under certain circumstances.

If multiple writes to SPI_TDBR occur while a transfer is already in progress, only the last data written is transmitted. None of the intermediate values written to SPI_TDBR are transmitted. Multiple writes to SPI_TDBR are possible, but not recommended.

SPI Receive Data Buffer

The SPI_RDBR register is a 16-bit read-only register. At the end of a data transfer, the data in the shift register is loaded into SPI_RDBR. During a DMA receive operation, the data in SPI_RDBR is automatically read by the DMA. When SPI_RDBR is read via software, the RXS bit is cleared and an SPI transfer may be initiated (if TIMOD = 00).

The SPI_SHADOW register has been provided for use in debugging software. This register is at a different address than the receive data buffer, SPI_RDBR, but its contents are identical to that of SPI_RDBR. When a software read of SPI_RDBR occurs, the RXS bit in SPI_STAT is cleared and an SPI transfer may be initiated (if TIMOD = 00 in SPI_CTL). No such hardware action occurs when the SPI_SHADOW register is read. The SPI_SHADOW register is read-only.

Description of Operation

The following sections describe the operation of the SPI.

SPI Transfer Protocols

The SPI protocol supports four different combinations of serial clock phase and polarity. These combinations are selected using the CPOL and CPHA bits in SPI_CTL.

The figures "SPI Transfer Protocol for CPHA = 0" on page 10-16 and "SPI Transfer Protocol for CPHA = 1" on page 10-16 demonstrate the two basic transfer formats as defined by the CPHA bit. Two waveforms are

shown for SCK—one for CPOL = 0 and the other for CPOL = 1. The diagrams may be interpreted as master or slave timing diagrams since the SCK, MISO, and MOSI pins are directly connected between the master and the slave. The MISO signal is the output from the slave (slave transmission), and the MOSI signal is the output from the master (master transmission). The SCK signal is generated by the master, and the $\overline{\text{SPISS}}$ signal is the slave device select input to the slave from the master. The diagrams represent an 8-bit transfer (SIZE = 0) with the Most Significant Bit (MSB) first (LSBF = 0). Any combination of the SIZE and LSBF bits of SPI_CTL is allowed. For example, a 16-bit transfer with the Least Significant Bit (LSB) first is another possible configuration.

The clock polarity and the clock phase should be identical for the master device and the slave device involved in the communication link. The transfer format from the master may be changed between transfers to adjust to various requirements of a slave device.

When CPHA = 0, the slave select line, $\overline{\text{SPISS}}$, must be inactive (high) between each serial transfer. This is controlled automatically by the SPI hardware logic. When CPHA = 1, $\overline{\text{SPISS}}$ may either remain active (low) between successive transfers or be inactive (high). This must be controlled by the software via manipulation of SPI_FLG.

Figure 10-5 shows the SPI transfer protocol for CPHA = 0. Note SCK starts toggling in the middle of the data transfer, SIZE = 0, and LSBF = 0.

Figure 10-6 shows the SPI transfer protocol for CPHA = 1. Note SCK starts toggling at the beginning of the data transfer, SIZE = 0, and LSBF = 0.

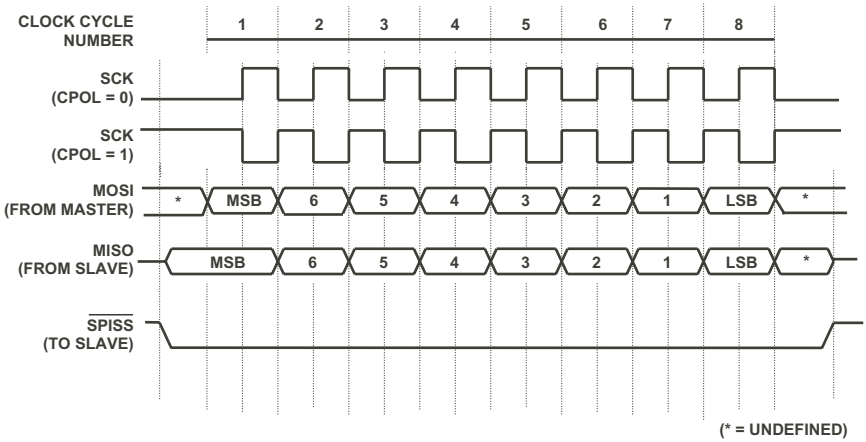


Figure 10-5. SPI Transfer Protocol for CPHA = 0

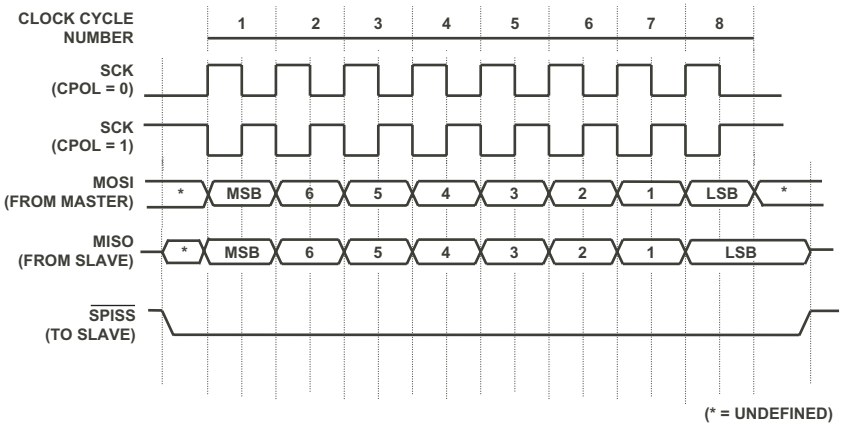


Figure 10-6. SPI Transfer Protocol for CPHA = 1

SPI General Operation

The SPI can be used in a single master as well as multimaster environment. The MOSI, MISO, and the SCK signals are all tied together in both configurations. SPI transmission and reception are always enabled simultaneously, unless the broadcast mode has been selected. In broadcast mode, several slaves can be enabled to receive, but only one of the slaves must be in transmit mode driving the MISO line. If the transmit or receive is not needed, it can simply be ignored. This section describes the clock signals, SPI operation as a master and as a slave, and error generation.

Precautions must be taken to avoid data corruption when changing the SPI module configuration. The configuration must not be changed during a data transfer. The clock polarity should only be changed when no slaves are selected. An exception to this is when an SPI communication link consists of a single master and a single slave, CPHA = 1, and the slave select input of the slave is always tied low. In this case, the slave is always selected and data corruption can be avoided by enabling the slave only after both the master and slave devices are configured.

In a multimaster or multislave SPI system, the data output pins (MOSI and MISO) can be configured to behave as open drain outputs, which prevents contention and possible damage to pin drivers. An external pull-up resistor is required on both the MOSI and MISO pins when this option is selected.

The WOM bit controls this option. When WOM is set and the SPI is configured as a master, the MOSI pin is three-stated when the data driven out on MOSI is a logic high. The MOSI pin is not three-stated when the driven data is a logic low. Similarly, when WOM is set and the SPI is configured as a slave, the MISO pin is three-stated if the data driven out on MISO is a logic high.

During SPI data transfers, one SPI device acts as the SPI link master, where it controls the data flow by generating the SPI serial clock and asserting the SPI device select signal ($\overline{\text{SPISS}}$). The other SPI device acts as

the slave and accepts new data from the master into its shift register, while it transmits requested data out of the shift register through its SPI transmit data pin. Multiple processors can take turns being the master device, as can other microcontrollers or microprocessors. One master device can also simultaneously shift data into multiple slaves (known as broadcast mode). However, only one slave may drive its output to write data back to the master at any given time. This must be enforced in broadcast mode, where several slaves can be selected to receive data from the master, but only one slave at a time can be enabled to send data back to the master.

In a multimaster or multidevice environment where multiple processors are connected via their SPI ports, all `MOSI` pins are connected together, all `MISO` pins are connected together, and all `SCK` pins are connected together.

For a multislave environment, the processor can make use of seven programmable flags that are dedicated SPI slave select signals for the SPI slave devices. See Table 10-2 on page 10-9.

At reset, the SPI is disabled and configured as a slave.

SPI Control

The `SPI_CTL` register is used to configure and enable the SPI system. This register is used to enable the SPI interface, select the device as a master or slave, and determine the data transfer format and word size.

The term “word” refers to a single data transfer of either 8 bits or 16 bits, depending on the word length (`SIZE`) bit in `SPI_CTL`. There are two special bits which can also be modified by the hardware: `SPE` and `MSTR`.

The `TIMOD` field is used to specify the action that initiates transfers to/from the receive/transmit buffers. When set to 00, a SPI port transaction is begun when the receive buffer is read. Data from the first read will need to be discarded since the read is needed to initiate the first SPI port transaction. When set to 01, the transaction is initiated when the transmit buffer is written. A value of 10 selects DMA receive mode and the first

transaction is initiated by enabling the SPI for DMA receive mode. Subsequent individual transactions are initiated by a DMA read of the SPI_RDBR. A value of 11 selects DMA transmit mode and the transaction is initiated by a DMA write of the SPI_TDBR.

The PSSE bit is used to enable the $\overline{\text{SPISS}}$ input for master. When not used, $\overline{\text{SPISS}}$ can be disabled, freeing up a chip pin as general-purpose I/O.

The EMISO bit enables the MISO pin as an output. This is needed in an environment where the master wishes to transmit to various slaves at one time (broadcast). Only one slave is allowed to transmit data back to the master. Except for the slave from whom the master wishes to receive, all other slaves should have this bit cleared.

The SPE and MSTR bits can be modified by hardware when the MODF bit of the SPI_STAT register is set. See "Mode Fault Error (MODF)" on page 10-21.

Figure 10-12 on page 10-41 provides the bit descriptions for SPI_CTL.

Clock Signals

The SCK signal is a gated clock that is only active during data transfers for the duration of the transferred word. The number of active edges is equal to the number of bits driven on the data lines. The clock rate can be as high as one-fourth of the SCLK rate. For master devices, the clock rate is determined by the 16-bit value of SPI_BAUD. For slave devices, the value in SPI_BAUD is ignored. When the SPI device is a master, SCK is an output signal. When the SPI is a slave, SCK is an input signal. Slave devices ignore the serial clock if the slave select input is driven inactive (high).

The SCK signal is used to shift out and shift in the data driven onto the MISO and MOSI lines. The data is always shifted out on one edge of the clock (the active edge) and sampled on the opposite edge of the clock (the sampling edge). Clock polarity and clock phase relative to data are programmable into SPI_CTL and define the transfer format.

SPI Baud Rate

The SPI_BAUD register is used to set the bit transfer rate for a master device. When configured as a slave, the value written to this register is ignored. The serial clock frequency is determined by this formula:

```
SCK Frequency = (Peripheral clock frequency SCLK)/(2 x SPI_BAUD)
```

Writing a value of 0 or 1 to the register disables the serial clock. Therefore, the maximum serial clock rate is one-fourth the system clock rate.

Table 10-3 lists several possible baud rate values for SPI_BAUD.

Table 10-3. SPI Master Baud Rate Example

SPI_BAUD Decimal Value	SPI Clock (SCK) Divide Factor	Baud Rate for SCLK at 100 MHz
0	N/A	N/A
1	N/A	N/A
2	4	25 MHz
3	6	16.7 MHz
4	8	12.5 MHz
65,535 (0xFFFF)	131,070	763 Hz

Error Signals and Flags

The SPI_STAT register is used to detect when an SPI transfer is complete or if transmission/reception errors occur. The SPI_STAT register can be read at any time.

Some of the bits in SPI_STAT are read-only and other bits are sticky. Bits that provide information only about the SPI are read-only. These bits are set and cleared by the hardware. Sticky bits are set when an error condition occurs. These bits are set by hardware and must be cleared by

software. To clear a sticky bit, the user must write a 1 to the desired bit position of SPI_STAT. For example, if the TXE bit is set, the user must write a 1 to bit 2 of SPI_STAT to clear the TXE error condition. This allows the user to read SPI_STAT without changing its value.

Sticky bits are cleared on a reset, but are not cleared on an SPI disable.

See Figure 10-14 on page 10-43 for more information.

Mode Fault Error (MODF)

The MODF bit is set in SPI_STAT when the $\overline{\text{SPISS}}$ input pin of a device enabled as a master is driven low by some other device in the system. This occurs in multimaster systems when another device is also trying to be the master. To enable this feature, the PSSE bit in SPI_CTL must be set. This contention between two drivers can potentially damage the driving pins. As soon as this error is detected, these actions occur:

- The MSTR control bit in SPI_CTL is cleared, configuring the SPI interface as a slave
- The SPE control bit in SPI_CTL is cleared, disabling the SPI system
- The MODF status bit in SPI_STAT is set
- An SPI error interrupt is generated

These four conditions persist until the MODF bit is cleared by software. Until the MODF bit is cleared, the SPI cannot be re-enabled, even as a slave. Hardware prevents the user from setting either SPE or MSTR while MODF is set.

When MODF is cleared, the interrupt is deactivated. Before attempting to re-enable the SPI as a master, the state of the $\overline{\text{SPISS}}$ input pin should be checked to make sure the pin is high. Otherwise, once SPE and MSTR are set, another mode fault error condition immediately occurs.

When SPE and MSTR are cleared, the SPI data and clock pin drivers (MOSI, MISO, and SCK) are disabled. However, the slave select output pins revert to being controlled by the general-purpose I/O port registers. This could lead to contention on the slave select lines if these lines are still driven by the processor. To ensure that the slave select output drivers are disabled once an MODF error occurs, the program must configure the general-purpose I/O port registers appropriately.

When enabling the MODF feature, the program must configure as inputs all of the port pins that will be used as slave selects. Programs can do this by configuring the direction of the port pins prior to configuring the SPI. This ensures that, once the MODF error occurs and the slave selects are automatically reconfigured as port pins, the slave select output drivers are disabled.

Transmission Error (TXE)

The TXE bit is set in SPI_STAT when all the conditions of transmission are met, and there is no new data in SPI_TDBR (SPI_TDBR is empty). In this case, the contents of the transmission depend on the state of the SZ bit in SPI_CTL. The TXE bit is sticky (W1C).

Reception Error (RBSY)

The RBSY flag is set in the SPI_STAT register when a new transfer is completed, but before the previous data can be read from SPI_RDBR. The state of the GM bit in the SPI_CTL register determines whether SPI_RDBR is updated with the newly received data. The RBSY bit is sticky (W1C).

Transmit Collision Error (TXCOL)

The TXCOL flag is set in SPI_STAT when a write to SPI_TDBR coincides with the load of the shift register. The write to SPI_TDBR can be via software or the DMA. The TXCOL bit indicates that corrupt data may have been loaded

into the shift register and transmitted. In this case, the data in SPI_TDBR may not match what was transmitted. This error can easily be avoided by proper software control. The TXCOL bit is sticky (W1C).

Interrupt Output

The SPI has two interrupt output signals: a data interrupt and an error interrupt.

The behavior of the SPI data interrupt signal depends on the TIMOD field in the SPI_CTL register. In DMA mode (TIMOD = 1X), the data interrupt acts as a DMA request and is generated when the DMA FIFO is ready to be written to (TIMOD = 11) or read from (TIMOD = 10). In non-DMA mode (TIMOD = 0X), a data interrupt is generated when the SPI_TDBR is ready to be written to (TIMOD = 01) or when the SPI_RDBR is ready to be read from (TIMOD = 00).

An SPI error interrupt is generated in a master when a mode fault error occurs, in both DMA and non-DMA modes. An error interrupt can also be generated in DMA mode when there is an underflow (TXE when TIMOD = 11) or an overflow (RBSY when TIMOD = 10) error condition. In non-DMA mode, the underflow and overflow conditions set the TXE and RBSY bits in the SPI_STAT register, respectively, but do not generate an error interrupt.

For more information about this interrupt output, see the discussion of the TIMOD bits in "SPI Control" on page 10-18.

Functional Description

The following sections describe the functional operation of the SPI.

Master Mode Operation

When the SPI is configured as a master (and DMA mode is not selected), the interface operates in the following manner.

1. The core writes to the PORTF_FER and/or PORT_MUX registers to properly configure the required PFx and/or PJx pins for SPI use as slave-select outputs and, if necessary, multimaster detection input ($\overline{\text{SPISS}}$).

2. The core writes to SPI_FLG, setting one or more of the SPI Flag Select bits (FLSx). This ensures that the desired slaves are properly deselected while the master is configured.

3. The core writes to the SPI_BAUD and SPI_CTL registers, enabling the device as a master and configuring the SPI system by specifying the appropriate word length, transfer format, baud rate, and other necessary information.

4. If CPHA = 1, the core activates the desired slaves by clearing one or more of the SPI flag bits (FLGx) of SPI_FLG.

5. The TIMOD bits in SPI_CTL determine the SPI transfer initiate mode. The transfer on the SPI link begins upon either a data write by the core to the transmit data buffer (SPI_TDBR) or a data read of the receive data buffer (SPI_RDBR).

6. The SPI then generates the programmed clock pulses on SCK and simultaneously shifts data out of MOSI and shifts data in from MISO. Before a shift, the shift register is loaded with the contents of the SPI_TDBR register. At the end of the transfer, the contents of the shift register are loaded into SPI_RDBR.

7. With each new transfer initiate command, the SPI continues to send and receive words, according to the SPI transfer initiate mode.

See Figure 10-7 on page 10-36 for additional information.

If the transmit buffer remains empty or the receive buffer remains full, the device operates according to the states of the SZ and GM bits in SPI_CTL. If SZ = 1 and the transmit buffer is empty, the device repeatedly transmits 0s on the MOSI pin. One word is transmitted for each new transfer initiate command. If SZ = 0 and the transmit buffer is empty, the device repeatedly transmits the last word it transmitted before the transmit buffer became empty. If GM = 1 and the receive buffer is full, the device continues to receive new data from the MISO pin, overwriting the older data in the SPI_RDBR buffer. If GM = 0 and the receive buffer is full, the incoming data is discarded, and SPI_RDBR is not updated.

Transfer Initiation From Master (Transfer Modes)

When a device is enabled as a master, the initiation of a transfer is defined by the two TIMOD bits of SPI_CTL. Based on those two bits and the status of the interface, a new transfer is started upon either a read of SPI_RDBR or a write to SPI_TDBR. This is summarized in Table 10-4.

If the SPI port is enabled with TIMOD = 01 or TIMOD = 11, the hardware immediately issues a first interrupt or DMA request.

Table 10-4. Transfer Initiation

TIMOD	Function	Transfer Initiated Upon	Action, Interrupt
00	Transmit and Receive	Initiate new single word transfer upon read of SPI_RDBR and previous transfer completed.	Interrupt active when receive buffer is full. Read of SPI_RDBR clears interrupt.
01	Transmit and Receive	Initiate new single word transfer upon write to SPI_TDBR and previous transfer completed.	Interrupt active when transmit buffer is empty. Writing to SPI_TDBR clears interrupt.

Table 10-4. Transfer Initiation (Cont'd)

TIMOD	Function	Transfer Initiated Upon	Action, Interrupt
10	Receive with DMA	Initiate new multiword transfer upon enabling SPI for DMA mode. Individual word transfers begin with a DMA read of SPI_RDBR, and last transfer completed.	Request DMA reads as long as SPI DMA FIFO is not empty.
11	Transmit with DMA	Initiate new multiword transfer upon enabling SPI for DMA mode. Individual word transfers begin with a DMA write to SPI_TDBR, and last transfer completed.	Request DMA writes as long as SPI DMA FIFO is not full.

Slave Mode Operation

When a device is enabled as a slave (and DMA mode is not selected), the start of a transfer is triggered by a transition of the $\overline{\text{SPISS}}$ select signal to the active state (low), or by the first active edge of the clock (SCK), depending on the state of CPHA.

These steps illustrate SPI operation in the slave mode:

1. The core writes to the PORTF_FER and PORT_MUX registers to properly configure the PF14 pin as the $\overline{\text{SPISS}}$ input signal.
2. The core writes to SPI_CTL to define the mode of the serial link to be the same as the mode setup in the SPI master.
3. To prepare for the data transfer, the core writes data to be transmitted into SPI_TDBR.
4. Once the $\overline{\text{SPISS}}$ falling edge is detected, the slave starts sending data on active SCK edges and sampling data on inactive SCK edges.

5. Reception/transmission continues until $\overline{\text{SPISS}}$ is released or until the slave has received the proper number of clock cycles.
6. The slave device continues to receive/transmit with each new falling edge transition on $\overline{\text{SPISS}}$ and/or active SCK clock edge.

See Figure 10-7 on page 10-36 for additional information.

If the transmit buffer remains empty or the receive buffer remains full, the device operates according to the states of the SZ and GM bits in SPI_CTL. If SZ = 1 and the transmit buffer is empty, the device repeatedly transmits 0s on the MISO pin. If SZ = 0 and the transmit buffer is empty, it repeatedly transmits the last word it transmitted before the transmit buffer became empty. If GM = 1 and the receive buffer is full, the device continues to receive new data from the MOSI pin, overwriting the older data in SPI_RDBR. If GM = 0 and the receive buffer is full, the incoming data is discarded, and SPI_RDBR is not updated.

Slave Ready for a Transfer

When a device is enabled as a slave, the actions shown in Table 10-5 are necessary to prepare the device for a new transfer.

Table 10-5. Transfer Preparation

TIMOD	Function	Action, Interrupt
00	Transmit and Receive	Interrupt active when receive buffer is full. Read of SPI_RDBR clears interrupt.
01	Transmit and Receive	Interrupt active when transmit buffer is empty. Writing to SPI_TDBR clears interrupt.
10	Receive with DMA	Request DMA reads as long as SPI DMA FIFO is not empty.
11	Transmit with DMA	Request DMA writes as long as SPI DMA FIFO is not full.

Programming Model

The following sections describe the SPI programming model.

Beginning and Ending an SPI Transfer

The start and finish of an SPI transfer depend on whether the device is configured as a master or a slave, whether the CPHA mode is selected, and whether the transfer initiation mode (TIMOD) is selected. For a master SPI with CPHA = 0, a transfer starts when either SPI_TDBR is written to or SPI_RDBR is read, depending on TIMOD. At the start of the transfer, the enabled slave select outputs are driven active (low). However, the SCK signal remains inactive for the first half of the first cycle of SCK. For a slave with CPHA = 0, the transfer starts as soon as the $\overline{\text{SPISS}}$ input goes low.

For CPHA = 1, a transfer starts with the first active edge of SCK for both slave and master devices. For a master device, a transfer is considered finished after it sends the last data and simultaneously receives the last data bit. A transfer for a slave device ends after the last sampling edge of SCK.

The RXS bit defines when the receive buffer can be read. The TXS bit defines when the transmit buffer can be filled. The end of a single word transfer occurs when the RXS bit is set, indicating that a new word has just been received and latched into the receive buffer, SPI_RDBR. For a master SPI, RXS is set shortly after the last sampling edge of SCK. For a slave SPI, RXS is set shortly after the last SCK edge, regardless of CPHA or CPOL. The latency is typically a few SCLK cycles and is independent of TIMOD and the baud rate. If configured to generate an interrupt when SPI_RDBR is full (TIMOD = 00), the interrupt goes active one SCLK cycle after RXS is set. When not relying on this interrupt, the end of a transfer can be detected by polling the RXS bit.

To maintain software compatibility with other SPI devices, the SPIF bit is also available for polling. This bit may have a slightly different behavior from that of other commercially available devices. For a slave device, SPIF

is cleared shortly after the start of a transfer ($\overline{\text{SPISS}}$ going low for CPHA = 0, first active edge of SCK on CPHA = 1), and is set at the same time as RXS. For a master device, SPIF is cleared shortly after the start of a transfer (either by writing the SPI_TDBR or reading the SPI_RDBR, depending on TIMOD), and is set one-half SCK period after the last SCK edge, regardless of CPHA or CPOL.

The time at which SPIF is set depends on the baud rate. In general, SPIF is set after RXS, but at the lowest baud rate settings (SPI_BAUD < 4). The SPIF bit is set before RXS is set, and consequently before new data is latched into SPI_RDBR, because of the latency. Therefore, for SPI_BAUD = 2 or SPI_BAUD = 3, RXS must be set before SPIF to read SPI_RDBR. For larger SPI_BAUD settings, RXS is guaranteed to be set before SPIF is set.

If the SPI port is used to transmit and receive at the same time, or to switch between receive and transmit operation frequently, then the TIMOD = 00 mode may be the best operation option. In this mode, software performs a dummy read from the SPI_RDBR register to initiate the first transfer. If the first transfer is used for data transmission, software should write the value to be transmitted into the SPI_TDBR register before performing the dummy read. If the transmitted value is arbitrary, it is good practice to set the SZ bit to ensure zero data is transmitted rather than random values. When receiving the last word of an SPI stream, software should ensure that the read from the SPI_RDBR register does not initiate another transfer. It is recommended to disable the SPI port before the final SPI_RDBR read access. Reading the SPI_SHADOW register is not sufficient as it does not clear the interrupt request.

In master mode with the CPHA bit set, software should manually assert the required slave select signal before starting the transaction. After all data has been transferred, software typically releases the slave select again. If the SPI slave device requires the slave select line to be asserted for the complete transfer, this can be done in the SPI interrupt service routine

only when operating in TIMOD = 00 or TIMOD = 10 mode. With TIMOD = 01 or TIMOD = 11, the interrupt is requested while the transfer is still in progress.

Master Mode DMA Operation

When enabled as a master with the DMA engine configured to transmit or receive data, the SPI interface operates as follows.

1. The core writes to the PORTF_FER and/or PORT_MUX registers to properly configure the required PFx and/or PJx pins for SPI use as slave-select outputs and, if necessary, multimaster detection input ($\overline{\text{SPISS}}$).

2. The processor core writes to the appropriate DMA registers to enable the SPI DMA channel and to configure the necessary work units, access direction, word count, and so on. For more information, see Chapter 5, "Direct Memory Access".

3. The processor core writes to the SPI_FLG register, setting one or more of the SPI flag select bits (FLSx).

4. The processor core writes to the SPI_BAUD and SPI_CTL registers, enabling the device as a master and configuring the SPI system by specifying the appropriate word length, transfer format, baud rate, and so on. The TIMOD field should be configured to select either "receive with DMA" (TIMOD = 10) or "transmit with DMA" (TIMOD = 11) mode.

5. If configured for receive, a receive transfer is initiated upon enabling of the SPI. Subsequent transfers are initiated as the SPI reads data from the SPI_RDBR register and writes to the SPI DMA FIFO. The SPI then requests a DMA write to memory. Upon a DMA grant, the DMA engine reads a word from the SPI DMA FIFO and writes to memory.

If configured for transmit, the SPI requests a DMA read from memory. Upon a DMA grant, the DMA engine reads a word from memory and writes to the SPI DMA FIFO. As the SPI writes data from the SPI DMA FIFO into the SPI_TDBR register, it initiates a transfer on the SPI link.

6. The SPI then generates the programmed clock pulses on SCK and simultaneously shifts data out of MOSI and shifts data in from MISO. For receive transfers, the value in the shift register is loaded into the SPI_RDBR register at the end of the transfer. For transmit transfers, the value in the SPI_TDBR register is loaded into the shift register at the start of the transfer.

7. In receive mode, as long as there is data in the SPI DMA FIFO (the FIFO is not empty), the SPI continues to request a DMA write to memory. The DMA engine continues to read a word from the SPI DMA FIFO and writes to memory until the SPI DMA word count register transitions from 1 to 0. The SPI continues receiving words until SPI DMA mode is disabled.

   In transmit mode, as long as there is room in the SPI DMA FIFO (the FIFO is not full), the SPI continues to request a DMA read from memory. The DMA engine continues to read a word from memory and write to the SPI DMA FIFO until the SPI DMA word count register transitions from 1 to 0. The SPI continues transmitting words until the SPI DMA FIFO is empty.

See Figure 10-8 on page 10-37 for additional information.

For receive DMA operations, if the DMA engine is unable to keep up with the receive datastream, the receive buffer operates according to the state of the GM bit. If GM = 1 and the DMA FIFO is full, the device continues to receive new data from the MISO pin, overwriting the older data in the SPI_RDBR register. If GM = 0, and the DMA FIFO is full, the incoming data is discarded, and the SPI_RDBR register is not updated. While performing receive DMA, the transmit buffer is assumed to be empty (and

TXE is set). If SZ = 1, the device repeatedly transmits 0s on the MOSI pin. If SZ = 0, it repeatedly transmits the contents of the SPI_TDBR register. The TXE underrun condition cannot generate an error interrupt in this mode.

For transmit DMA operations, the master SPI initiates a word transfer only when there is data in the DMA FIFO. If the DMA FIFO is empty, the SPI waits for the DMA engine to write to the DMA FIFO before starting the transfer. All aspects of SPI receive operation should be ignored when configured in transmit DMA mode, including the data in the SPI_RDBR register, and the status of the RXS and RBSY bits. The RBSY overrun conditions cannot generate an error interrupt in this mode. The TXE underrun condition cannot happen in this mode (master DMA TX mode), because the master SPI will not initiate a transfer if there is no data in the DMA FIFO.

Writes to the SPI_TDBR register during an active SPI transmit DMA operation should not occur because the DMA data will be overwritten. Writes to the SPI_TDBR register during an active SPI receive DMA operation are allowed. Reads from the SPI_RDBR register are allowed at any time.

DMA requests are generated when the DMA FIFO is not empty (when TIMOD = 10), or when the DMA FIFO is not full (when TIMOD = 11).

Error interrupts are generated when there is an RBSY overflow error condition (when TIMOD = 10).

A master SPI DMA sequence may involve back-to-back transmission and/or reception of multiple DMA work units. The SPI controller supports such a sequence with minimal core interaction.

Slave Mode DMA Operation

When enabled as a slave with the DMA engine configured to transmit or receive data, the start of a transfer is triggered by a transition of the $\overline{\text{SPISS}}$ signal to the active-low state or by the first active edge of SCK, depending on the state of CPHA.

The following steps illustrate the SPI receive or transmit DMA sequence in an SPI slave (in response to a master command).

1. The core writes to the PORTF_FER register to properly configure the PF14 pin as the $\overline{\text{SPISS}}$ input signal.

2. The processor core writes to the appropriate DMA registers to enable the SPI DMA channel and configure the necessary work units, access direction, word count, and so on. For more information, see Chapter 5, "Direct Memory Access".

3. The processor core writes to the SPI_CTL register to define the mode of the serial link to be the same as the mode setup in the SPI master. The TIMOD field will be configured to select either "receive with DMA" (TIMOD = 10) or "transmit with DMA" (TIMOD = 11) mode.

4. If configured for receive, once the slave select input is active, the slave starts receiving and transmitting data on active SCK edges. The value in the shift register is loaded into the SPI_RDBR register at the end of the transfer. As the SPI reads data from the SPI_RDBR register and writes to the SPI DMA FIFO, it requests a DMA write to memory. Upon a DMA grant, the DMA engine reads a word from the SPI DMA FIFO and writes to memory.

   If configured for transmit, the SPI requests a DMA read from memory. Upon a DMA grant, the DMA engine reads a word from memory and writes to the SPI DMA FIFO. The SPI then reads data from the SPI DMA FIFO and writes to the SPI_TDBR register, awaiting the start of the next transfer. Once the slave select input is active, the slave starts receiving and transmitting data on active SCK edges. The value in the SPI_TDBR register is loaded into the shift register at the start of the transfer.

5. In receive mode, as long as there is data in the SPI DMA FIFO (FIFO not empty), the SPI slave continues to request a DMA write to memory. The DMA engine continues to read a word from the SPI DMA FIFO and writes to memory until the SPI DMA word count register transitions from 1 to 0. The SPI slave continues receiving words on active SCK edges as long as the slave select input is active.

   In transmit mode, as long as there is room in the SPI DMA FIFO (FIFO not full), the SPI slave continues to request a DMA read from memory. The DMA engine continues to read a word from memory and write to the SPI DMA FIFO until the SPI DMA word count register transitions from 1 to 0. The SPI slave continues transmitting words on active SCK edges as long as the slave select input is active.

See Figure 10-8 on page 10-37 for additional information.

For receive DMA operations, if the DMA engine is unable to keep up with the receive datastream, the receive buffer operates according to the state of the GM bit. If GM = 1 and the DMA FIFO is full, the device continues to receive new data from the MOSI pin, overwriting the older data in the SPI_RDBR register. If GM = 0 and the DMA FIFO is full, the incoming data is discarded, and the SPI_RDBR register is not updated. While performing receive DMA, the transmit buffer is assumed to be empty and TXE is set. If SZ = 1, the device repeatedly transmits 0s on the MISO pin. If SZ = 0, it repeatedly transmits the contents of the SPI_TDBR register. The TXE underrun condition cannot generate an error interrupt in this mode.

For transmit DMA operations, if the DMA engine is unable to keep up with the transmit stream, the transmit port operates according to the state of the SZ bit. If SZ = 1 and the DMA FIFO is empty, the device repeatedly transmits 0s on the MISO pin. If SZ = 0 and the DMA FIFO is empty, it repeatedly transmits the last word it transmitted before the DMA buffer became empty. All aspects of SPI receive operation should be ignored

when configured in transmit DMA mode, including the data in the SPI_RDBR register, and the status of the RXS and RBSY bits. The RBSY overrun conditions cannot generate an error interrupt in this mode.

Writes to the SPI_TDBR register during an active SPI transmit DMA operation should not occur because the DMA data will be overwritten. Writes to the SPI_TDBR register during an active SPI receive DMA operation are allowed. Reads from the SPI_RDBR register are allowed at any time.

DMA requests are generated when the DMA FIFO is not empty (when TIMOD = 10), or when the DMA FIFO is not full (when TIMOD = 11).

Error interrupts are generated when there is an RBSY overflow error condition (when TIMOD = 10), or when there is a TXE underflow error condition (when TIMOD = 11).

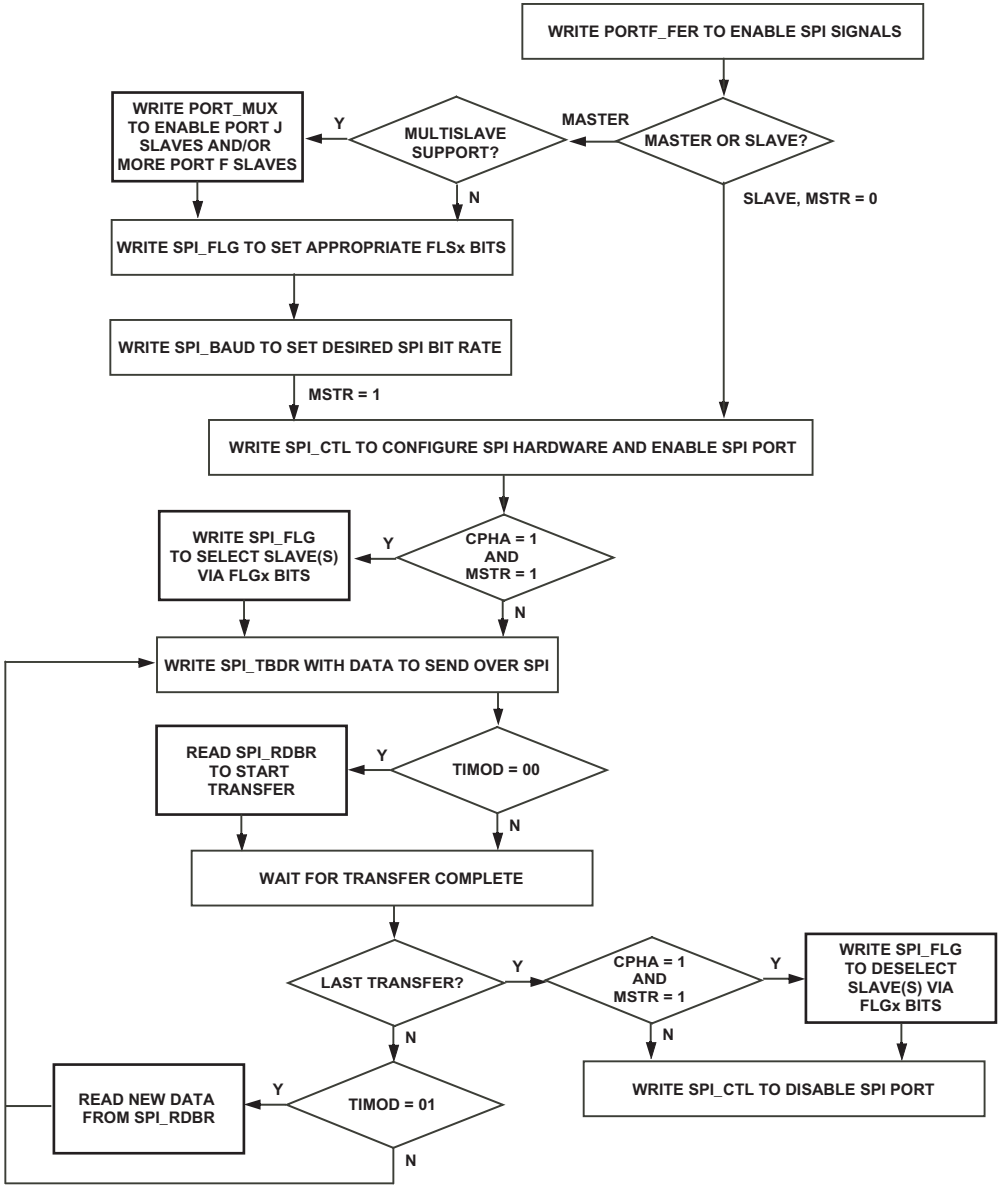


Figure 10-7. Core-Driven SPI Flow Chart

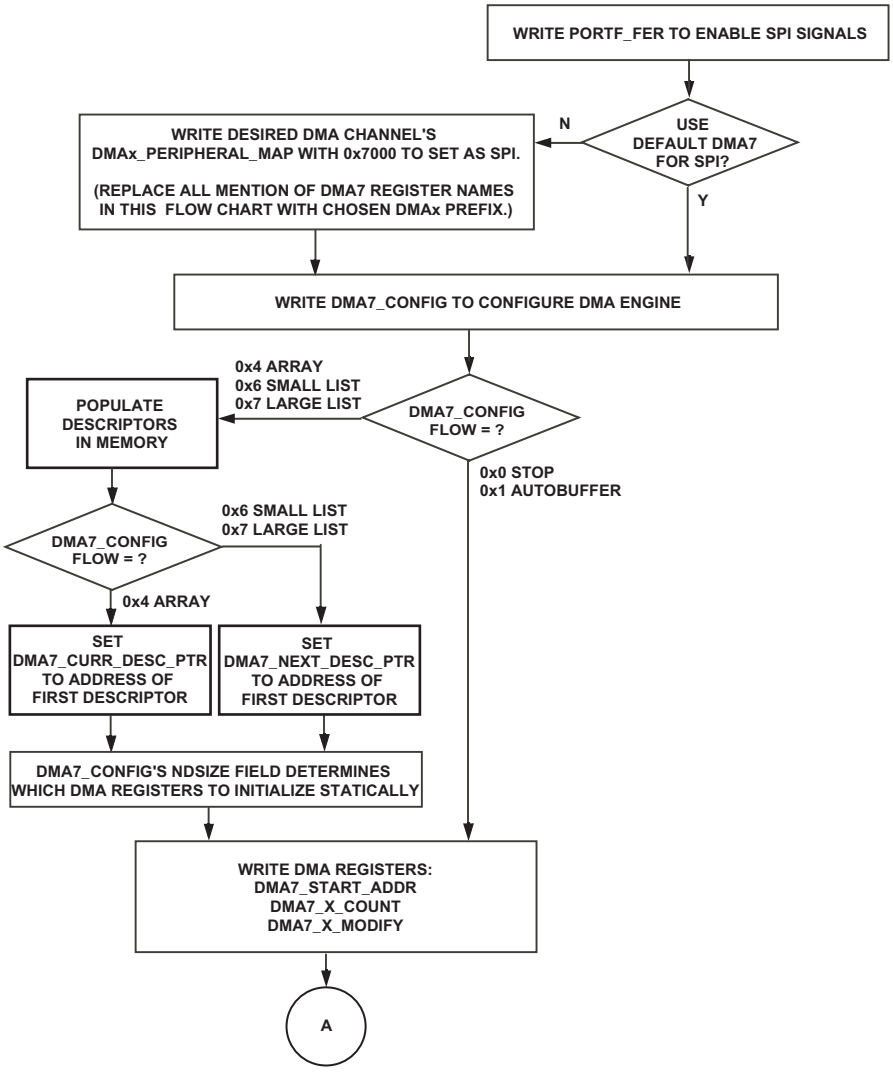


Figure 10-8. SPI DMA Flow Chart (Part 1 of 3)

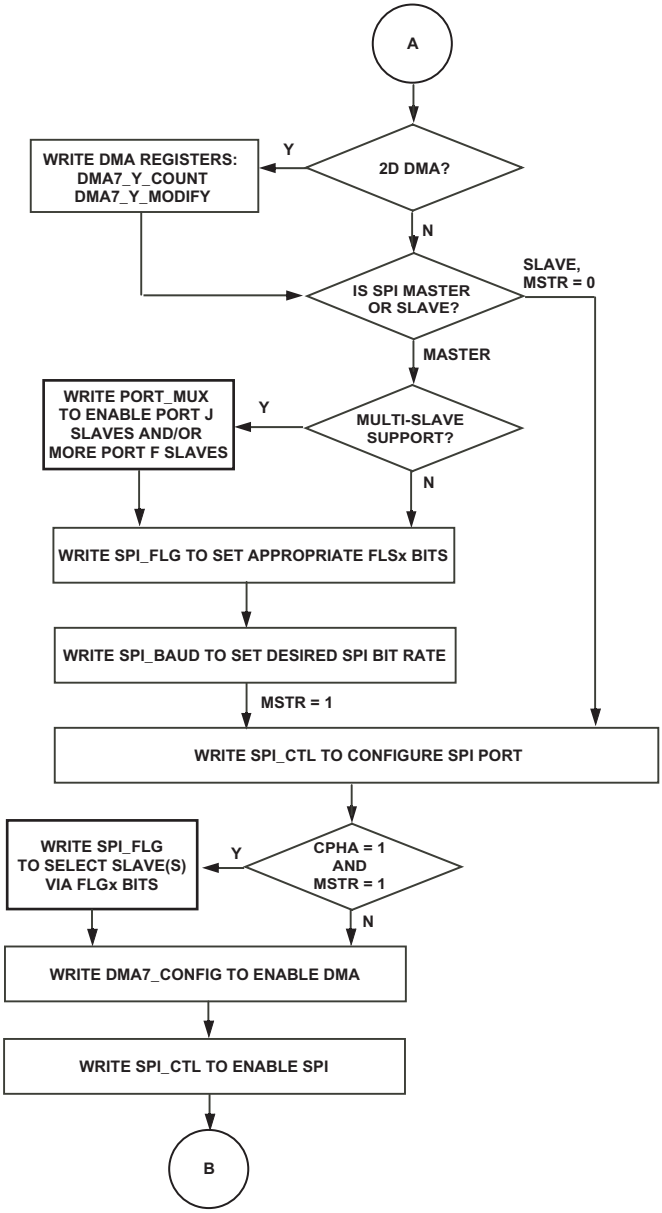


Figure 10-9. SPI DMA Flow Chart (Part 2 of 3)

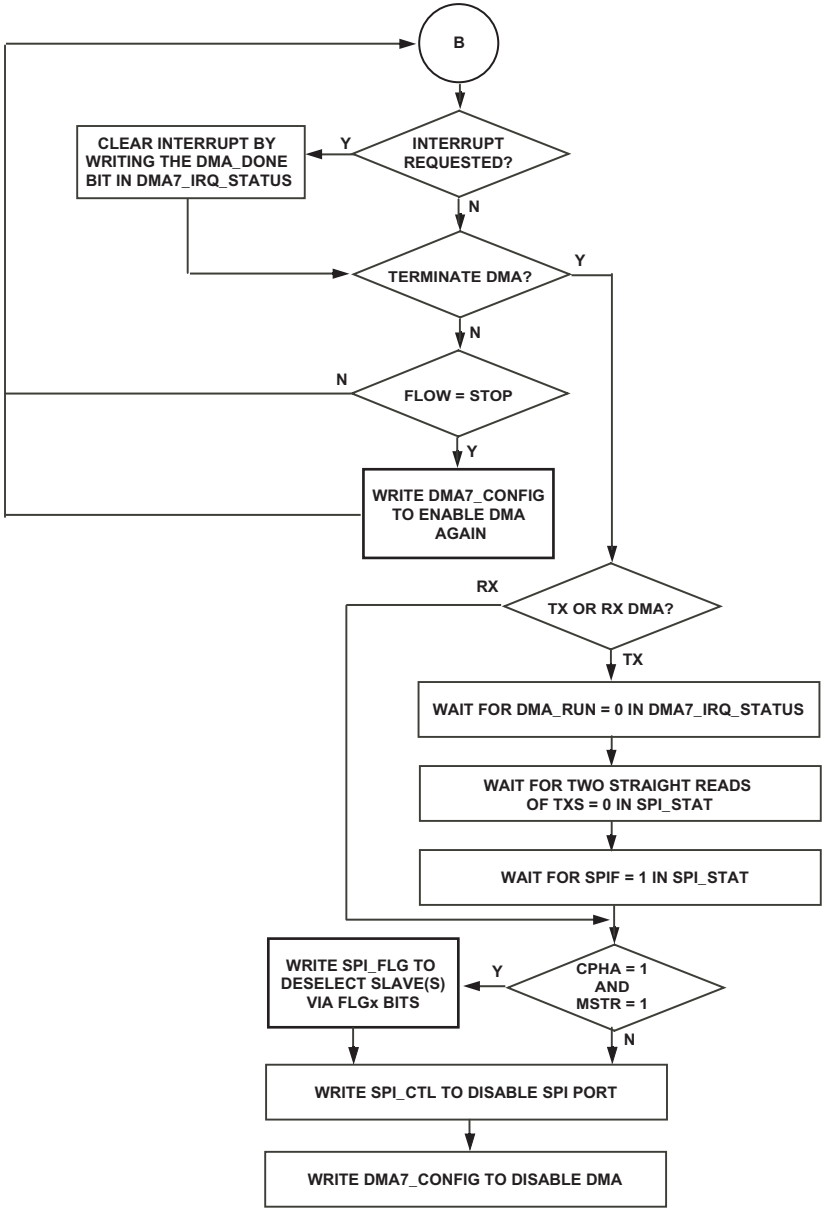


Figure 10-10. SPI DMA Flow Chart (Part 3 of 3)

SPI Registers

The SPI peripheral includes a number of user-accessible registers. Some of these registers are also accessible through the DMA bus. Four registers contain control and status information: SPI_BAUD, SPI_CTL, SPI_FLG, and SPI_STAT. Two registers are used for buffering receive and transmit data: SPI_RDBR and SPI_TDBR. For information about DMA-related registers, see Chapter 5, "Direct Memory Access". The shift register, SFDR, is internal to the SPI module and is not directly accessible.

See "Error Signals and Flags" on page 10-20 for more information about how the bits in these registers are used to signal errors and other conditions.

Table 10-6 shows the functions of the SPI registers. Figure 10-11 through Figure 10-17 on page 10-44 provide details.

Table 10-6. SPI Register Mapping

Register Name	Function	Notes
SPI_BAUD	SPI port baud control	Value of 0 or 1 disables the serial clock
SPI_CTL	SPI port control	SPE and MSTR bits can also be modified by hardware (when MODF is set)
SPI_FLG	SPI port flag	Bits 0 and 8 are reserved
SPI_STAT	SPI port status	SPIF bit can be set by clearing SPE in SPI_CTL
SPI_TDBR	SPI port transmit data buffer	Register contents can also be modified by hardware (by DMA and/or when SZ = 1 in SPI_CTL)
SPI_RDBR	SPI port receive data buffer	When register is read, hardware events can be triggered
SPI_SHADOW	SPI port data	Register has the same contents as SPI_RDBR, but no action is taken when it is read

SPI Baud Rate Register (SPI_BAUD)

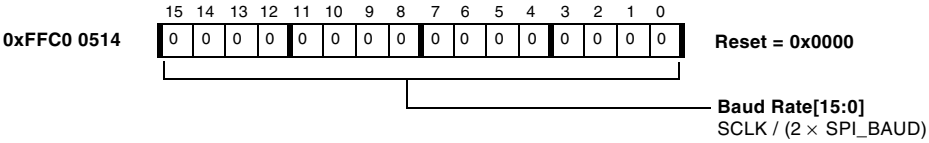


Figure 10-11. SPI Baud Rate Register

SPI Control Register (SPI_CTL)

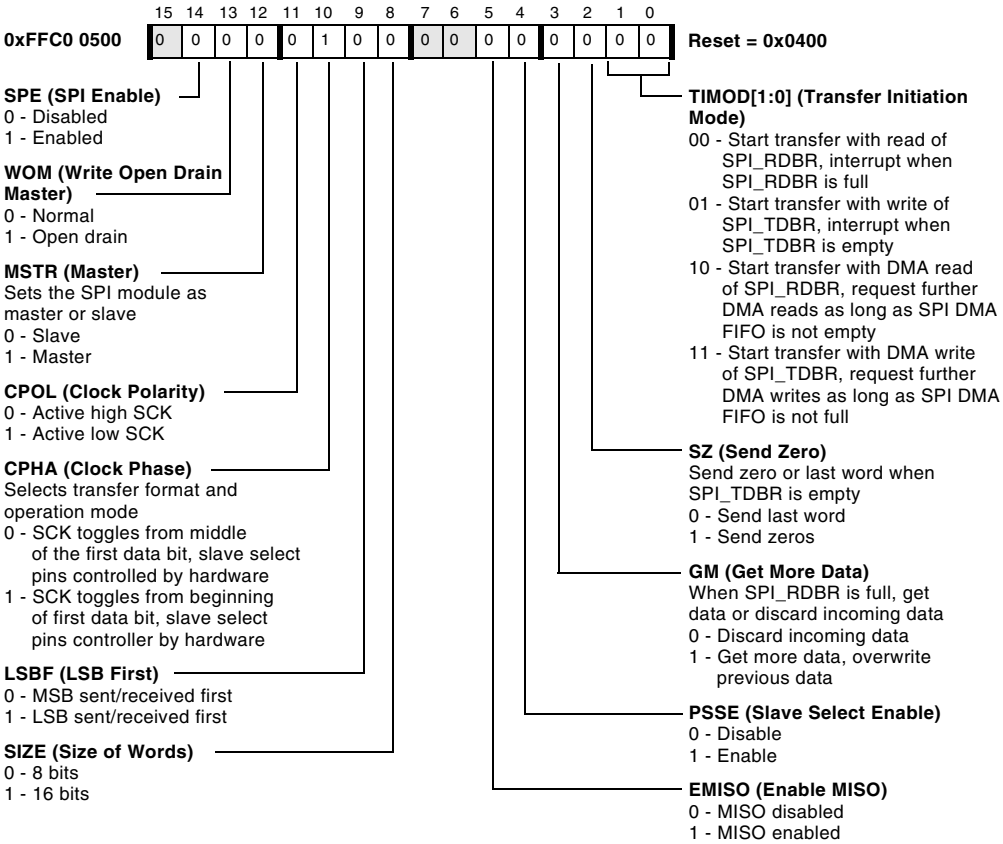


Figure 10-12. SPI Control Register

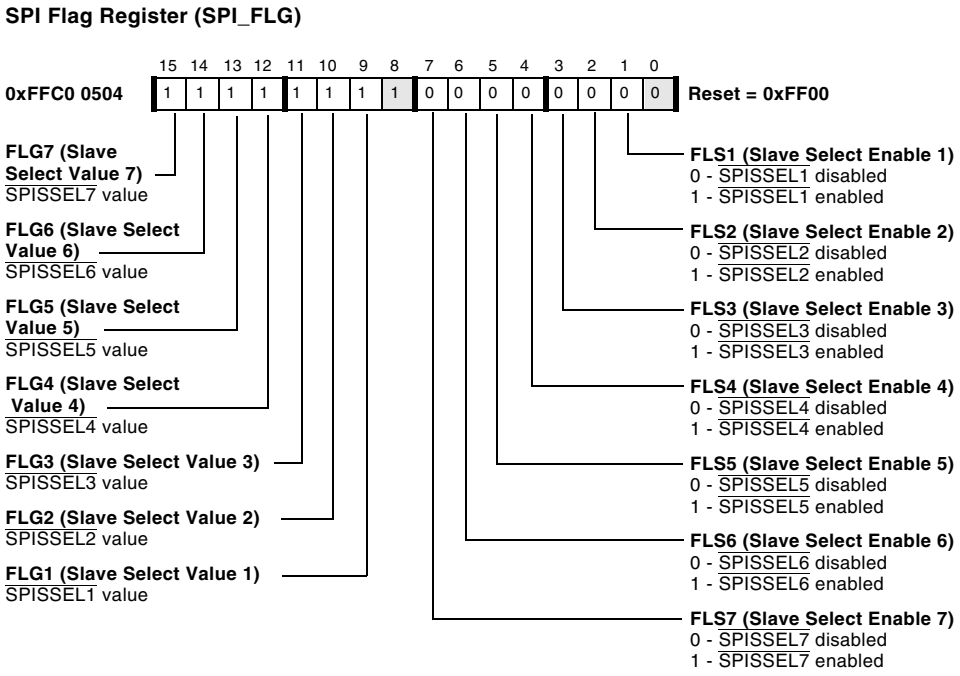


Figure 10-13. SPI Flag Register

The SPI_FLG register consists of two sets of bits that function as follows.

- Slave select enable (FLSx) bits

  Each FLSx bit corresponds to a general purpose port (PFx/PJx) pin. When an FLSx bit is set, the corresponding port pin is driven as a slave select. For example, if FLS1 is set in SPI_FLG, PF10 is driven as a slave select ($\overline{\text{SPISSEL1}}$). Table 10-2 on page 10-9 shows the association of the FLSx bits and the corresponding port pins.

  If the FLSx bit is not set, the general-purpose port registers (PORTFIO_DIR and others) configure and control the corresponding port pins.

- Slave select value (FLGx) bits

  When a PFx pin is configured as a slave select output, the FLGx bits can determine the value driven onto the output. If the CPHA bit in SPI_CTL is set, the output value is set by software control of the FLGx bits. The SPI protocol permits the slave select line to either remain asserted (low) or be deasserted between transferred words. The user must set or clear the appropriate FLGx bits. For example, to drive PJ10 as a slave select, FLS3 in SPI_FLG must be set. Clearing FLG3 in SPI_FLG drives PJ10 low; setting FLG3 drives PJ10 high. The PJ10 pin can be cycled high and low between transfers by setting and clearing FLG3. Otherwise, PJ10 remains active (low) between transfers.

  If CPHA = 0, the SPI hardware sets the output value and the FLGx bits are ignored. The SPI protocol requires that the slave select be deasserted between transferred words. In this case, the SPI hardware controls the pins. For example, to use PJ10 as a slave select pin, it is only necessary to set the FLS3 bit in SPI_FLG. It is not necessary to write to the FLG3 bit, because the SPI hardware automatically drives the PJ10 pin.

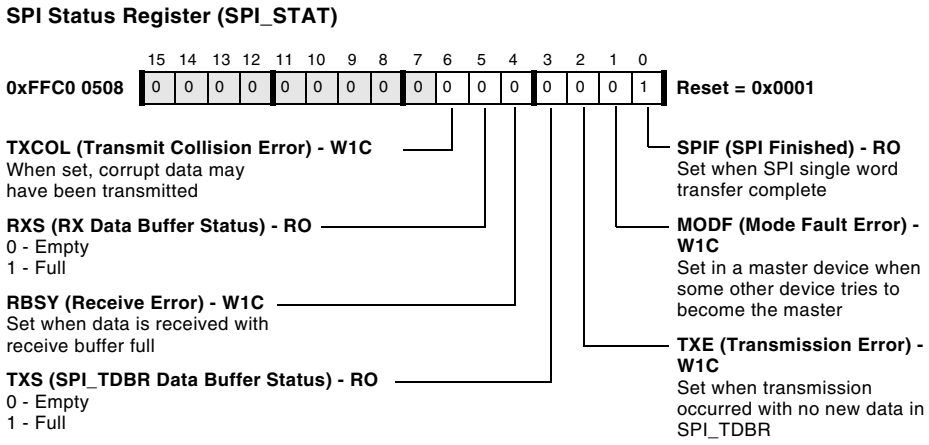


Figure 10-14. SPI Status Register

SPI Transmit Data Buffer Register (SPI_TDBR)

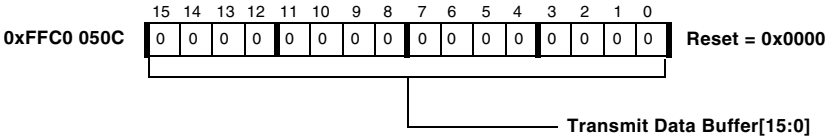


Figure 10-15. SPI Transmit Data Buffer Register

SPI Receive Data Buffer Register (SPI_RDBR)
RO

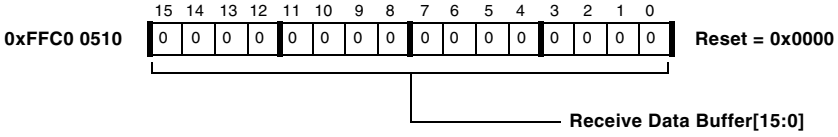


Figure 10-16. SPI Receive Data Buffer Register

SPI RDBR Shadow Register (SPI_SHADOW)
RO

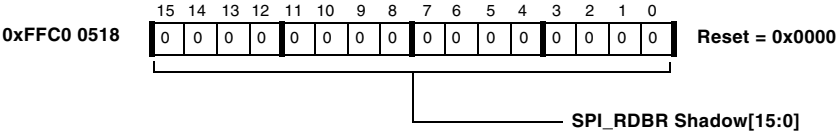


Figure 10-17. SPI RDBR Shadow Register

Programming Examples

This section includes examples (Listing 10-1 through Listing 10-8 on page 10-52) for core generated transfer and for use with DMA. Each code example assumes that the appropriate `defBF53x` header file is included.

Core Generated Transfer

The following core-driven master-mode SPI example shows how to initialize the hardware, signal the start of a transfer, handle the interrupt and issue the next transfer, and generate a stop condition.

Initialization Sequence

Before the SPI can transfer data, the registers must be configured as follows.

Listing 10-1. SPI Register Initialization

```
SPI_Register_Initialization:
  P0.H = hi(SPI_FLG);
  P0.L = lo(SPI_FLG);
  R0 = W[P0] (Z);
  BITSET (R0,0x7); /* FLS7 */
  W[P0] = R0; /* Enable slave-select output pin */

  P0.H = hi(SPI_BAUD);
  P0.L = lo(SPI_BAUD);
  R0.L = 0x208E;  /* Write to SPI Baud rate register */
  W[P0] = R0.L; ssync;  /* If SCLK = 133 MHz, SPI clock ~= 8 kHz
*/

/* Setup SPI Control Register */
/***********************************************
```

```
 * TIMOD [1:0] = 00 : Transfer On RDBR Read.
 * SZ [2]      =  0 : Send Last Word When TDBR0 Is Empty
 * GM [3]      =  1 : Discard Incoming Data If RDBR0 Is Full
 * PSSE [4]    =  0 : Disables Slave-Select As Input (Master)
 * EMISO [5]   =  0 : MISO Disabled For Output (Master)
 * [7] and [6] =  0 : RESERVED
 * SIZE [8]    =  1 : 16 Bit Word Length Select
 * LSBF [9]    =  0 : Transmit MSB First
 * CPHA [10]   =  0 : SCLK Starts Toggling At START Of First Data
Bit
 * CPOL [11]   =  1 : Active HIGH Serial Clock
 * MSTR [12]   =  1 : Device Is Master
 * WOM [13]    =  0 : Normal MOSI/MISO Data Output (No Open Drain)
 * SPE [14]    =  1 : SPI Module Is Enabled
 * [15]        =  0 : RESERVED
 ************************************************/
  P0.H = hi(SPI_CTL) ;
  P0.L = lo(SPI_CTL) ;
  R0 = 0x5908;
  W[P0] = R0.L; ssync; /* Enable SPI as MASTER */
```

Starting a Transfer

After the initialization procedure in the given master mode, a transfer begins following a dummy read of `SPI_RDBR`. Typically, known data which is desired to be transmitted to the slave is preloaded into the `SPI_TDBR`. In the following code, `P1` is assumed to point to the start of the 16-bit transmit data buffer and `P2` is assumed to point to the start of the 16-bit receive data buffer. In addition, the user must ensure appropriate interrupts are enabled for SPI operation.

Listing 10-2. Initiate Transfer

```
Initiate_Transfer:
  P0.H = hi(SPI_FLG);
  P0.L = lo(SPI_FLG);
  R0 = W[P0] (Z);
  BITCLR (R0,0xF);  /* FLG7 */
  W[P0] = R0;  /* Drive 0 on enabled slave-select pin */

  P0.H = hi(SPI_TDBR);  /* SPI Transmit Register */
  P0.L = lo(SPI_TDBR);
  R0 = W[P1++] (z);  /* Get First Data To Be Transmitted And
Increment Pointer */
 W[P0] = R0;  /* Write to SPI_TDBR */

  P0.H = hi(SPI_RDBR);
  P0.L = lo(SPI_RDBR);
  R0 = W[P0] (z);  /* Dummy read of SPI_RDBR kicks off transfer
*/
```

Post Transfer and Next Transfer

Following the transfer of data, the SPI generates an interrupt, which is serviced if the interrupt is enabled during initialization. In the interrupt routine, software must write the next value to be transmitted prior to reading the byte received. This is because a read of the `SPI_RDBR` initiates the next transfer.

Listing 10-3. SPI Interrupt Handler

```
SPI_Interrupt_Handler:
Process_SPI_Sample:
  P0.H = hi(SPI_TDBR);  /* SPI transmit register */
  P0.L = lo(SPI_TDBR);
```

```
  R0 = W[P1++](z);  /* Get next data to be transmitted */
  W[P0] = R0.l;  /* Write that data to SPI_TDBR */

Kick_Off_Next:
  P0.H = hi(SPI_RDBR);  /* SPI receive register */
  P0.L = lo(SPI_RDBR);
  R0 = W[P0] (z);  /* Read SPI receive register (also kicks off
next transfer) */
  W[P2++] = R0;  /* Store received data to memory */
  RTI;  /* Exit interrupt handler */
```

Stopping

In order for a data transfer to end after the user has transferred all data, the following code can be used to stop the SPI. Note that this is typically done in the interrupt handler to ensure the final data has been sent in its entirety.

Listing 10-4. Stopping SPI

```
Stopping_SPI:
  P0.H = hi(SPI_CTL);
  P0.L = lo(SPI_CTL);
  R0 = W[P0];
  BITCLR(R0, 14);  /* Clear SPI enable bit */
  W[P0] = R0.L; ssync;  /* Disable SPI */
```

DMA Transfer

The following DMA-driven master-mode SPI autobuffer example shows how to initialize DMA, initialize SPI, signal the start of a transfer, and generate a stop condition.

DMA Initialization Sequence

The following code initializes the DMA to perform a 16-bit memory read DMA operation in autobuffer mode, and generates an interrupt request when the buffer has been sent. This code assumes that P1 points to the start of the data buffer to be transmitted and that NUM_SAMPLES is a defined macro indicating the number of elements being sent.

Listing 10-5. DMA Initialization

```
Initialize_DMA: /* DMA7 = default channel for SPI DMA */
  P0.H = hi(DMA7_CONFIG);
  P0.L = lo(DMA7_CONFIG);
  R0 = 0x1084(z);  /* Autobuffer mode, IRQ on complete, linear
16-bit, mem read */
  w[P0] = R0;

  P0.H = hi(DMA7_START_ADDR);
  P0.L = lo(DMA7_START_ADDR);
  [p0] = p1;  /* Start address of TX buffer */

  P0.H  = hi(DMA7_X_COUNT);
  P0.L  = lo(DMA7_X_COUNT);
  R0    = NUM_SAMPLES;
  w[p0] = R0;  /* Number of samples to transfer */

  R0 = 2;
  P0.H = hi(DMA7_X_MODIFY);
  P0.L = lo(DMA7_X_MODIFY);
  w[p0] = R0;  /* 2 byte stride for 16-bit words */

  R0 = 1;  /* single dimension DMA means 1 row */
  P0.H = hi(DMA7_Y_COUNT);
  P0.L = lo(DMA7_Y_COUNT);
  w[p0] = R0;
```

SPI Initialization Sequence

Before the SPI can transfer data, the registers must be configured as follows.

Listing 10-6. SPI Initialization

```
SPI_Register_Initialization:
  P0.H = hi(SPI_FLG);
  P0.L = lo(SPI_FLG);
  R0 = W[P0] (Z);
  BITSET (R0,0x7); /* FLS7 */
  W[P0] = R0;  /* Enable slave-select output pin */

  P1.H = hi(SPI_BAUD);
  P1.L = lo(SPI_BAUD);
  R0.L   = 0x208E;  /* Write to SPI baud rate register */
  W[P0] = R0.L; ssync;  /* If SCLK = 133MHz, SPI clock ~= 8kHz */

  /* Setup SPI Control Register */
/***************************************************
 * TIMOD [1:0] = 11 : Transfer on DMA TDBR write
 * SZ [2]      =  0 : Send last word when TDBR0 is empty
 * GM [3]      =  1 : Discard incoming data if RDBR0 is full
 * PSSE [4]    =  0 : Disables slave-select as input (master)
 * EMISO [5]   =  0 : MISO disabled for output (master)
 * [7] and [6] =  0 : RESERVED
 * SIZE [8]    =  1 : 16 Bit word length select
 * LSBF [9]    =  0 : Transmit MSB first
 * CPHA [10]    =  0 : SCLK starts toggling at START of first data
bit
 * CPOL [11]   =  1 : Active HIGH serial clock
 * MSTR [12]   =  1 : Device is master
 * WOM [13]    =  0 : Normal MOSI/MISO data output (no open drain)
*
```

```
 * SPE [14]    =  1 : SPI module is enabled
 * [15]        =  0 : RESERVED
 ************************************************/
  /* Configure SPI as MASTER */
  R1 = 0x190B(z);  /* Leave disabled until DMA is enabled*/
  P1.L = lo(SPI_CTL);
  W[P1] = R1; ssync;
```

Starting a Transfer

After the initialization procedure in the given master mode, a transfer begins following enabling of SPI. However, the DMA must be enabled before enabling the SPI.

Listing 10-7. Starting a Transfer

```
Initiate_Transfer:
  P0.H = hi(DMA7_CONFIG);
  P0.L = lo(DMA7_CONFIG);
  R2 = w[P0](z);
  BITSET (R2, 0);  /*Set DMA enable bit */
  w[p0] = R2.L;  /* Enable TX DMA */

  P4.H = hi(SPI_CTL);
  P4.L = lo(SPI_CTL);
  R2=w[p4](z);
  BITSET (R2, 14);  /* Set SPI enable bit */
  w[p4] = R2;  /* Enable SPI */
```

Stopping a Transfer

In order for a data transfer to end after the DMA has transferred all required data, the following code is executed in the SPI DMA interrupt handler. The example code below clears the DMA interrupt, then waits for the DMA engine to stop running. When the DMA engine has

completed, SPI_STAT is polled to determine when the transmit buffer is empty. If there is data in the SPI Transmit FIFO, it is loaded as soon as the TXS bit clears. A second consecutive read with the TXS bit clear indicates the FIFO is empty and the last word is in the shift register. Finally, polling for the SPIF bit determines when the last bit of the last word has been shifted out. At that point, it is safe to shut down the SPI port and the DMA engine.

Listing 10-8. Stopping a Transfer

```
SPI_DMA_INTERRUPT_HANDLER:
  P0.L = lo(DMA7_IRQ_STATUS);
  P0.H = hi(DMA7_IRQ_STATUS);
  R0 = 1 ;
  W[P0] = R0 ;  /* Clear DMA interrupt */

  /* Wait for DMA to complete */
  P0.L = lo(DMA7_IRQ_STATUS);
  P0.H = hi(DMA7_IRQ_STATUS);
  R0 = DMA_RUN;  /* 0x08 */

CHECK_DMA_COMPLETE:  /* Poll for DMA_RUN bit to clear */
  R3 = W[P0] (Z);
  R1 = R3 & R0;
  CC = R1 == 0;
  IF !CC JUMP CHECK_DMA_COMPLETE;

  /* Wait for TXS to clear */
  P0.L = lo(SPI_STAT);
  P0.H = hi(SPI_STAT);
  R1 = TXS;  /* 0x08 */

Check_TXS:  /* Poll for TXS = 0 */
  R2 = W[P0] (Z);
```

```
  R2 = R2 & R1;
  CC = R0 == 0;
  IF !CC JUMP Check_TXS;

  R2 = W[P0] (Z);  /* Check if TXS stays clear for 2 reads */
  R2 = R2 & R1;
  CC = R0 == 0;
  IF !CC JUMP Check_TXS;

/* Wait for final word to transmit from SPI */
Final_Word:
  R0 = W[P0](Z);
  R2 = SPIF;  /* 0x01 */
  R0 = R0 & R2;
  CC = R0 == 0;
  IF !CC JUMP Final_Word;

Disable_SPI:
  P0.L = lo(SPI_CTL);
  P0.H = hi(SPI_CTL);
  R0 = W[P0] (Z);
  BITCLR (R0,0xe); /* Clear SPI enable bit */
  W[P0] = R0;  /* Disable SPI */

Disable_DMA:
  P0.L = lo(DMA7_CONFIG);
  P0.H = hi(DMA7_CONFIG);
  R0 = W[P0](Z);
  BITCLR (R0,0x0);  /* Clear DMA enable bit */
  W[P0] = R0;  /* Disable DMA */

  RTI;  /* Exit Handler */
```

11 TWO WIRE INTERFACE CONTROLLER

This chapter describes the Two Wire Interface (TWI) port. Following an overview and a list of key features is a description of operation and functional modes of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

Overview

The TWI controller allows a device to interface to an Inter IC bus as specified by the *Philips I^2C Bus Specification version 2.1* dated January 2000. This feature applies to the ADSP-BF534, ADSP-BF536, and ADSP-BF537 processors.

The TWI is fully compatible with the widely used I^2C bus standard. It was designed with a high level of functionality and is compatible with multi-master, multi-slave bus configurations. To preserve processor bandwidth the TWI controller can be set up with transfer initiated 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 I^2C bus protocol. The TWI controller includes these features:

- Simultaneous master and slave operation on multiple device systems
- Support for multi-master bus arbitration
- 7-bit addressing

- 100 kbits/second and 400 kbits/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 lock-up
- 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*.

Interface Overview

Figure 11-1 provides a block diagram of the TWI controller. 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. The SCL synchronizes the shifting and sampling of the data on the serial data pin.

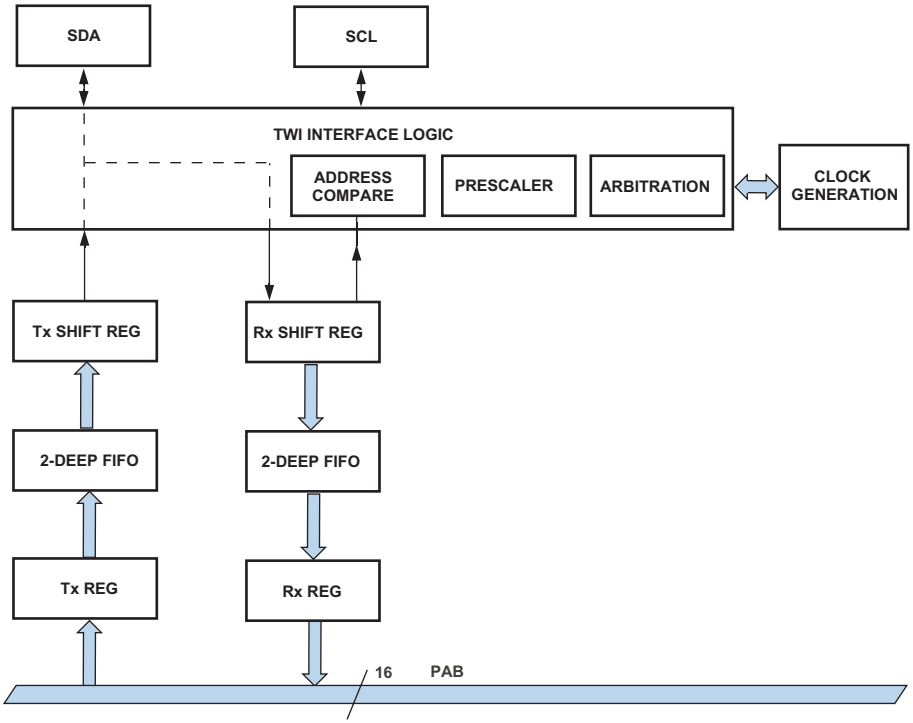


Figure 11-1. TWI Block Diagram

External Interface

The TWI signals are dedicated to this interface, that is, they are not multiplexed with any other signals. These signals, SDA (serial data) and SCL (serial clock) are open drain and as such require pull up resistors.

Serial Clock signal (SCL)

In slave mode this signal is an input and an external master is responsible for providing the clock.

In master mode the TWI controller must set this signal to the desired frequency. The TWI controller supports the standard mode of operation (up to 100 KHz) or fast mode (up to 400 KHz).

The TWI control register (TWI_CONTROL) is used to set the PRESCALE value which gives the relationship between the system clock (SCLK) and the TWI controller's internally timed events. The internal time reference is derived from SCLK using a prescaled value.

$$PRESCALE = f_{SCLK}/10MHz$$

The PRESCALE value is the number of system clock (SCLK) periods used in the generation of one internal time reference. The value of PRESCALE must be set to create an internal time reference with a period of 10 MHz. It is represented as a 7-bit binary value.

Serial data signal (SDA)

This is a bidirectional signal on which serial data is transmitted or received depending on the direction of the transfer.

TWI Pins

Table 11-1 shows the pins for the TWI. Two bidirectional pins externally interface the TWI controller to the I^2C bus. The interface is simple and no other external connections or logic are required.

Table 11-1. TWI Pins

Pin	Description
SDA	In/Out TWI serial data, high impedance reset value.
SCL	In/Out TWI serial clock, high impedance reset value.

Internal Interfaces

The peripheral bus interface supports the transfer of 16-bit wide data and is used by the processor in the support of register and FIFO buffer reads and writes.

The register block contains all control and status bits and reflects what can be written or read as outlined by the programmer's model. Status bits can be updated by their respective functional blocks.

The FIFO buffer is configured as a1-byte-wide 2-deep transmit FIFO buffer and a 1-byte-wide 2-deep receive FIFO buffer.

The transmit shift register serially shifts its data out externally off chip. The output can be controlled for generation of acknowledgements or it can be manually overwritten.

The receive shift register receives its data serially from off chip. The receive shift register is 1 byte wide and data received can either be transferred to the FIFO buffer or used in an address comparison.

The address compare block supports address comparison in the event the TWI controller module is accessed as a slave.

The prescaler block must be programmed to generate a 10 MHz time reference relative to the system clock. This time base is used for filtering of data and timing events specified by the electrical data sheet (See the Philips Specification), as well as for `SCL` clock generation.

The clock generation module is used to generate an external `SCL` clock when in master mode. It includes the logic necessary for synchronization in a multi-master clock configuration and clock stretching when configured in slave mode.

Description of Operation

The following sections describe the operation of the TWI interface.

TWI Transfer Protocols

The TWI controller follows the transfer protocol of the *Philips I²C Bus Specification version 2.1* dated January 2000. A simple complete transfer is diagrammed in Figure 11-2.

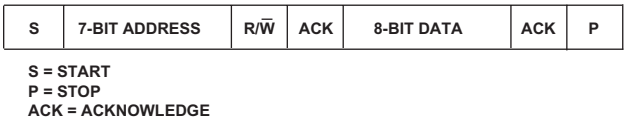


Figure 11-2. Basic Data Transfer

To better understand the mapping of TWI controller register contents to a basic transfer, Figure 11-3 details the same transfer as above noting the corresponding TWI controller bit names. In this illustration, the TWI controller successfully transmits one byte of data. The slave has acknowledged both address and data.

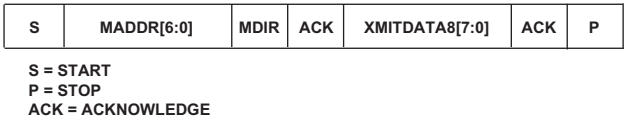


Figure 11-3. Data Transfer With Bit Illustration

Clock Generation and Synchronization

The TWI controller implementation only issues a clock during master mode operation and only at the time a transfer has been initiated. If arbitration for the bus is lost, the serial clock output immediately three-states.

If multiple clocks attempt to drive the serial clock line, the TWI controller synchronizes its clock with the other remaining clocks. This is shown in Figure 11-4.

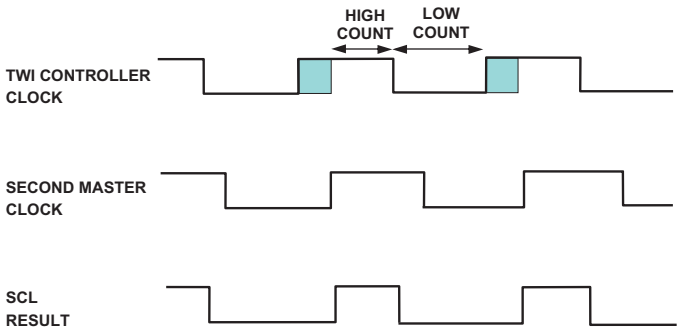


Figure 11-4. TWI Clock Synchronization

The TWI controller's serial clock (SCL) output follows these rules:

- Once the clock high (CLKHI) count is complete, the serial clock output is driven low and the clock low (CLKLOW) count begins.
- Once the clock low count is complete, the serial clock line is three-stated and the clock synchronization logic enters into a delay mode (shaded area) until the SCL line is detected at a logic 1 level. At this time the clock high count begins.

Bus Arbitration

The TWI controller initiates a master mode transmission (MEN) only when the bus is idle. If the bus is idle and two masters initiate a transfer, arbitration for the bus begins. This is shown in Figure 11-5.

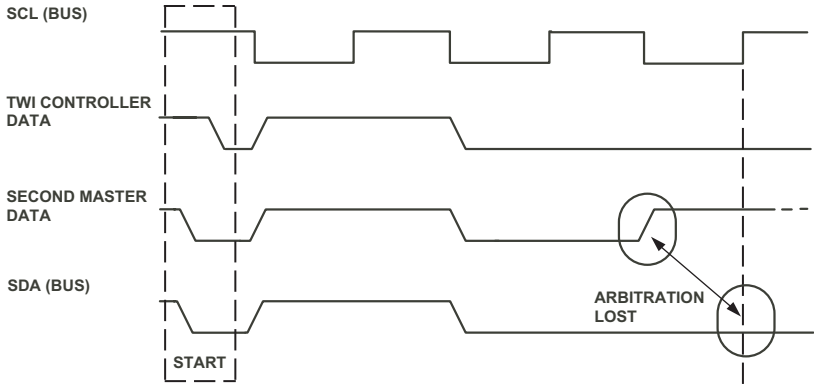


Figure 11-5. TWI Bus Arbitration

The TWI controller monitors the serial data bus (SDA) while SCL is high and if SDA is determined to be an active logic 0 level while the TWI controller's data is a logic 1 level, the TWI controller has lost arbitration and ends generation of clock and data. Note arbitration is not performed only at serial clock edges, but also during the entire time SCL is high.

Start and Stop Conditions

Start and stop conditions involve serial data transitions while the serial clock is a logic 1 level. The TWI controller generates and recognizes these transitions. Typically start and stop conditions occur at the beginning and at the conclusion of a transmission with the exception repeated start "combined" transfers, as shown in Figure 11-6.

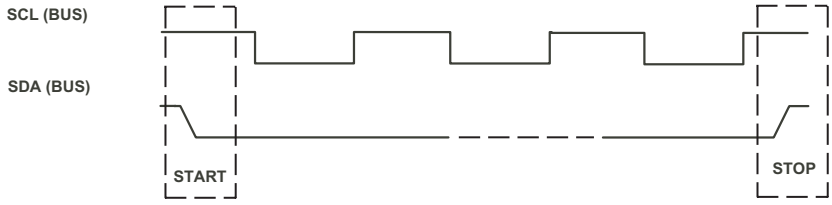


Figure 11-6. TWI Start and Stop Conditions

The TWI controller's special case start and stop conditions include:

- TWI controller addressed as a slave-receiver

  If the master asserts a stop condition during the data phase of a transfer, the TWI controller concludes the transfer (SCOMP).

- TWI controller addressed as a slave-transmitter

  If the master asserts a stop condition during the data phase of a transfer, the TWI controller concludes the transfer (SCOMP) and indicates a slave transfer error (SERR).

- TWI controller as a master-transmitter or master-receiver

  If the stop bit is set during an active master transfer, the TWI controller issues a stop condition as soon as possible avoiding any error conditions (as if data transfer count had been reached).

General Call Support

The TWI controller always decodes and acknowledges a general call address if it is enabled as a slave (SEN) and if general call is enabled (GEN). general call addressing (0x00) is indicated by the GCALL bit being set and by nature of the transfer the TWI controller is a slave-receiver. If the data associated with the transfer is to be NAK'ed, the NAK bit can be set.

If the TWI controller is to issue a general call as a master-transmitter the appropriate address and transfer direction can be set along with loading transmit FIFO data.

Fast Mode

Fast mode essentially uses the same mechanics as standard mode of operation. It is the electrical specifications and timing that are most effected. When fast mode is enabled (FAST) the following timings are modified to meet the electrical requirements.

- Serial data rise times before arbitration evaluation (t_r)
- Stop condition set-up time from serial clock to serial data ($t_{SU;STO}$)
- Bus free time between a stop and start condition (t_{BUF})

TWI General Operation

The following sections describe the general operation of the TWI.

TWI Control

The TWI control register (TWI_CONTROL) is used to enable the TWI module as well as to establish a relationship between the system clock (SCLK) and the TWI controller's internally timed events. The internal time reference is derived from SCLK using a prescaled value.

$$PRESCALE = f_{SCLK}/10MHz$$

SCCB compatibility is an optional feature and should not be used in an I^2C bus system. This feature is turned on by setting the SCCB bit in the TWI_CONTROL register. When this feature is set all slave asserted acknowledgement bits are ignored by this master. This feature is valid only during transfers where the TWI is mastering an SCCB bus. Slave mode transfers should be avoided when this feature is enabled because the TWI controller always generates an acknowledge in slave mode.

For either master and/or slave mode of operation, the TWI controller is enabled by setting the TWI_ENA bit in the TWI_CONTROL register. It is recommended that this bit be set at the time PRESCALE is initialized and remain set. This guarantees accurate operation of bus busy detection logic.

The PRESCALE field of the TWI_CONTROL register specifies the number of system clock (SCLK) periods used in the generation of one internal time reference. The value of PRESCALE must be set to create an internal time reference with a period of 10 MHz. It is represented as a 7-bit binary value.

Clock Signal

The clock signal SCL is an output in master mode and an input in slave mode.

During master mode operation, the SCL clock divider register (TWI_CLKDIV) values are used to create the high and low durations of the serial clock (SCL). Serial clock frequencies can vary from 400 KHz to less than 20 KHz. The resolution of the clock generated is 1/10 MHz or 100 ns.

```
CLKDIV = TWI SCL period / 10 MHz time reference
```

For example, for an SCL of 400 KHz (period = 1/400 KHz = 2500 ns) and an internal time reference of 10 MHz (period = 100 ns):

```
CLKDIV = 2500 ns / 100 ns = 25
```

For an SCL with a 30% duty cycle, then CLKLOW = 17 and CLKHI = 8. Note that CLKLOW and CLKHI add up to CLKDIV.

The clock high field of the TWI_CLKDIV register specifies the number of 10 MHz time reference periods the serial clock (SCL) waits before a new clock low period begins, assuming a single master. It is represented as an 8-bit binary value.

The clock low field of the TWI_CLKDIV register number of internal time reference periods the serial clock (SCL) is held low. It is represented as an 8-bit binary value.

Error Signals and Flags

The following sections describe the TWI error signals and flags.

TWI Master Status

The TWI master mode status register (TWI_MASTER_STAT) holds information during master mode transfers and at their conclusion. Generally, master mode status bits are not directly associated with the generation of interrupts but offer information on the current transfer. Slave mode operation does not affect master mode status bits.

- **Bus busy** (BUSBUSY)

  Indicates whether the bus is currently busy or free. This indication is not limited to only this device but is for all devices. Upon a start condition, the setting of the register value is delayed due to the input filtering. Upon a stop condition the clearing of the register value occurs after t_{BUF}.

  [1] The bus is busy. Clock or data activity has been detected.

  [0] The bus is free. The clock and data bus signals have been inactive for the appropriate bus free time.

- **Serial clock sense** (SCLSEN)

  This status bit can be used when direct sensing of the serial clock line is required. The register value is delayed due to the input filter (nominally 50 ns). Normal master and slave mode operation should not require this feature.

[1] An active “zero” is currently being sensed on the serial clock. The source of the active driver is not known and can be internal or external.

[0] An inactive “one” is currently being sensed on the serial clock.

- **Serial data sense** (SDASEN)

  This status bit can be used when direct sensing of the serial data line is required. The register value is delayed due to the input filter (nominally 50 ns). Normal master and slave mode operation should not require this feature.

  [1] An active “zero” is currently being sensed on the serial data line. The source of the active driver is not known and can be internal or external.

  [0] An inactive “one” is currently being sensed on the serial data line.

- **Buffer write error** (BUFWRERR)

  [1] The current master transfer was aborted due to a receive buffer write error. The receive buffer and receive shift register were both full at the same time. This bit is W1C.

  [0] The current master receive has not detected a receive buffer write error.

- **Buffer read error** (BUFRDERR)

  [1] The current master transfer was aborted due to a transmit buffer read error. At the time data was required by the transmit shift register the buffer was empty. This bit is W1C.

  [0] The current master transmit has not detected a buffer read error.

- **Data not acknowledged** (DNAK)

  [1] The current master transfer was aborted due to the detection of a NAK during data transmission. This bit is W1C.

  [0] The current master receive has not detected a NAK during data transmission.

- **Address not acknowledged** (ANAK)

  [1] The current master transfer was aborted due to the detection of a NAK during the address phase of the transfer. This bit is W1C.

  [0] The current master transmit has not detected NAK during addressing.

- **Lost arbitration** (LOSTARB)

  [1] The current transfer was aborted due to the loss of arbitration with another master. This bit is W1C.

  [0] The current transfer has not lost arbitration with another master.

- **Master transfer in progress** (MPROG)

  [1] A master transfer is in progress.

  [0] Currently no transfer is taking place. This can occur once a transfer is complete or while an enabled master is waiting for an idle bus.

TWI Slave Status

During and at the conclusion of slave mode transfers, the TWI slave mode status register (TWI_SLAVE_STAT) holds information on the current transfer. Generally slave mode status bits are not associated with the generation of interrupts. Master mode operation does not affect slave mode status bits.

- **General call** (GCALL)

  This bit self clears if slave mode is disabled (SEN = 0).

  [1] At the time of addressing, the address was determined to be a general call.

  [0] At the time of addressing, the address was not determined to be a general call.

- **Slave transfer direction** (SDIR)

  This bit self clears if slave mode is disabled (SEN = 0).

  [1] At the time of addressing, the transfer direction was determined to be slave transmit.

  [0] At the time of addressing, the transfer direction was determined to be slave receive.

TWI FIFO Status

The fields in the TWI FIFO status register (TWI_FIFO_STAT) indicate the state of the FIFO buffers' receive and transmit contents. The FIFO buffers do not discriminate between master data and slave data. By using the status and control bits provided, the FIFO can be managed to allow simultaneous master and slave operation.

- **Receive FIFO status** (RCVSTAT[1:0])

  The RCVSTAT field is read only. It indicates the number of valid data bytes in the receive FIFO buffer. The status is updated with each FIFO buffer read using the peripheral data bus or write access by the receive shift register. Simultaneous accesses are allowed.

  [11] The FIFO is full and contains two bytes of data. Either a single or double byte peripheral read of the FIFO is allowed.

  [10] Reserved

  [01] The FIFO contains one byte of data. A single byte peripheral read of the FIFO is allowed.

  [00] The FIFO is empty.

- **Transmit FIFO status** (XMTSTAT[1:0])

  The XMTSTAT field is read only. It indicates the number of valid data bytes in the FIFO buffer. The status is updated with each FIFO buffer write using the peripheral data bus or read access by the transmit shift register. Simultaneous accesses are allowed.

  [11] The FIFO is full and contains two bytes of data.

  [10] Reserved

  [01] The FIFO contains one byte of data. A single byte peripheral write of the FIFO is allowed.

  [00] The FIFO is empty. Either a single or double byte peripheral write of the FIFO is allowed.

TWI Interrupt Status

The TWI interrupt status register (TWI_INT_STAT) contains information about functional areas requiring servicing. Many of the bits serve as an indicator to further read and service various status registers. After servicing the interrupt source associated with a bit, the user must clear that interrupt source bit by writing a 1 to it.

- **Receive FIFO service** (RCVSERV)

  If RCVINTLEN in the TWI_FIFO_CTL register is 0, this bit is set each time the RCVSTAT field in the TWI_FIFO_STAT register is updated to either 01 or 11. If RCVINTLEN is 1, this bit is set each time RCVSTAT is updated to or 11.

  [1] The FIFO does not require servicing or the RCVSTAT field has not changed since this bit was last cleared.

  [0] No errors have been detected.

- **Transmit FIFO service** (XMTSERV)

  If XMTINTLEN in the TWI_FIFO_CTL register is 0, this bit is set each time the XMTSTAT field in the TWI_FIFO_STAT register is updated to either 01 or 00. If XMTINTLEN is 1, this bit is set each time XMTSTAT is updated to 00.

  [1] The transmit FIFO buffer has one or two 8-bit locations available to be written.

  [0] FIFO does not require servicing or XMTSTAT field has not changed since this bit was last cleared.

- **Master transfer error** (MERR)

  [1] A master error has occurred. The conditions surrounding the error are indicated by the master status register (TWI_MASTER_STAT).

  [0] No errors have been detected.

- **Master transfer complete** (MCOMP)

  [1] The initiated master transfer has completed. In the absence of a repeat start, the bus has been released.

  [0] The completion of a transfer has not been detected.

- **Slave overflow** (SOVF)

  [1] The slave transfer complete (SCOMP) bit was set at the time a subsequent transfer has acknowledged an address phase. The transfer continues, however, it may be difficult to delineate data of one transfer from another.

  [0] No overflow has been detected.

- **Slave transfer error** (SERR)

  [1] A slave error has occurred. A restart or stop condition has occurred during the data receive phase of a transfer.

  [0] No errors have been detected.

- **Slave transfer complete** (SCOMP)

  [1] The transfer is complete and either a stop, or a restart was detected.

  [0] The completion of a transfer has not been detected.

- **Slave transfer initiated** (SINIT)

  [1] The slave has detected an address match and a transfer has been initiated.

  [0] A transfer is not in progress. An address match has not occurred since the last time this bit was cleared.

Functional Description

The following sections describe the functional operation of the TWI.

General Setup

General setup refers to register writes that are required for both slave mode operation and master mode operation. General setup should be performed before either the master or slave enable bits are set.

- Program the TWI_CONTROL register to enable the TWI controller and set the prescale value. Program the prescale value to the binary representation of f_{SCLK} / 10MHz

All values should be rounded up to the next whole number. The TWI_ENA bit enable must be set. Note once the TWI controller is enabled a bus busy condition may be detected. This condition should clear after t_{BUF} has expired assuming no additional bus activity has been detected.

Slave Mode

When enabled, slave mode operation supports both receive and transmit data transfers. It is not possible to enable only one data transfer direction and not acknowledge (NAK) the other. This is reflected in the following setup.

1. Program TWI_SLAVE_ADDR. The appropriate 7 bits are used in determining a match during the address phase of the transfer.
2. Program TWI_XMT_DATA8 or TWI_XMT_DATA16. These are the initial data values to be transmitted in the event the slave is addressed and a transmit is required. This is an optional step. If no data is written and the slave is addressed and a transmit is required, the serial clock (SCL) is stretched and an interrupt is generated until data is written to the transmit FIFO.

3. Program TWI_INT_MASK. Enable bits are associated with the desired interrupt sources. As an example, programming the value 0x000F results in an interrupt output to the processor in the event that a valid address match is detected, a valid slave transfer completes, a slave transfer has an error, a subsequent transfer has begun yet the previous transfer has not been serviced.

4. Program TWI_SLAVE_CTL. Ultimately this prepares and enables slave mode operation. As an example, programming the value 0x0005 enables slave mode operation, requires 7-bit addressing, and indicates that data in the transmit FIFO buffer is intended for slave mode transmission.

Table 11-2 shows what the interaction between the TWI controller and the processor might look like using this example.

Table 11-2. Slave Mode Setup Interaction

TWI Controller Master	Processor
Interrupt: SINIT – Slave transfer in progress.	Acknowledge: Clear interrupt source bits.
Interrupt: RCVFULL – Receive buffer is full.	Read receive FIFO buffer. Acknowledge: Clear interrupt source bits.
...	...
Interrupt: SCOMP – Slave transfer complete.	Read receive FIFO buffer. Acknowledge: Clear interrupt source bits.

Master Mode Clock Setup

Master mode operation is set up and executed on a per-transfer basis. An example of programming steps for a receive and for a transmit are given separately in following sections. The clock setup programming step listed here is common to both transfer types.

- Program TWI_CLKDIV. This defines the clock high duration and clock low duration.

Master Mode Transmit

Follow these programming steps for a single master mode transmit:

1. Program TWI_MASTER_ADDR. This defines the address transmitted during the address phase of the transfer.
2. Program TWI_XMT_DATA8 or TWI_XMT_DATA16. This is the initial data transmitted. It is considered an error to complete the address phase of the transfer and not have data available in the transmit FIFO buffer.
3. Program TWI_FIFO_CTL. Indicate if transmit FIFO buffer interrupts should occur with each byte transmitted (8 bits) or with each 2 bytes transmitted (16 bits).
4. Program TWI_INT_MASK. Enable bits associated with the desired interrupt sources. As an example, programming the value 0x0030 results in an interrupt output to the processor in the event that the master transfer completes, and the master transfer has an error.
5. Program TWI_MASTER_CTL. Ultimately this prepares and enables master mode operation. As an example, programming the value 0x0201 enables master mode operation, generates a 7-bit address, sets the direction to master-transmit, uses standard mode timing, and transmits 8 data bytes before generating a Stop condition.

Table 11-3 shows what the interaction between the TWI controller and the processor might look like using this example.

Table 11-3. Master Mode Transmit Setup Interaction

TWI Controller Master	Processor
Interrupt: XMTEMPTY – Transmit buffer is empty.	Write transmit FIFO buffer. Acknowledge: Clear interrupt source bits.
...	...
Interrupt: MCOMP – Master transfer complete.	Acknowledge: Clear interrupt source bits.

Master Mode Receive

Follow these programming steps for a single master mode receive:

1. Program `TWI_MASTER_ADDR`. This defines the address transmitted during the address phase of the transfer.
2. Program `TWI_FIFO_CTL`. Indicate if receive FIFO buffer interrupts should occur with each byte received (8 bits) or with each 2 bytes received (16 bits).
3. Program `TWI_INT_MASK`. Enable bits associated with the desired interrupt sources. For example, programming the value 0x0030 results in an interrupt output to the processor in the event that the master transfer completes, and the master transfer has an error.
4. Program `TWI_MASTER_CTL`. Ultimately this prepares and enables master mode operation. As an example, programming the value 0x0205 enables master mode operation, generates a 7-bit address, sets the direction to master-receive, uses standard mode timing, and receives 8 data bytes before generating a Stop condition.

Table 11-4 shows what the interaction between the TWI controller and the processor might look like using this example.

Table 11-4. Master Mode Receive Setup Interaction

TWI Controller Master	Processor
Interrupt: RCVFULL – Receive buffer is full.	Read receive FIFO buffer. Acknowledge: Clear interrupt source bits.
...	...
Interrupt: MCOMP – Master transfer complete.	Acknowledge: Clear interrupt source bits. Read receive FIFO buffer.

Repeated Start Condition

In general, a repeated start condition is the absence of a stop condition between two transfers. The two transfers can be of any direction type. Examples include a transmit followed by a receive, or a receive followed by a transmit. During a repeated start transfer, each interrupt must be serviced correctly to avoid errors. The following sections contain information intended to be a guide to assist the programmer in their service routine development.

Transmit/Receive Repeated Start Sequence

Figure 11-7 illustrates a repeated start data transmit followed by a data receive sequence.

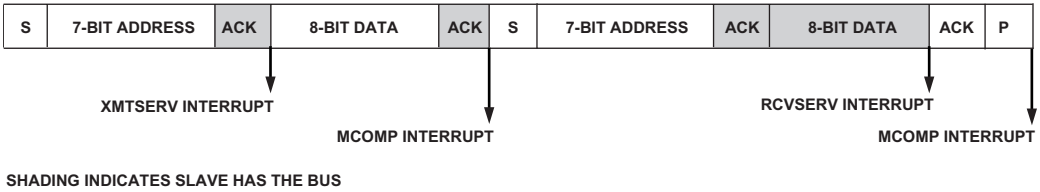


Figure 11-7. Transmit/Receive Data Repeated Start

The tasks performed at each interrupt are:

- `XMTSERV` interrupt

  This interrupt was generated due to a FIFO access. Since this is the last byte of this transfer, `FIFO_STATUS` would indicate the transmit FIFO is empty. When read, `DCNT` would be zero. At this time `RSTART` should be set to indicate a repeated start should be issued and `MDIR` should be set to indicate the subsequent transfer should be a data receive.

- `MCOMP` interrupt

  This interrupt was generated since all data has been transferred (`DCNT = 0`). If no errors were generated, a start condition is initiated. At this time `RSTART` should be cleared and `DCNT` should be programmed with the desired number of bytes to receive.

- `RCVSERV` interrupt

  This interrupt is generated due to the arrival of a byte into the receive FIFO. Simple data handling is all that is required.

- `MCOMP` interrupt

  The transfer is complete.

Receive/Transmit Repeated Start Sequence

Figure 11-8 illustrates a repeated start data receive followed by a data transmit sequence.

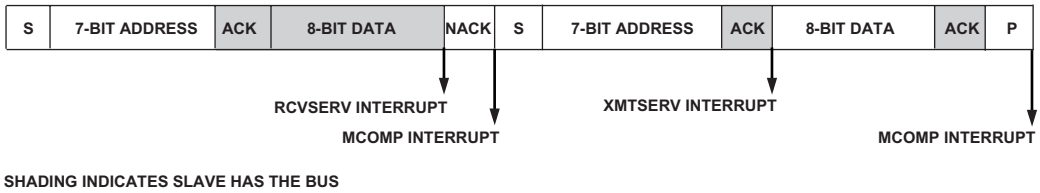


Figure 11-8. Receive/Transmit Data Repeated Start

If more than one byte is being received, the RCVSERV interrupt should be performed after the first data byte received. This allows for the largest interrupt latency and prevents a control bit such as MDIR from being set too late

The tasks performed at each interrupt are:

- RCVSERV interrupt

  This interrupt in generated due to the arrival of a data byte into the receive FIFO. The RSTART bit should be set at this time (or earlier) and MDIR should be cleared to reflect the change in direction of the next transfer. The MDIR bit must be cleared before the addressing phase of the subsequent transfer begins.

- MCOMP interrupt

  This interrupt has occurred due to the completion of the data receive transfer. At this time the data transmit transfer has begun. The DCNT field should be set to reflect the number of bytes to be transmitted. Clear RSTART if this is the last transfer.

- `XMTSERV` interrupt

  This interrupt is generated due to a FIFO access. Simple data handling is all that is required.

- `MCOMP` interrupt

  The transfer is complete.

Programming Model

Figure 11-9 and Figure 11-10 illustrate the programming model for the TWI.

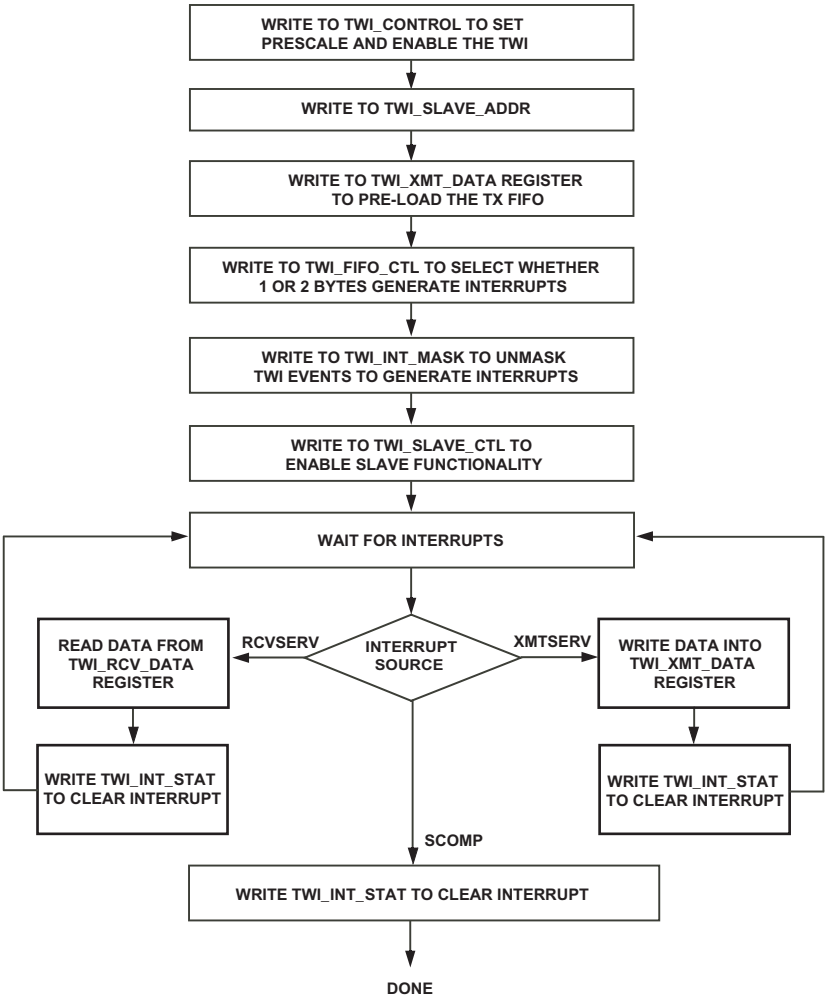


Figure 11-9. TWI Slave Mode

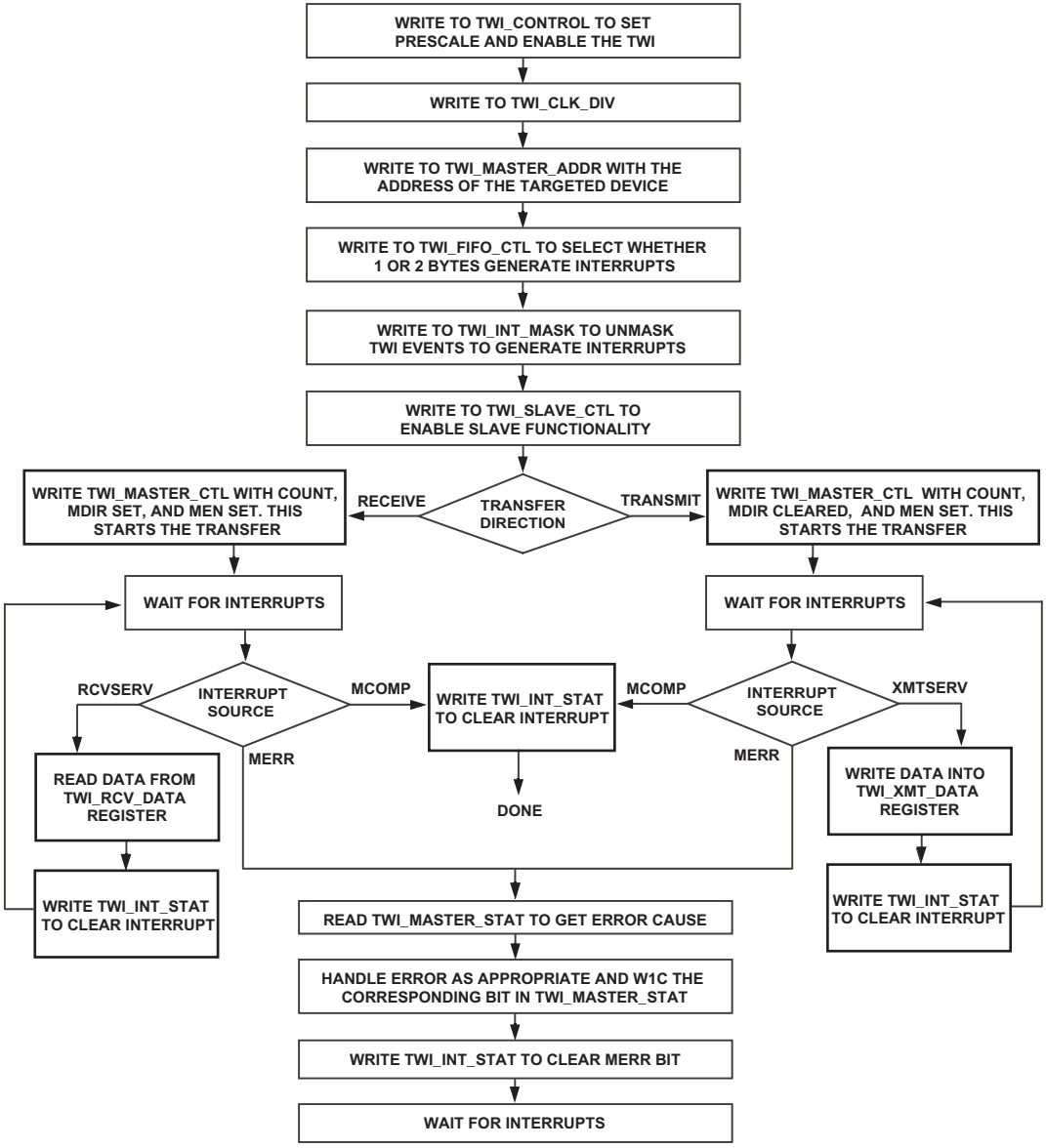


Figure 11-10. TWI Master Mode

Register Descriptions

The TWI controller has 16 registers described in the following sections. Figure 11-11 through Figure 11-28 on page 11-47 illustrate the registers.

TWI_CONTROL Register

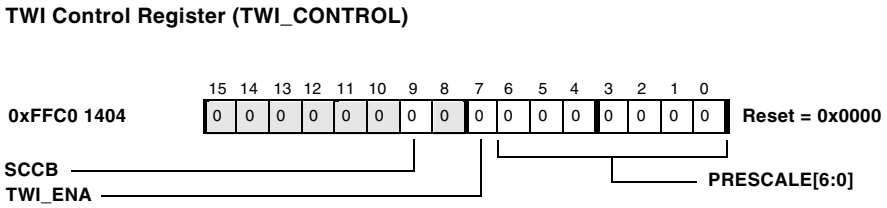


Figure 11-11. TWI Control Register

TWI_CLKDIV Register

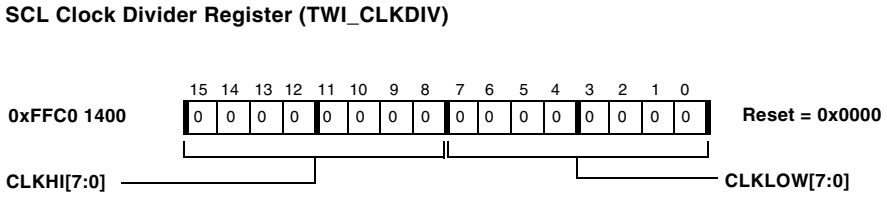


Figure 11-12. SCL Clock Divider Register

TWI_SLAVE_CTL Register

The TWI slave mode control register (TWI_SLAVE_CTL) controls the logic associated with slave mode operation. Settings in this register do not affect master mode operation and should not be modified to control master mode functionality.

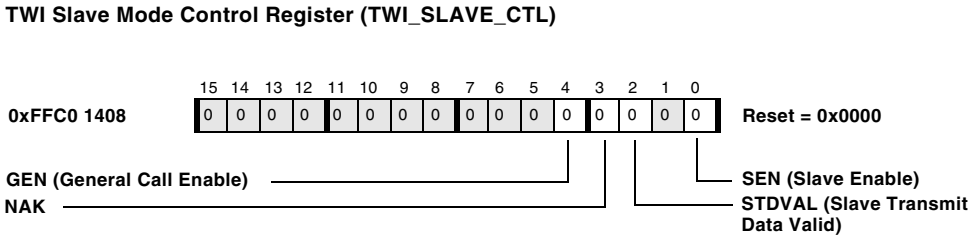


Figure 11-13. TWI Slave Mode Control Register

Additional information for the TWI_SLAVE_CTL register bits includes:

- **General call enable** (GEN)

  General call address detection is available only when slave mode is enabled.

  [1] General call address matching is enabled. A general call slave receive transfer is accepted. All status and interrupt source bits associated with transfers are updated.

  [0] General call address matching is not enabled.

- **NAK** (NAK)

  [1] Slave receive transfers generate a data NAK (not acknowledge) at the conclusion of a data transfer. The slave is still considered to be addressed.

  [0] Slave receive transfers generate an ACK at the conclusion of a data transfer.

- **Slave transmit data valid** (STDVAL)

  [1] Data in the transmit FIFO is available for a slave transmission.

  [0] Data in the transmit FIFO is for master mode transmits and is not allowed to be used during a slave transmit, and the transmit FIFO is treated as if it is empty.

- **Slave enable** (SEN)

  [1] The slave is enabled. Enabling slave and master modes of operation concurrently is allowed.

  [0] The slave is not enabled. No attempt is made to identify a valid address. If cleared during a valid transfer, clock stretching ceases, the serial data line is released, and the current byte is not acknowledged.

TWI_SLAVE_ADDR Register

The TWI slave mode address register (TWI_SLAVE_ADDR) holds the slave mode address, which is the valid address that the slave-enabled TWI controller responds to. The TWI controller compares this value with the received address during the addressing phase of a transfer.

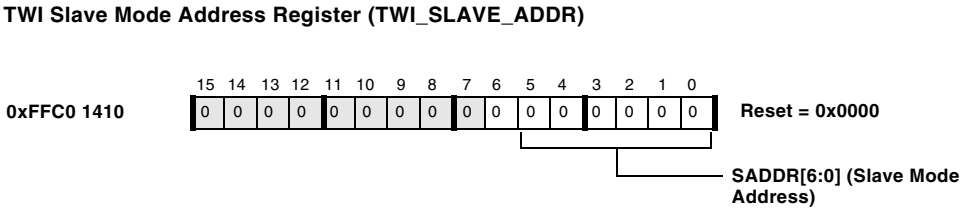


Figure 11-14. TWI Slave Mode Address Register

TWI_SLAVE_STAT Register

During and at the conclusion of slave mode transfers, the TWI slave mode status register (TWI_SLAVE_STAT) holds information on the current transfer. Generally slave mode status bits are not associated with the generation of interrupts. Master mode operation does not affect slave mode status bits.

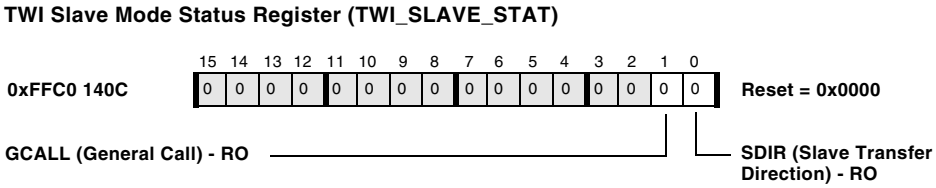


Figure 11-15. TWI Slave Mode Status Register

TWI_MASTER_CTL Register

The TWI master mode control register (TWI_MASTER_CTL) controls the logic associated with master mode operation. Bits in this register do not affect slave mode operation and should not be modified to control slave mode functionality.

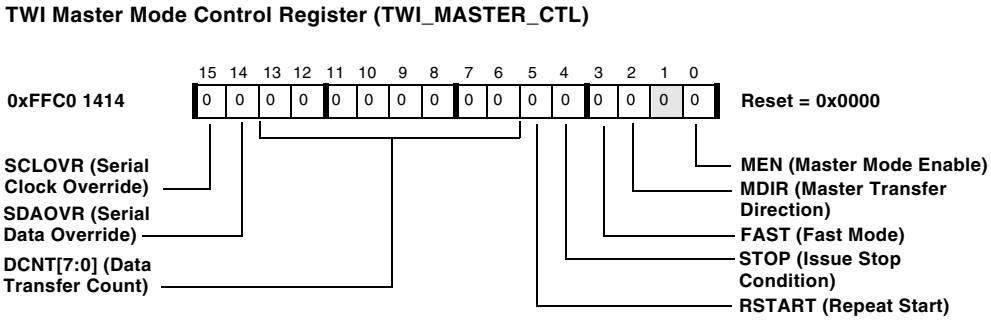


Figure 11-16. TWI Master Mode Control Register

Additional information for the TWI_MASTER_CTL register bits includes:

- **Serial clock override** (SCLOVR)

  This bit can be used when direct control of the serial clock line is required. Normal master and slave mode operation should not require override operation.

  [1] Serial clock output is driven to an active 0 level overriding all other logic. This state is held until this bit is cleared.

  [0] Normal serial clock operation under the control of master mode clock generation and slave mode clock stretching logic.

- **Serial data (SDA) override** (SDAOVR)

  This bit can be used when direct control of the serial data line is required. Normal master and slave mode operation should not require override operation.

  [1] Serial data output is driven to an active 0 level overriding all other logic. This state is held until this bit is cleared.

  [0] Normal serial data operation under the control of the transmit shift register and acknowledge logic.

- **Data transfer count** (DCNT[7:0])

  Indicates the number of data bytes to transfer. As each data word is transferred, DCNT is decremented. When DCNT is 0, a stop condition is generated. Setting DCNT to 0xFF disables the counter. In this transfer mode, data continues to be transferred until it is concluded by setting the STOP bit.

- **Repeat start** (RSTART)

  [1] Issue a repeat start condition at the conclusion of the current transfer (DCNT = 0) and begin the next transfer. The current transfer concludes with updates to the appropriate status and interrupt bits. If errors occurred during the previous transfer, a repeat start does not occur. In the absence of any errors, master enable (MEN) does not self clear on a repeat start.

  [0] Transfer concludes with a stop condition.

- **Issue stop condition** (STOP)

  [1] The transfer concludes as soon as possible avoiding any error conditions (as if data transfer count had been reached) and at that time the TWI interrupt mask register (TWI_INT_MASK) is updated along with any associated status bits.

  [0] Normal transfer operation.

- **Fast mode** (FAST)

  [1] Fast mode (up to 400K bits/s) timing specifications in use.

  [0] Standard mode (up to 100K bits/s) timing specifications in use.

- **Master transfer direction** (MDIR)

  [1] The initiated transfer is master receive.

  [0] The initiated transfer is master transmit.

- **Master mode enable** (MEN)

  This bit self clears at the completion of a transfer. This includes transfers terminated due to errors.

  [1] Master mode functionality is enabled. A start condition is generated if the bus is idle.

  [0] Master mode functionality is disabled. If this bit is cleared during operation, the transfer is aborted and all logic associated with master mode transfers are reset. Serial data and serial clock (SDA, SCL) are no longer driven. Write-1-to-clear status bits are not affected.

TWI_MASTER_ADDR Register

During the addressing phase of a transfer, the TWI controller, with its master enabled, transmits the contents of the TWI master mode address register (`TWI_MASTER_ADDR`). When programming this register, omit the read/write bit. That is, only the upper 7 bits that make up the slave address should be written to this register. For example, if the slave address is `b#1010000X`, where `X` is the read/write bit, then `TWI_MASTER_ADDR` is programmed with `b#1010000`, which corresponds to `0x50`. When sending out the address on the bus, the TWI controller appends the read/write bit as appropriate based on the state of the `MDIR` bit in the master mode control register.

TWI Master Mode Address Register (TWI_MASTER_ADDR)

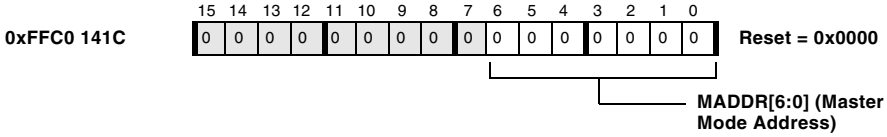


Figure 11-17. TWI Master Mode Address Register

TWI_MASTER_STAT Register

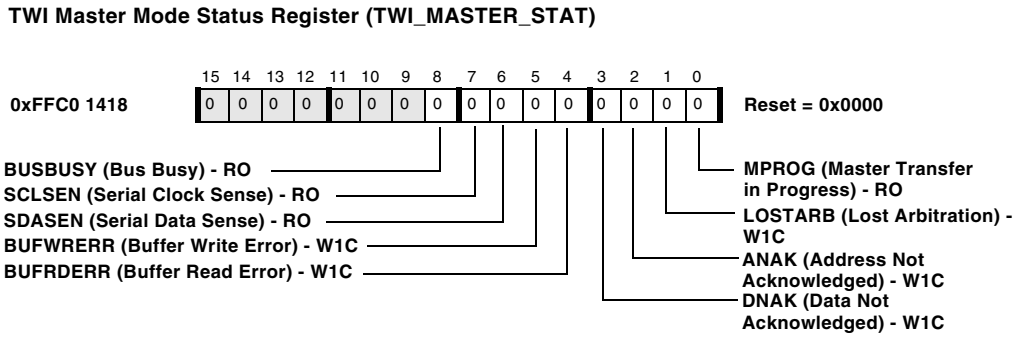


Figure 11-18. TWI Master Mode Status Register

TWI_FIFO_CTL Register

The TWI FIFO control register (`TWI_FIFO_CTL`) control bits affect only the FIFO and are not tied in any way with master or slave mode operation.

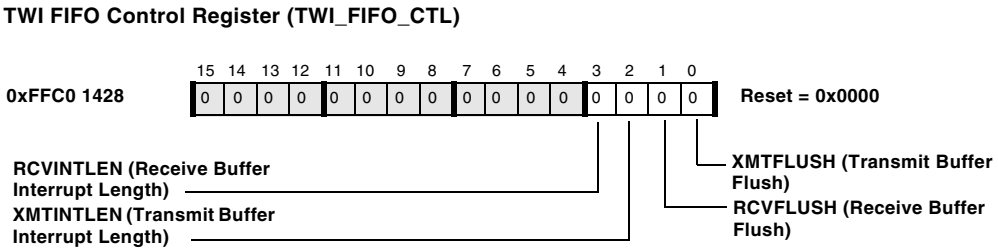


Figure 11-19. TWI FIFO Control Register

Additional information for the TWI_FIFO_CTL register bits includes:

- **Receive buffer interrupt length** (RCVINTLEN)

  This bit determines the rate at which receive buffer interrupts are to be generated. Interrupts may be generated with each byte received or after two bytes are received.

  [1] An interrupt (RCVSERV) is set when the RCVSTAT field in the TWI_FIFO_STAT register indicates two bytes in the FIFO are full (11).

  [0] An interrupt (RCVSERV) is set when RCVSTAT indicates one or two bytes in the FIFO are full (01 or 11).

- **Transmit buffer interrupt length** (XMTINTLEN)

  This bit determines the rate at which transmit buffer interrupts are to be generated. Interrupts may be generated with each byte transmitted or after two bytes are transmitted.

  [1] An interrupt (XMTSERV) is set when the XMTSTAT field in the TWI_FIFO_STAT register indicates two bytes in the FIFO are empty (00).

  [0] An interrupt (XMTSERV) is set when XMTSTAT indicates one or two bytes in the FIFO are empty (01 or 00).

- **Receive buffer flush** (RCVFLUSH)

  [1] Flush the contents of the receive buffer and update the RCVSTAT status bit to indicate the buffer is empty. This state is held until this bit is cleared. During an active receive the receive buffer in this state responds to the receive logic as if it is full.

  [0] Normal operation of the receive buffer and its status bits.

- **Transmit buffer flush** (XMTFLUSH)

  [1] Flush the contents of the transmit buffer and update the XMTSTAT status bit to indicate the buffer is empty. This state is held until this bit is cleared. During an active transmit the transmit buffer in this state responds as if the transmit buffer is empty.

  [0] Normal operation of the transmit buffer and its status bits.

TWI_FIFO_STAT Register

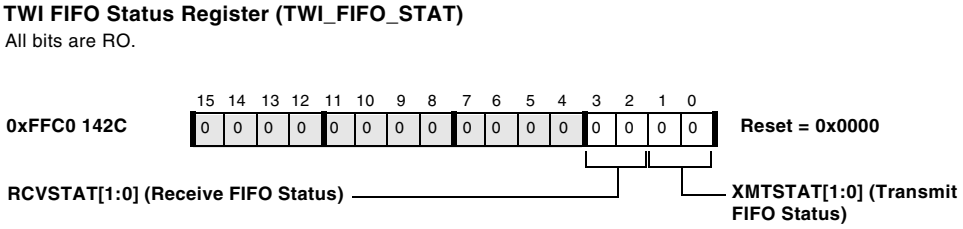


Figure 11-20. TWI FIFO Status Register

TWI_INT_MASK Register

The TWI interrupt mask register (TWI_INT_MASK) enables interrupt sources to assert the interrupt output. Each mask bit corresponds with one interrupt source bit in the TWI interrupt status (TWI_INT_STAT) register. Reading and writing the TWI interrupt mask register does not affect the contents of the TWI interrupt status register.

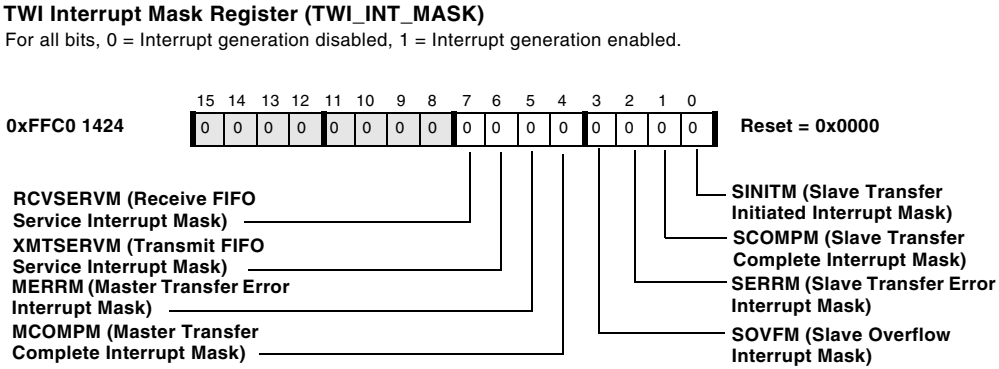


Figure 11-21. TWI Interrupt Mask Register

Additional information for the `TWI_INT_MASK` register bits includes:

- **Receive FIFO service interrupt mask** (`RCVSERVM`)

  [1] The corresponding interrupt source is prevented from asserting the interrupt output.

  [0] A contents of 1 in the corresponding interrupt source results in asserting the interrupt output.

- **Transmit FIFO service interrupt mask** (`XMTSERVM`)

  [1] The corresponding interrupt source is prevented from asserting the interrupt output.

  [0] A contents of 1 in the corresponding interrupt source results in asserting the interrupt output.

- **Master transfer error interrupt mask** (MERRM)

  [1] The corresponding interrupt source is prevented from asserting the interrupt output.

  [0] A contents of 1 in the corresponding interrupt source results in asserting the interrupt output.

- **Master transfer complete interrupt mask** (MCOMPM)

  [1] The corresponding interrupt source is prevented from asserting the interrupt output.

  [0] A contents of 1 in the corresponding interrupt source results in asserting the interrupt output.

- **Slave overflow interrupt mask** (SOVFM)

  [1] The corresponding interrupt source is prevented from asserting the interrupt output.

  [0] A contents of 1 in the corresponding interrupt source results in asserting the interrupt output.

- **Slave transfer error interrupt mask** (SERRM)

  [1] The corresponding interrupt source is prevented from asserting the interrupt output.

  [0] A contents of 1 in the corresponding interrupt source results in asserting the interrupt output.

- **Slave transfer complete interrupt mask** (SCOMPM)

  [1] The corresponding interrupt source is prevented from asserting the interrupt output.

  [0] A contents of 1 in the corresponding interrupt source results in asserting the interrupt output.

- **Slave transfer initiated interrupt mask** (SINITM)

  [1] The corresponding interrupt source is prevented from asserting the interrupt output.

  [0] A contents of 1 in the corresponding interrupt source results in asserting the interrupt output.

TWI_INT_STAT Register

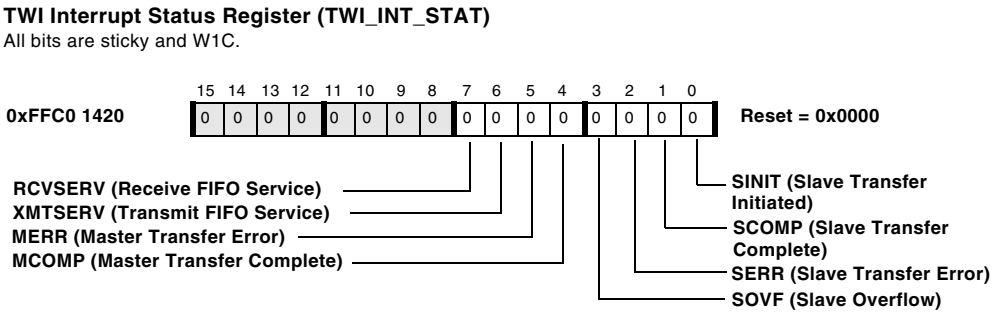


Figure 11-22. TWI Interrupt Status Register

TWI_XMT_DATA8 Register

The TWI FIFO transmit data single byte register (TWI_XMT_DATA8) holds an 8-bit data value written into the FIFO buffer. Transmit data is entered into the corresponding transmit buffer in a first-in first-out order.

Although peripheral bus writes are 16 bits, a write access to TWI_XMT_DATA8 adds only one transmit data byte to the FIFO buffer. With each access, the transmit status (XMTSTAT) field in the TWI_FIFO_STAT register is updated. If an access is performed while the FIFO buffer is full, the write is ignored and the existing FIFO buffer data and its status remains unchanged.

TWI FIFO Transmit Data Single Byte Register (TWI_XMT_DATA8)
All bits are WO.

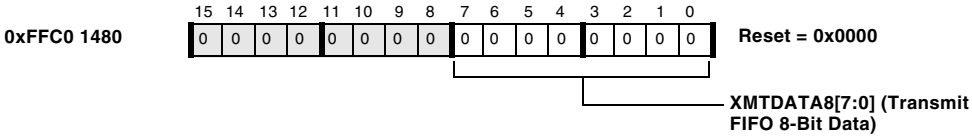


Figure 11-23. TWI FIFO Transmit Data Single Byte Register

TWI_XMT_DATA16 Register

The TWI FIFO transmit data double byte register (TWI_XMT_DATA16) holds a 16-bit data value written into the FIFO buffer. To reduce interrupt output rates and peripheral bus access times, a double byte transfer data access can be performed. Two data bytes can be written, effectively filling the transmit FIFO buffer with a single access. The data is written in little endian byte order as shown in Figure 11-24 where byte 0 is the first byte to be transferred and byte 1 is the second byte to be transferred. With each access, the transmit status (XMTSTAT) field in the TWI_FIFO_STAT

register is updated. If an access is performed while the FIFO buffer is not empty, the write is ignored and the existing FIFO buffer data and its status remains unchanged.

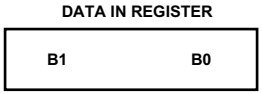


Figure 11-24. Little Endian Byte Order

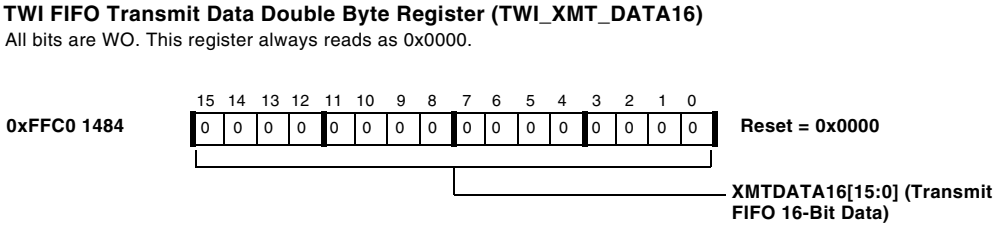


Figure 11-25. TWI FIFO Transmit Data Double Byte Register

TWI_RCV_DATA8 Register

The TWI FIFO receive data single byte register (`TWI_RCV_DATA8`) holds an 8-bit data value read from the FIFO buffer. Receive data is read from the corresponding receive buffer in a first-in first-out order. Although peripheral bus reads are 16 bits, a read access to `TWI_RCV_DATA8` will access only one transmit data byte from the FIFO buffer. With each access, the

receive status (RCVSTAT) field in the TWI_FIFO_STAT register is updated. If an access is performed while the FIFO buffer is empty, the data is unknown and the FIFO buffer status remains indicating it is empty.

TWI FIFO Receive Data Single Byte Register (TWI_RCV_DATA8)
All bits are RO.

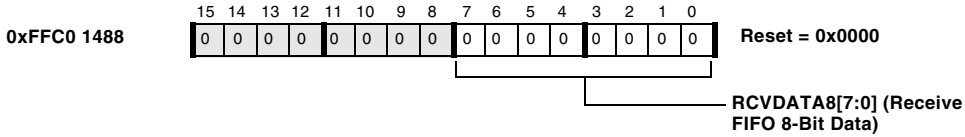


Figure 11-26. TWI FIFO Receive Data Single Byte Register

TWI_RCV_DATA16 Register

The TWI FIFO receive data double byte register (TWI_RCV_DATA16) holds a 16-bit data value read from the FIFO buffer. To reduce interrupt output rates and peripheral bus access times, a double byte receive data access can be performed. Two data bytes can be read, effectively emptying the receive FIFO buffer with a single access. The data is read in little endian byte order as shown in Figure 11-27 where byte 0 is the first byte received and byte 1 is the second byte received. With each access, the receive status (RCVSTAT) field in the TWI_FIFO_STAT register is updated to indicate it is empty. If an access is performed while the FIFO buffer is not full, the read data is unknown and the existing FIFO buffer data and its status remains unchanged.

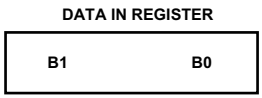


Figure 11-27. Little Endian Byte Order

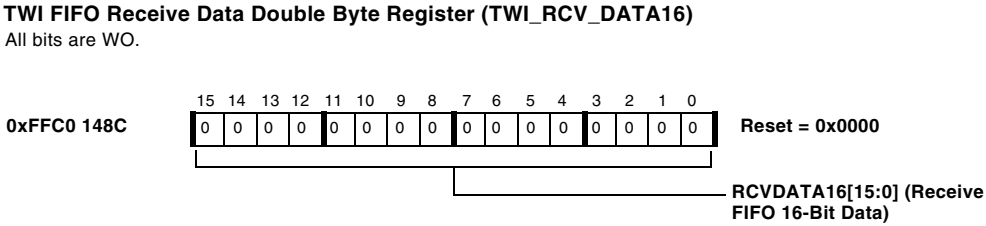


Figure 11-28. TWI FIFO Receive Data Double Byte Register

Programming Examples

The following sections include programming examples for general setup, slave mode, and master mode, as well as guidance for repeated start conditions.

Master Mode Setup

Listing 11-1 shows how to initiate polled receive and transmit transfers in master mode.

Listing 11-1. Master Mode Receive/Transmit Transfer

```
/*********************************************************
  macro for the Count field of the TWI_MASTER_CTL register
  x can be any value between 0 and 0xFE (254).  a value of
  0xFF disables the counter
*********************************************************/
#define TWICount(x) (DCNT & ((x) << 6))

.section L1_data_b;
.byte TX_file[file_size] = "DATA.hex";
.BYTE RX_CHECK[file_size];
```

```
.byte rcvFirstWord[2];

.SECTION program;
_main:
/*********************************************************
TWI Master Initialization subroutine
*********************************************************/

TWI_INIT:
/*********************************************************
Enable the TWI controller and set the Prescale value
Prescale = 10 (0xA) for an SCLK = 100 MHz (CLKIN = 50MHz)
Prescale = SCLK / 10 MHz

P1 points to the base of the system MMRs
*********************************************************/

R1 = TWI_ENA | 0xA (z);
W[P1 + LO(TWI_CONTROL)] = R1;

/*********************************************************
Set CLKDIV:
For example, for an SCL of 400 KHz (period = 1/400 KHz = 2500 ns)
and an internal time reference of 10 MHz (period = 100 ns):
CLKDIV = 2500 ns / 100 ns = 25
For an SCL with a 30% duty cycle, then CLKLOW = 17 (0x11) and
CLKHI = 8.
*********************************************************/
R5 = CLKHI(0x8) | CLKLOW(0x11) (z);
W[P1 + LO(TWI_CLKDIV)] = R5;

/*********************************************************
enable these signals to generate a TWI interrupt: optional
*********************************************************/
```

```
R1 = RCVSERV | XMTSERV | MERR | MCOMP (z);
W[P1 + LO(TWI_INT_MASK)] = R1;

/*********************************************************
The address needs to be shifted one place to the right
e.g., 1010 001x becomes 0101 0001 (0x51) the TWI controller
will actually send out 1010 001x where x is either a 0 for
writes or 1 for reads
*********************************************************/
R6 = 0xBF;
R6 = R6 >> 1;
TWI_INIT.END: W[P1 + LO(TWI_MASTER_ADDR)] = R6;

/******************* END OF TWI INIT *********************/

/*********************************************************
Starting the Read transfer
Program the Master Control register with:
1. the number of bytes to transfer: TWICount(x)
2. Repeated Start (RESTART): optional
3. speed mode: FAST or SLOW
4. direction of transfer:
      MDIR = 1 for reads, MDIR = 0 for writes
5. Master Enable MEN. This will kick off the master transfer
*********************************************************/
R1 = TWICount(0x2) | FAST | MDIR | MEN;
W[P1 + LO(TWI_MASTER_CTL)] = R1;
ssync;

/*********************************************************
Poll the FIFO Status register to know when
2 bytes have been shifted into the RX FIFO
*********************************************************/
Rx_stat:
```

```
R1 = W[P1 + LO(TWI_FIFO_STAT)](Z);
R0 = 0xC;
R1 = R1 & R0;
CC = R1 == R0;
IF !cc jump Rx_stat;
R0 = W[P1 + LO(TWI_RCV_DATA16)](Z);  /* Read data from the RX
fifo */
ssync;

/*********************************************************
check that master transfer has completed
MCOMP will be set when Count reaches zero
*********************************************************/
M_COMP:
  R1 = W[P1 + LO(TWI_INT_STAT)](z);
  CC = BITTST (R1, bitpos(MCOMP));
  if !CC jump M_COMP;
M_COMP.END:  W[P1 + LO(TWI_INT_STAT)] = R1;

/* load the pointer with the address of the transmit buffer */
P2.H = TX_file;
P2.L = TX_file;

/*********************************************************
Pre-load the tx FIFO with the first two bytes: this is
necessary to avoid the generation of the Buffer Read Error
(BUFRDERR) which occurs whenever a transmit transfer is
initiated while the transmit buffer is empty
*********************************************************/
R3 = W[P2++](Z);
W[P1 + LO(TWI_XMT_DATA16)] = R3;

/*********************************************************
Initiating the Write operation
```

```
Program the Master Control register with:
1. the number of bytes to transfer: TWICount(x)
2. Repeated Start (RESTART): optional
3. speed mode: FAST or Standard
4. direction of transfer:
      MDIR = 1 for reads, MDIR = 0 for writes
5. Master Enable MEN. Setting this bit will kick off the transfer
************************************************************/
R1 = TWICount(0xFE) | FAST | MEN;
W[P1 + LO(TWI_MASTER_CTL)] = R1;
SSYNC;

/************************************************************
  loop to write data to a TWI slave device P3 times
************************************************************/
P3 = length(TX_file);

LSETUP (Loop_Start, Loop_End) LC0 = P3;
  Loop_Start:
    /*******************************************************
    check that there's at least one byte location empty in
    the tx fifo
    *******************************************************/
    XMTSERV_Status:
    R1 = W[P1 + LO(TWI_INT_STAT)](z);
    CC = BITTST (R1, bitpos(XMTSERV)); /* test XMTSERV bit */
    if !CC jump XMTSERV_Status;
    W[P1 + LO(TWI_INT_STAT)] = R1; /* clear status */
    SSYNC;

    /*******************************************************
    write byte into the transmit FIFO
    *******************************************************/
    R3 = B[P2++](Z);
```

```
    W[P1 + LO(TWI_XMT_DATA8)] = R3;
  Loop_End:  SSYNC;

/* check that master transfer has completed */
M_COMP:
R1 = W[P1 + LO(TWI_INT_STAT)](z);
CC = BITTST (R1, bitpos(MCOMP));
if !CC jump M_COMP;
M_COMP.END:W[P1 + LO(TWI_INT_STAT)] = R1;

idle;
_main.end:
```

Slave Mode Setup

Listing 11-2 shows how to configure the slave for interrupt based transfers. The interrupts are serviced in the subroutine _TWI_ISR shown in Listing 11-3.

Listing 11-2. Slave Mode Setup

```
#include <defBF537.h>
#include "startup.h"

#define file_size 254
#define SYSMMR_BASE 0xFFC00000
#define COREMMR_BASE 0xFFE00000

.GLOBAL _main;
.EXTERN _TWI_ISR;

.section L1_data_b;
.BYTE TWI_RX[file_size];
```

```
.BYTE TWI_TX[file_size] = "transmit.dat";

.section L1_code;
_main:

/*********************************************************
TWI Slave Initialization subroutine
*********************************************************/
TWI_SLAVE_INIT:

/*********************************************************
Enable the TWI controller and set the Prescale value
Prescale = 10 (0xA) for an SCLK = 100 MHz (CLKIN = 50MHz)
Prescale = SCLK / 10 MHz
P1 points to the base of the system MMRs
P0 points to the base of the core MMRs
*********************************************************/
R1 = TWI_ENA | 0xA (z);
W[P1 + LO(TWI_CONTROL)] = R1;

/*********************************************************
Slave address
program the address to which this slave will respond to.
this is an arbitrary 7-bit value
*********************************************************/
R1 = 0x5F;
W[P1 + LO(TWI_SLAVE_ADDR)] = R1;

/*********************************************************
Pre-load the TX FIFO with the first two bytes to be
transmitted in the event the slave is addressed and a transmit
is required
*********************************************************/
R3=0xB537(Z);
```

```
W[P1 + LO(TWI_XMT_DATA16)] = R3;

/*********************************************************
FIFO Control determines whether an interrupt is generated
for every byte transferred or for every two bytes.
A value of zero which is the default, allows for single byte
events to generate interrupts
*********************************************************/
R1 = 0;
 W[P1 + LO(TWI_FIFO_CTL)] = R1;

/*********************************************************
enable these signals to generate a TWI interrupt
*********************************************************/
R1 = RCVSERV | XMTSERV | SOVF | SERR | SCOMP | SINIT (z);
W[P1 + LO(TWI_INT_MASK)] = R1;

/*********************************************************
Enable the TWI Slave
Program the Slave Control register with:
1. Slave transmit data valid (STDVAL) set so that the contents of
the TX FIFO can be used by this slave when a master requests data
from it.
2. Slave Enable SEN to enable Slave functionality
*********************************************************/
R1 = STDVAL | SEN;
W[P1 + LO(TWI_SLAVE_CTL)] = R1;
TWI_SLAVE_INIT.END:

P2.H = HI(TWI_RX);
P2.L = LO(TWI_RX);

P4.H = HI(TWI_TX);
P4.L = LO(TWI_TX);
```

```
/*********************************************************
Remap the vector table pointer from the default __I10HANDLER
to the new _TWI_ISR interrupt service routine
*********************************************************/
R1.H = HI(_TWI_ISR);
R1.L = LO(_TWI_ISR);
[P0 + LO(EVT10)] = R1; /* note that P0 points to the base of the
core MMR registers */

/*********************************************************
ENABLE TWI generate to interrupts at the system level
*********************************************************/
R1 = [P1 + LO(SIC_IMASK)];
BITSET(R1,BITPOS(IRQ_TWI));
[P1 + LO(SIC_IMASK)] = R1;

/*********************************************************
ENABLE TWI to generate interrupts at the core level
*********************************************************/
R1 = [P0 + LO(IMASK)];
BITSET(R1,BITPOS(EVT_IVG10));
[P0 + LO(IMASK)] = R1;

/*********************************************************
 wait for interrupts
*********************************************************/
idle;

_main.END:
```

Listing 11-3. TWI Slave Interrupt Service Routine

```
/*************************************************************
 Function:  _TWI_ISR
 Description:  This ISR is executed when the TWI controller
detects a slave initiated transfer. After an interrupt is ser-
viced, its corresponding bit is cleared in the TWI_INT_STAT
register. This done by writing a 1 to the particular bit posi-
tion. All bits are write 1 to clear.
*************************************************************/
#include <defBF537.h>

.GLOBAL _TWI_ISR;

.section L1_code;
_TWI_ISR:

/*************************************************************
read the source of the interrupt
*************************************************************/
R1 = W[P1 + LO(TWI_INT_STAT)](z);

/*************************************************************
Slave Transfer Initiated
*************************************************************/
CC = BITTST(R1, BITPOS(SINIT));
if !CC JUMP RECEIVE;
R0 = SINIT (Z);
W[P1 + LO(TWI_INT_STAT)] = R0;  /* clear interrupt source bit */
ssync;

/*************************************************************
Receive service
*************************************************************/
```

```
RECEIVE:
CC = BITTST(R1, BITPOS(RCVSERV));
if !CC JUMP TRANSMIT;
R0 = W[P1 + LO(TWI_RCV_DATA8)] (Z); /* read data */
B[P2++] = R0 ; /* store bytes into a buffer pointed to by P2 */
R0 = RCVSERV(Z);
W[P1 + LO(TWI_INT_STAT)] = R0; /*clear interrupt source bit */
ssync;
JUMP _TWI_ISR.END; /* exit */

/*************************************************************
Transmit service
*************************************************************/
TRANSMIT:
CC = BITTST(R1, BITPOS(XMTSERV));
if !CC JUMP SlaveError;
R0 = B[P4++](Z);
W[P1 + LO(TWI_XMT_DATA8)] = R0;
R0 = XMTSERV(Z);
W[P1 + LO(TWI_INT_STAT)] = R0; /* clear interrupt source bit */
ssync;
JUMP _TWI_ISR.END; /* exit */

/*************************************************************
slave transfer error
*************************************************************/
SlaveError:
CC = BITTST(R1, BITPOS(SERR));
if !CC SlaveOverflow;
R0 = SERR(Z);
W[P1 + LO(TWI_INT_STAT)] = R0; /* clear interrupt source bit */
ssync;
JUMP _TWI_ISR.END; /* exit */
```

```
/*********************************************************
slave overflow
*********************************************************/
SlaveOverflow:
CC = BITTST(R1, BITPOS(SOVF));
if !CC JUMP SlaveTransferComplete;
R0 = SOVF(Z);
W[P1 + LO(TWI_INT_STAT)] = R0; /* clear interrupt source bit */
ssync;
JUMP _TWI_ISR.END; /* exit */

/*********************************************************
 slave transfer complete
*********************************************************/
SlaveTransferComplete:
CC = BITTST(R1, BITPOS(SCOMP));
if !CC JUMP _TWI_ISR.END;
R0 = SCOMP(Z);
W[P1 + LO(TWI_INT_STAT)] = R0; /* clear interrupt source bit */
ssync;
/* Transfer complete read receive FIFO buffer and set/clear sema-
phores etc... */
R0 = W[P1 + LO(TWI_FIFO_STAT)](z);
CC = BITTST(R0,BITPOS(RCV_HALF));  /* BIT 2 indicates whether
there's a byte in the FIFO or not */
if !CC JUMP _TWI_ISR.END;
R0 = W[P1 + LO(TWI_RCV_DATA8)] (Z); /* read data */
B[P2++] = R0 ; /* store bytes into a buffer pointed to by P2 */

_TWI_ISR.END:RTI;
```

Electrical Specifications

All logic complies with the Electrical Specification outlined in the *Philips I^2C Bus Specification version 2.1* dated January 2000.

12 SPORT CONTROLLERS

This chapter describes the synchronous Serial peripheral Port (SPORT). Following an overview and a list of key features is a description of operation and functional modes of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

Overview

The ADSP-BF534, ADSP-BF536, and ADSP-BF537 processors feature two identical synchronous serial ports, called SPORTs. Unlike the SPI interface which has been designed for SPI-compatible communication only, the SPORT modules support a variety of serial data communication protocols, for example:

- A-law or µ-law companding according to G.711 specification
- Multichannel or Time-Division-Multiplexed (TDM) modes
- Stereo Audio I2S Mode
- H.100 Telephony standard support

In addition to these standard protocols, the SPORT modules provide straight-forward modes to connect to standard peripheral devices, such as ADCs or codecs, without external glue logic. With support for high data rates, independent transmit and receive channels, and dual data paths, the SPORT interface is a perfect choice for direct serial interconnection

between two or more processors in a multiprocessor system. Many processors provide compatible interfaces, including DSPs from Analog Devices and other manufacturers.

Both SPORTs have the same capabilities and are programmed in the same way. Each SPORT has its own set of control registers and data buffers.

Features

The SPORTs can operate at up to 1/2 the system clock (SCLK) rate for an internally generated or external serial clock. The SPORT external clock must always be less than the SCLK frequency. Independent transmit and receive clocks provide greater flexibility for serial communications.

Each of the SPORTs offers these features and capabilities:

- Provides independent transmit and receive functions.
- Transfers serial data words from 3 to 32 bits in length, either MSB first or LSB first.
- Provides alternate framing and control for interfacing to I^2S serial devices, as well as other audio formats (for example, left-justified stereo serial data).
- Has FIFO plus double buffered data (both receive and transmit functions have a data buffer register and a shift register), providing additional time to service the SPORT.
- Provides two synchronous transmit and two synchronous receive data signals and buffers in each SPORT to double the total supported datastreams.
- Performs A-law and μ-law hardware companding on transmitted and received words. (See "Companding" on page 12-29 for more information.)

- Internally generates serial clock and frame sync signals in a wide range of frequencies or accepts clock and frame sync input from an external source.
- Operates with or without frame synchronization signals for each data word, with internally generated or externally generated frame signals, with active high or active low frame signals, and with either of two configurable pulse widths and frame signal timing.
- Performs interrupt-driven, single word transfers to and from on-chip memory under processor control.
- Provides direct memory access transfer to and from memory under DMA master control. DMA can be autobuffer-based (a repeated, identical range of transfers) or descriptor-based (individual or repeated ranges of transfers with differing DMA parameters).
- Has a multichannel mode for TDM interfaces. Each SPORT can receive and transmit data selectively from a time-division-multiplexed serial bitstream on 128 contiguous channels from a stream of up to 1024 total channels. This mode can be useful as a network communication scheme for multiple processors. The 128 channels available to the processor can be selected to start at any channel location from 0 to 895 = (1023 – 128). Note the multichannel select registers and the `WSIZE` register control which subset of the 128 channels within the active region can be accessed.

Interface Overview

SPORT0 and SPORT1 provide an I/O interface to a wide variety of peripheral serial devices. SPORT0 is accessible via port J and SPORT1 is accessible via port G. For more information on the port configuration, see Chapter 14, “General-Purpose Ports”. SPORTs provide synchronous serial data transfer only. Each SPORT has one group of signals (primary data, secondary data, clock, and frame sync) for transmit and a second set

of signals for receive. The receive and transmit functions are programmed separately. Each SPORT is a full duplex device, capable of simultaneous data transfer in both directions. The SPORTs can be programmed for bit rate, frame sync, and number of bits per word by writing to memory-mapped registers.

In this text, the naming conventions for registers and signals use a lower case x to represent a digit. In this chapter, for example, the name RFSx signals indicates RFS0 and RFS1 (corresponding to SPORT0 and SPORT1, respectively). In this chapter, LSB refers to least significant bit, and MSB refers to most significant bit.

Port J contains the SPORT0 pins. Some of the SPORT0 pins are multiplexed and can be used for other purposes if the entire SPORT0 block or some of its signals are not required by an application. However, all pins default to the SPORT0 module settings after reset.

The secondary data pins of SPORT0 are multiplexed with the CAN interface. Unless the PJCE bit in the PORT_MUX register is set, the CAN signals are disabled and the secondary data signals control the associated pins, by default. Similarly, the PJSE bit can disconnect some of the transmit signals and redirect the respective pins to the SPI module. The remaining SPORT0 signals aren't multiplexed. Nevertheless, they can be sensed by the timer module as alternative clock modules, regardless of whether SPORT0 is enabled or not. See Figure 14-4 on page 14-8 for details.

SPORT1 resides in port G. Its signals are mainly shared with upper PPI data lines. By default, all port G pins are configured in GPIO mode. Writing a 1 to bits 8–15 of the PORTG_FER register enables either PPI or SPORT1 signals on the respective pins. If the PGSE, PGRE, and PGTE bits in the PORT_MUX register are also set, the full SPORT1 functionality is enabled, while the PPI can still operate in 8-bit mode. See Figure 14-2 on page 14-6 for details.

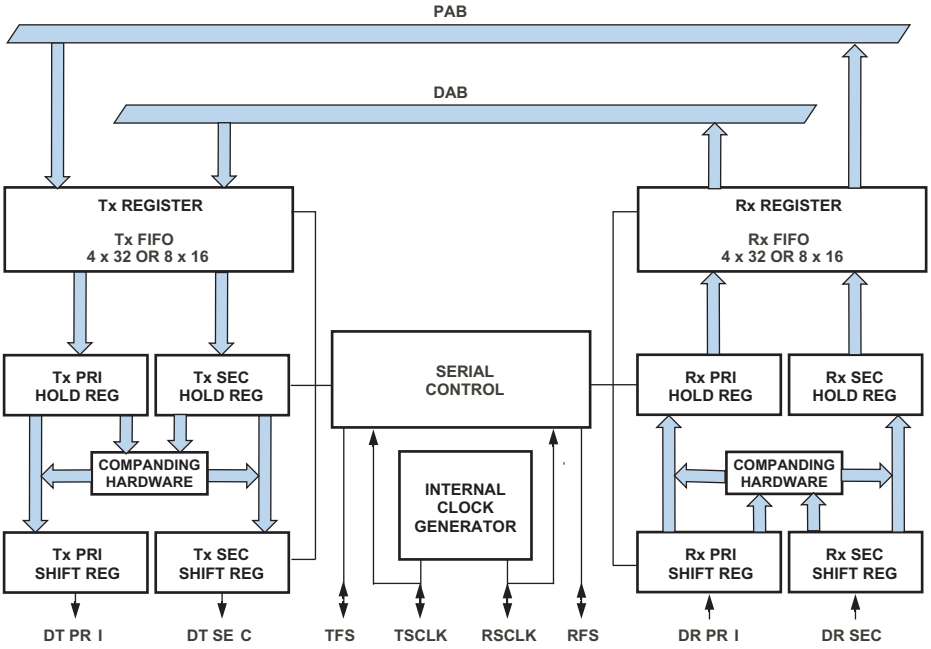
If the secondary data signals of SPORT1 are not used, 10-bit PPI operation is enabled by keeping the PGSE bit cleared. The SPORT1 transmit channel remains fully functional, even when the PPI operates up to 13-bit mode and the PGRE bit is cleared. Regardless of the multiplexing scheme used, any pin that is not required by SPORT1 or by the PPI in a specific application can function as GPIO by keeping that bit in the PORTG_FER register cleared.

Figure 12-1 shows a simplified block diagram of a single SPORT. Data to be transmitted is written from an internal processor register to the SPORT's SPORTx_TX register via the peripheral bus. This data is optionally compressed by the hardware and automatically transferred to the TX shift register. The bits in the shift register are shifted out on the SPORT's DTxPRI/DTxSEC pin, MSB first or LSB first, synchronous to the serial clock on the TSCLKx pin. The receive portion of the SPORT accepts data from the DRxPRI/DRxSEC pin synchronous to the serial clock on the RSCLKx pin. When an entire word is received, the data is optionally expanded, then automatically transferred to the SPORT's SPORTx_RX register, and then into the RX FIFO where it is available to the processor. Table 12-1 shows the signals for each SPORT.

Table 12-1. SPORTx Signals

Pin[1]	Description
DTxPRI	Transmit Data Primary
DTxSEC	Transmit Data Secondary
TSCLKx	Transmit Clock
TFSx	Transmit Frame Sync
DRxPRI	Receive Data Primary
DRxSEC	Receive Data Secondary
RSCLKx	Receive Clock
RFSx	Receive Frame Sync

1 A lowercase x within a signal name represents a possible value of 0 or 1 (corresponding to SPORT0 or SPORT1).



NOTE 1: ALL WIDE ARROW DATA PATHS ARE 16 OR 32 BITS WIDE, DEPENDING ON SLEN. FOR SLEN = 2 TO 15, A 16-BIT DATA PATH WITH 8-DEEP FIFO IS USED. FOR SLEN = 16 TO 31, A 32-BIT DATA PATH WITH 4-DEEP FIFO IS USED.
NOTE 2: Tx REGISTER IS THE BOTTOM OF THE Tx FIFO, Rx REGISTER IS THE TOP OF THE Rx FIFO.

Figure 12-1. SPORT Block Diagram

A SPORT receives serial data on its DRxPRI and DRxSEC inputs and transmits serial data on its DTxPRI and DTxSEC outputs. It can receive and transmit simultaneously for full-duplex operation. For transmit, the data bits (DTxPRI and DTxSEC) are synchronous to the transmit clock (TSCLKx). For receive, the data bits (DRxPRI and DRxSEC) are synchronous to the receive clock (RSCLKx). The serial clock is an output if the processor generates it, or an input if the clock is externally generated. Frame synchronization signals RFSx and TFSx are used to indicate the start of a serial data word or stream of serial words.

The primary and secondary data pins, if enabled by the port configuration, provide a method to increase the data throughput of the serial port. They do not behave as totally separate SPORTs; rather, they operate in a synchronous manner (sharing clock and frame sync) but on separate data. The data received on the primary and secondary signals is interleaved in main memory and can be retrieved by setting a stride in the Data Address Generators (DAG) unit. For more information about DAGs, see the "Data Address Generators" chapter in the *Blackfin Processor Programming Reference*. Similarly, for TX, data should be written to the TX register in an alternating manner—first primary, then secondary, then primary, then secondary, and so on. This is easily accomplished with the processor's powerful DAGs.

In addition to the serial clock signal, data must be signalled by a frame synchronization signal. The framing signal can occur either at the beginning of an individual word or at the beginning of a block of words.

Figure 12-2 shows a possible port connection for the SPORTs. Note serial devices A and B must be synchronous, as they share common frame syncs and clocks. The same is true for serial devices C and D.

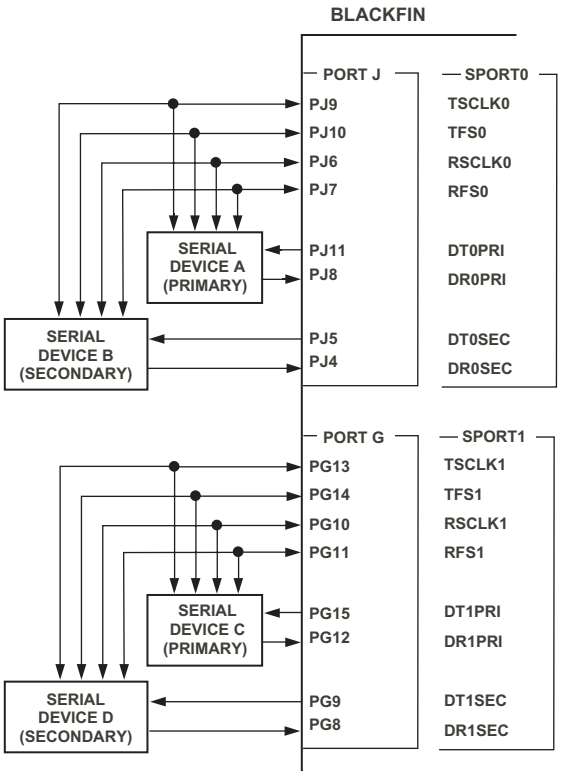


Figure 12-2. SPORT Connections

Figure 12-3 shows an example of a stereo serial device with three transmit and two receive channels connected to the processor.

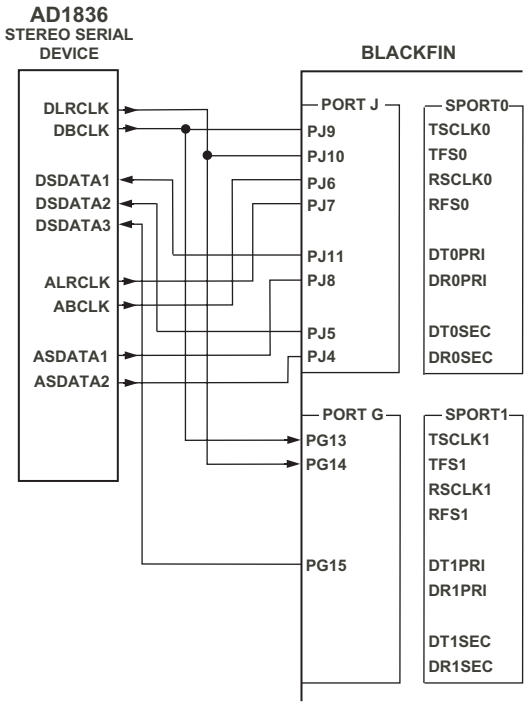


Figure 12-3. Stereo Serial Connection

SPORT Pin/Line Terminations

The processor has very fast drivers on all output pins, including the SPORTs. If connections on the data, clock, or frame sync lines are longer than six inches, consider using a series termination for strip lines on point-to-point connections. This may be necessary even when using low speed serial clocks, because of the edge rates.

Description of Operation

SPORT Operation

This section describes general SPORT operation, illustrating the most common use of a SPORT. Since the SPORT functionality is configurable, this description represents just one of many possible configurations.

Writing to a SPORT's SPORTx_TX register readies the SPORT for transmission. The TFS signal initiates the transmission of serial data. Once transmission has begun, each value written to the SPORTx_TX register is transferred through the FIFO to the internal transmit shift register. The bits are then sent, beginning with either the MSB or the LSB as specified in the SPORTx_TCR1 register. Each bit is shifted out on the driving edge of TSCLKx. The driving edge of TSCLKx can be configured to be rising or falling. The SPORT generates the transmit interrupt or requests a DMA transfer as long as there is space in the TX FIFO.

As a SPORT receives bits, they accumulate in an internal receive register. When a complete word has been received, it is written to the SPORT FIFO register and the receive interrupt for that SPORT is generated or a DMA transfer is initiated. Interrupts are generated differently if DMA block transfers are performed. For information about DMA, see Chapter 5, "Direct Memory Access".

SPORT Disable

The SPORTs are automatically disabled by a processor hardware or software reset. A SPORT can also be disabled directly by clearing the SPORT's transmit or receive enable bits (TSPEN in the SPORTx_TCR1 register and RSPEN in the SPORTx_RCR1 register, respectively). Each method has a different effect on the SPORT.

A processor reset disables the SPORTs by clearing the SPORTx_TCR1, SPORTx_TCR2, SPORTx_RCR1, and SPORTx_RCR2 registers (including the TSPEN and RSPEN enable bits) and the SPORTx_TCLKDIV, SPORTx_RCLKDIV, SPORTx_TFSDIVx, and SPORTx_RFSDIVx clock and frame sync divisor registers. Any ongoing operations are aborted.

Clearing the TSPEN and RSPEN enable bits disables the SPORTs and aborts any ongoing operations. Status bits are also cleared. Configuration bits remain unaffected and can be read by the software in order to be altered or overwritten. To disable the SPORT output clock, set the SPORT to be disabled.

Note that disabling a SPORT via TSPEN/RSPEN may shorten any currently active pulses on the TFSx/RFSx and TSCLKx/RSCLKx outputs, if these signals are configured to be generated internally.

The SPORTs are ready to start transmitting or receiving data no later than three serial clock cycles after they are enabled in the SPORTx_TCR1 or SPORTx_RCR1 register. No serial clock cycles are lost from this point on. The first internal frame sync will occur one frame sync delay after the SPORTs are ready. External frame syncs can occur as soon as the SPORT is ready.

When disabling the SPORT from multichannel operation, first disable TXEN and then disable RXEN. Note both TXEN and RXEN must be disabled before reenabling. Disabling only TX or RX is not allowed.

Setting SPORT Modes

SPORT configuration is accomplished by setting bit and field values in configuration registers. Each SPORT must be configured prior to being enabled. Once the SPORT is enabled, further writes to the SPORT configuration registers are disabled (except for SPORTx_RCLKDIV, SPORTx_TCLKDIV, and multichannel mode channel select registers). To change values in all other SPORT configuration registers, disable the SPORT by clearing TSPEN in SPORTx_TCR1 and/or RSPEN in SPORTx_RCR1.

Each SPORT has its own set of control registers and data buffers. These registers are described in detail in the "SPORT Registers" section. All control and status bits in the SPORT registers are active high unless otherwise noted.

Stereo Serial Operation

Several stereo serial modes can be supported by the SPORT, including the popular I^2S format. To use these modes, set bits in the SPORT_RCR2 or SPORT_TCR2 registers. Setting RSFSE or TSFSE in SPORT_RCR2 or SPORT_TCR2 changes the operation of the frame sync pin to a left/right clock as required for I^2S and left-justified stereo serial data. Setting this bit enables the SPORT to generate or accept the special LRCLK-style frame sync. All other SPORT control bits remain in effect and should be set appropriately. Figure 12-4 and Figure 12-5 show timing diagrams for stereo serial mode operation.

Table 12-2 shows several modes that can be configured using bits in SPORTx_TCR1 and SPORTx_RCR1. The table shows bits for the receive side of the SPORT, but corresponding bits are available for configuring the transmit portion of the SPORT. A control field which may be either set or cleared depending on the user's needs, without changing the standard, is indicated by an "X."

Table 12-2. Stereo Serial Settings

Bit Field	Stereo Audio Serial Scheme		
	I^2S	Left-Justified	DSP Mode
RSFSE	1	1	0
RRFST	0	0	0
LARFS	0	1	0
LRFS	0	1	0
RFSR	1	1	1

Table 12-2. Stereo Serial Settings (Cont'd)

Bit Field	Stereo Audio Serial Scheme		
	I^2S	Left-Justified	DSP Mode
RCKFE	1	0	0
SLEN	2 – 31	2 – 31	2 – 31
RLSBIT	0	0	0
RFSDIV (If internal FS is selected.)	2 – Max	2 – Max	2 – Max
RXSE (Secondary Enable is available for RX and TX.)	X	X	X

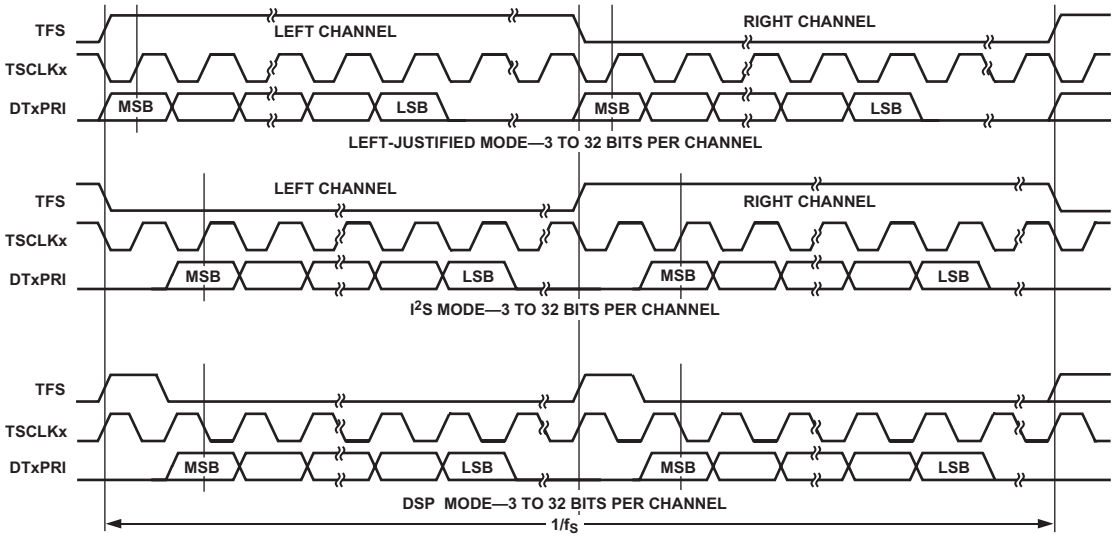
Note most bits shown as a 0 or 1 may be changed depending on the user's preference, creating many other "almost standard" modes of stereo serial operation. These modes may be of use in interfacing to codecs with slightly non-standard interfaces. The settings shown in Table 12-2 provide glueless interfaces to many popular codecs.

Note `RFSDIV` or `TFSDIV` must still be greater than or equal to `SLEN`. For I^2S operation, `RFSDIV` or `TFSDIV` is usually 1/64 of the serial clock rate. With `RSFSE` set, the formulas to calculate frame sync period and frequency (discussed in "Clock and Frame Sync Frequencies" on page 12-26) still apply, but now refer to one half the period and twice the frequency. For instance, setting `RFSDIV` or `TFSDIV = 31` produces an `LRCLK` that transitions every 32 serial clock cycles and has a period of 64 serial clock cycles.

The `LRFS` bit determines the polarity of the frame sync pin that is considered a "right" channel. Thus, setting `LRFS = 0` indicates that a low signal on the `RFS` or `TFS` pin is the left channel. This is the default setting.

The `RRFST` and `TRFST` bits determine whether the first word received or transmitted is a left or a right channel. If the bit is set, the first word received or transmitted is a right channel. The default is to receive or transmit the left channel word first.

The secondary DRxSEC and DTxSEC pins are useful extensions of the SPORT which pair well with stereo serial mode. Multiple I^2S streams of data can be transmitted or received using a single SPORT. Note the primary and secondary pins are synchronous, as they share clock and LRCLK (frame sync) pins. The transmit and receive sides of the SPORT need not be synchronous, but may share a single clock in some designs. See Figure 12-3 on page 12-9, which shows multiple stereo serial connections being made between the processor and an AD1836 codec.



NOTES:
1. DSP MODE DOES NOT IDENTIFY CHANNEL.
2. TFS NORMALLY OPERATES AT f_S EXCEPT FOR DSP MODE WHICH IS 2 x f_S.
3. TSCLKx FREQUENCY IS NORMALLY 64 x TFS BUT MAY BE OPERATED IN BURST MODE.

Figure 12-4. SPORT Stereo Serial Modes, Transmit

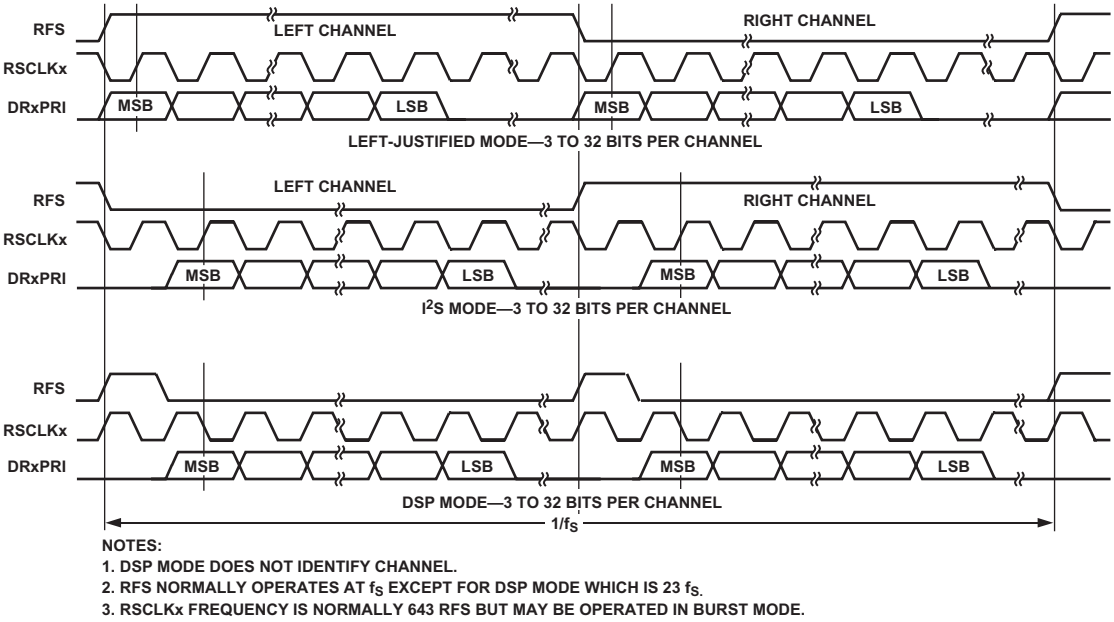


Figure 12-5. SPORT Stereo Serial Modes, Receive

Multichannel Operation

The SPORTs offer a multichannel mode of operation which allows the SPORT to communicate in a Time-Division-Multiplexed (TDM) serial system. In multichannel communications, each data word of the serial bit-stream occupies a separate channel. Each word belongs to the next consecutive channel so that, for example, a 24-word block of data contains one word for each of 24 channels.

The SPORT can automatically select words for particular channels while ignoring the others. Up to 128 channels are available for transmitting or receiving; each SPORT can receive and transmit data selectively from any of the 128 channels. These 128 channels can be any 128 out of the 1024

total channels. RX and TX must use the same 128-channel region to selectively enable channels. The SPORT can do any of the following on each channel:

- Transmit data
- Receive data
- Transmit and receive data
- Do nothing

Data companding and DMA transfers can also be used in multichannel mode.

The DTPRI pin is always driven (not three-stated) if the SPORT is enabled (TSPEN = 1 in the SPORTx_TCR1 register), unless it is in multichannel mode and an inactive time slot occurs. The DTSEC pin is always driven (not three-stated) if the SPORT is enabled and the secondary transmit is enabled (TXSE = 1 in the SPORTx_TCR2 register), unless the SPORT is in multichannel mode and an inactive time slot occurs.

The SPORT multichannel transmit select register and the SPORT multichannel receive select register must be programmed before enabling SPORTx_TX or SPORTx_RX operation for multichannel mode. This is especially important in "DMA data unpacked mode," since SPORT FIFO operation begins immediately after RSPEN and TSPEN are set, enabling both RX and TX. The MCMEN bit (in SPORTx_MCMC2) must be enabled prior to enabling SPORTx_TX or SPORTx_RX operation. When disabling the SPORT from multichannel operation, first disable TXEN and then disable RXEN. Note both TXEN and RXEN must be disabled before reenabling. Disabling only TX or RX is not allowed.

Figure 12-6 shows example timing for a multichannel transfer that has these characteristics:

- Use TDM method where serial data is sent or received on different channels sharing the same serial bus
- Can independently select transmit and receive channels
- RFS signals start of frame
- TFS is used as "transmit data valid" for external logic, true only during transmit channels
- Receive on channels 0 and 2, transmit on channels 1 and 2
- Multichannel frame delay is set to 1

See "Timing Examples" on page 12-39 for more examples.

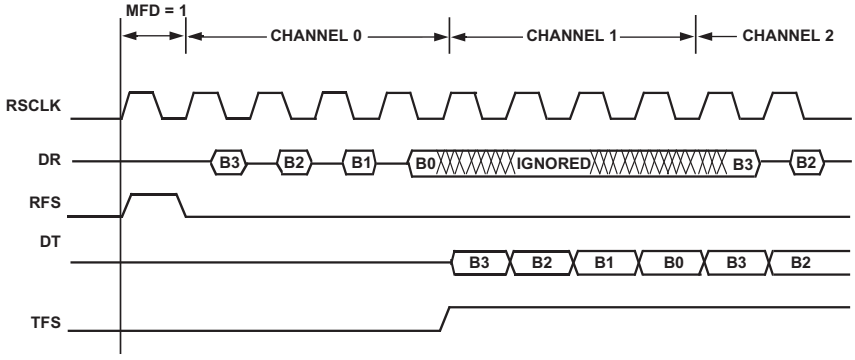


Figure 12-6. Multichannel Operation

Multichannel Enable

Setting the MCMEN bit in the SPORTx_MCM2 register enables multichannel mode. When MCMEN = 1, multichannel operation is enabled; when MCMEN = 0, all multichannel operations are disabled.

Setting the MCMEN bit enables multichannel operation for *both* the receive and transmit sides of the SPORT. Therefore, if a receiving SPORT is in multichannel mode, the transmitting SPORT must also be in multichannel mode.

When in multichannel mode, do not enable the stereo serial frame sync modes or the late frame sync feature, as these features are incompatible with multichannel mode.

Table 12-3 shows the dependencies of bits in the SPORT configuration register when the SPORT is in multichannel mode.

Table 12-3. Multichannel Mode Configuration

SPORTx_RCR1 or SPORTx_RCR2	SPORTx_TCR1 or SPORTx_TCR2	Notes
RSPEN	TSPEN	Set or clear both
IRCLK	-	Independent
-	ITCLK	Independent
RDTYPE	TDTYPE	Independent
RLSBIT	TLSBIT	Independent
IRFS	-	Independent
-	ITFS	Ignored
RFSR	TFSR	Ignored
-	DITFS	Ignored
LRFS	LTFS	Independent
LARFS	LATFS	Both must be 0

Table 12-3. Multichannel Mode Configuration (Cont'd)

SPORTx_RCR1 or SPORTx_RCR2	SPORTx_TCR1 or SPORTx_TCR2	Notes
RCKFE	TCKFE	Set or clear both to same value
SLEN	SLEN	Set or clear both to same value
RXSE	TXSE	Independent
RSFSE	TSFSE	Both must be 0
RRFST	TRFST	Ignored

Frame Syncs in Multichannel Mode

All receiving and transmitting devices in a multichannel system must have the same timing reference. The RFS signal is used for this reference, indicating the start of a block or frame of multichannel data words.

When multichannel mode is enabled on a SPORT, both the transmitter and the receiver use RFS as a frame sync. This is true whether RFS is generated internally or externally. The RFS signal is used to synchronize the channels and restart each multichannel sequence. Assertion of RFS indicates the beginning of the channel 0 data word.

Since RFS is used by both the SPORTx_TX and SPORTx_RX channels of the SPORT in multichannel mode configuration, the corresponding bit pairs in SPORTx_RCR1 and SPORTx_TCR1, and in SPORTx_RCR2 and SPORTx_TCR2, should always be programmed identically, with the possible exception of the RXSE and TXSE pair and the RDTYPE and TDTYPE pair. This is true even if SPORTx_RX operation is not enabled.

In multichannel mode, RFS timing similar to late (alternative) frame mode is entered automatically; the first bit of the transmit data word is available and the first bit of the receive data word is sampled in the same serial clock cycle that the frame sync is asserted, provided that MFD is set to 0.

The TFS signal is used as a transmit data valid signal which is active during transmission of an enabled word. The SPORT's data transmit pin is three-stated when the time slot is not active, and the TFS signal serves as an output-enabled signal for the data transmit pin. The SPORT drives TFS in multichannel mode whether or not ITFS is cleared. The TFS pin in multichannel mode still obeys the LTFS bit. If LTFS is set, the transmit data valid signal will be active low—a low signal on the TFS pin indicates an active channel.

Once the initial RFS is received, and a frame transfer has started, all other RFS signals are ignored by the SPORT until the complete frame has been transferred.

If MFD > 0, the RFS may occur during the last channels of a previous frame. This is acceptable, and the frame sync is not ignored as long as the delayed channel 0 starting point falls outside the complete frame.

In multichannel mode, the RFS signal is used for the block or frame start reference, after which the word transfers are performed continuously with no further RFS signals required. Therefore, internally generated frame syncs are always data independent.

The Multichannel Frame

A multichannel frame contains more than one channel, as specified by the window size and window offset. A complete multichannel frame consists of 1 – 1024 channels, starting with channel 0. The particular channels of the multichannel frame that are selected for the SPORT are a combination of the window offset, the window size, and the multichannel select registers. See Figure 12-7.

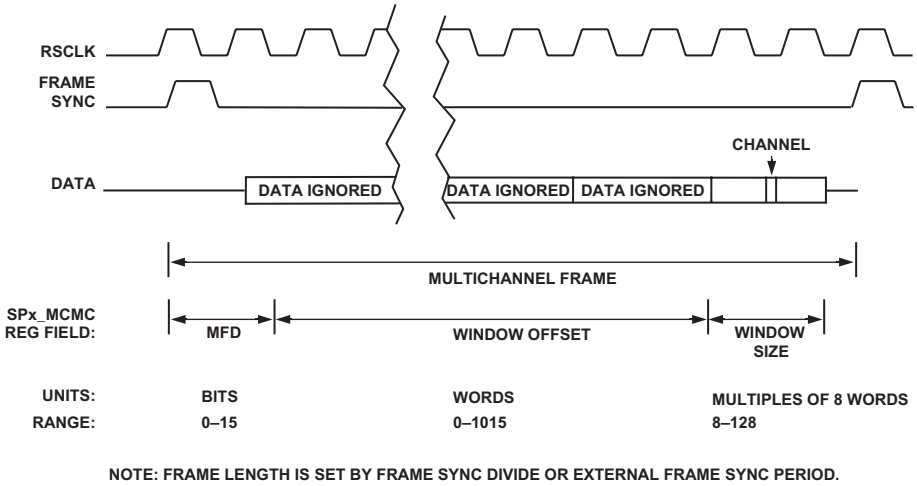


Figure 12-7. Relationships for Multichannel Parameters

Multichannel Frame Delay

The 4-bit `MFD` field in `SPORTx_MCMC2` specifies a delay between the frame sync pulse and the first data bit in multichannel mode. The value of `MFD` is the number of serial clock cycles of the delay. Multichannel frame delay allows the processor to work with different types of interface devices.

A value of 0 for `MFD` causes the frame sync to be concurrent with the first data bit. The maximum value allowed for `MFD` is 15. A new frame sync may occur before data from the last frame has been received, because blocks of data occur back-to-back.

Window Size

The window size (`WSIZE[3:0]`) defines the number of channels that can be enabled/disabled by the multichannel select registers. This range of words is called the active window. The number of channels can be any value in the range of 0 to 15, corresponding to active window size of 8 to 128, in

increments of 8; the default value of 0 corresponds to a minimum active window size of 8 channels. To calculate the active window size from the WSIZE register, use this equation:

```
Number of words in active window = 8 x (WSIZE + 1)
```

Since the DMA buffer size is always fixed, it is possible to define a smaller window size (for example, 32 words), resulting in a smaller DMA buffer size (in this example, 32 words instead of 128 words) to save DMA bandwidth. The window size cannot be changed while the SPORT is enabled.

Multichannel select bits that are enabled but fall outside the window selected are ignored.

Window Offset

The window offset (WOFF[9:0]) specifies where in the 1024-channel range to place the start of the active window. A value of 0 specifies no offset and 896 is the largest value that permits using all 128 channels. As an example, a program could define an active window with a window size of 8 (WSIZE = 0) and an offset of 93 (WOFF = 93). This 8-channel window would reside in the range from 93 to 100. Neither the window offset nor the window size can be changed while the SPORT is enabled.

If the combination of the window size and the window offset would place any portion of the window outside of the range of the channel counter, none of the out-of-range channels in the frame are enabled.

Other Multichannel Fields in SPORTx_MCMC2

The FSDR bit in the SPORTx_MCMC2 register changes the timing relationship between the frame sync and the clock received. This change enables the SPORT to comply with the H.100 protocol.

Normally (When FSDR = 0), the data is transmitted on the same edge that the TFS is generated. For example, a positive edge on TFS causes data to be transmitted on the positive edge of the TSCLK—either the same edge or the following one, depending on when LATFS is set.

When the frame sync/data relationship is used (FSDR = 1), the frame sync is expected to change on the falling edge of the clock and is sampled on the rising edge of the clock. This is true even though data received is sampled on the negative edge of the receive clock.

Channel Selection Register

A channel is a multibit word from 3 to 32 bits in length that belongs to one of the TDM channels. Specific channels can be individually enabled or disabled to select which words are received and transmitted during multichannel communications. Data words from the enabled channels are received or transmitted, while disabled channel words are ignored. Up to 128 contiguous channels may be selected out of 1024 available channels. The SPORTx_MRCSn and SPORTx_MTCSn multichannel select registers are used to enable and disable individual channels; the SPORTx_MRCSn registers specify the active receive channels, and the SPORTx_MTCSn registers specify the active transmit channels.

Four registers make up each multichannel select register. Each of the four registers has 32 bits, corresponding to 32 channels. Setting a bit enables that channel, so the SPORT selects its word from the multiple word block of data (for either receive or transmit). See Figure 12-8.

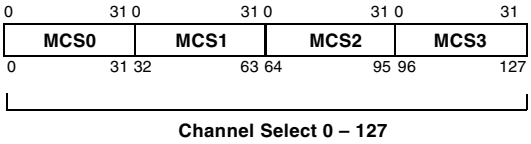


Figure 12-8. Multichannel Select Registers

Channel select bit 0 always corresponds to the first word of the active window. To determine a channel's absolute position in the frame, add the window offset words to the channel select position. For example, setting bit 7 in MCS2 selects word 71 of the active window to be enabled. Setting bit 2 in MCS1 selects word 34 of the active window, and so on.

Setting a particular bit in the SPORTx_MTCSn register causes the SPORT to transmit the word in that channel's position of the datastream. Clearing the bit in the SPORTx_MTCSn register causes the SPORT's data transmit pin to three-state during the time slot of that channel.

Setting a particular bit in the SPORTx_MRCSn register causes the SPORT to receive the word in that channel's position of the datastream; the received word is loaded into the SPORTx_RX buffer. Clearing the bit in the SPORTx_MRCSn register causes the SPORT to ignore the data.

Companding may be selected for all channels or for no channels. A-law or µ-law companding is selected with the TDTYPE field in the SPORTx_TCR1 register and the RDTYPE field in the SPORTx_RCR1 register, and applies to all active channels. (See "Companding" on page 12-29 for more information about companding.)

Multichannel DMA Data Packing

Multichannel DMA data packing and unpacking are specified with the MCDTXPE and MCDRXPE bits in the SPORTx_MCMC2 multichannel configuration register.

If the bits are set, indicating that data is packed, the SPORT expects the data contained by the DMA buffer corresponds only to the enabled SPORT channels. For example, if an MCM frame contains 10 enabled channels, the SPORT expects the DMA buffer to contain 10 consecutive words for each frame. It is not possible to change the total number of enabled channels without changing the DMA buffer size, and reconfiguring is not allowed while the SPORT is enabled.

If the bits are cleared (the default, indicating that data is not packed), the SPORT expects the DMA buffer to have a word for each of the channels in the active window, whether enabled or not, so the DMA buffer size must be equal to the size of the window. For example, if channels 1 and 10 are enabled, and the window size is 16, the DMA buffer size would have to be 16 words (unless the secondary side is enabled). The data to be transmitted or received would be placed at addresses 1 and 10 of the buffer, and the rest of the words in the DMA buffer would be ignored. This mode allows changing the number of enabled channels while the SPORT is enabled, with some caution. First read the channel register to make sure that the active window is not being serviced. If the channel count is 0, then the multichannel select registers can be updated.

Support for H.100 Standard Protocol

The processor supports the H.100 standard protocol. The following SPORT parameters must be set to support this standard.

- Set for external frame sync. Frame sync generated by external bus master.
- `TFSR/RFSR` set (frame syncs required)
- `LTFS/LRFS` set (active low frame syncs)
- Set for external clock
- `MCMEN` set (multichannel mode selected)
- `MFD` = 0 (no frame delay between frame sync and first data bit)
- `SLEN` = 7 (8-bit words)
- `FSDR` = 1 (set for H.100 configuration, enabling half-clock-cycle early frame sync)

2X Clock Recovery Control

The SPORTs can recover the data rate clock from a provided 2X input clock. This enables the implementation of H.100 compatibility modes for MVIP-90 (2 Mbps data) and HMVIP (8 Mbps data), by recovering 2 MHz from 4 MHz or 8 MHz from the 16 MHz incoming clock with the proper phase relationship. A 2-bit mode signal (MCCRM[1:0] in the SPORTx_MCMC2 register) chooses the applicable clock mode, which includes a non-divide or bypass mode for normal operation. A value of MCCRM = 00 chooses non-divide or bypass mode (H.100-compatible), MCCRM = 10 chooses MVIP-90 clock divide (extract 2 MHz from 4 MHz), and MCCRM = 11 chooses HMVIP clock divide (extract 8 MHz from 16 MHz).

Functional Description

The following sections provide a functional description of the SPORTs.

Clock and Frame Sync Frequencies

The maximum serial clock frequency (for either an internal source or an external source) is SCLK/2. The frequency of an internally generated clock is a function of the system clock frequency (SCLK) and the value of the 16-bit serial clock divide modulus registers, SPORTx_TCLKDIV and SPORTx_RCLKDIV.

```
TSCLKx frequency = (SCLK frequency)/(2 x (SPORTx_TCLKDIV + 1))
RSCLKx frequency = (SCLK frequency)/(2 x (SPORTx_RCLKDIV + 1))
```

If the value of SPORTx_TCLKDIV or SPORTx_RCLKDIV is changed while the internal serial clock is enabled, the change in TSCLK or RSCLK frequency takes effect at the start of the drive edge of TSCLK or RSCLK that follows the next leading edge of TFS or RFS.

When an internal frame sync is selected (ITFS = 1 in the SPORTx_TCR1 register or IRFS = 1 in the SPORTx_RCR1 register) and frame syncs are not required, the first frame sync does not update the clock divider if the value in SPORTx_TCLKDIV or SPORTx_RCLKDIV has changed. The second frame sync will cause the update.

The SPORTx_TFSDIV and SPORTx_RFSDIV registers specify the number of transmit or receive clock cycles that are counted before generating a TFS or RFS pulse (when the frame sync is internally generated). This enables a frame sync to initiate periodic transfers. The counting of serial clock cycles applies to either internally or externally generated serial clocks.

The formula for the number of cycles between frame sync pulses is:

```
# of transmit serial clocks between frame sync assertions =
TFSDIV + 1
# of receive serial clocks between frame sync assertions =
RFSDIV + 1
```

Use the following equations to determine the correct value of TFSDIV or RFSDIV, given the serial clock frequency and desired frame sync frequency:

```
SPORTxTFS frequency = (TSCLKx frequency)/(SPORTx_TFSDIV + 1)

SPORTxRFS frequency = (RSCLKx frequency)/(SPORTx_RFSDIV + 1)
```

The frame sync would thus be continuously active (for transmit if TFSDIV = 0 or for receive if RFSDIV = 0). However, the value of TFSDIV (or RFSDIV) should not be less than the serial word length minus 1 (the value of the SLEN field in SPORTx_TCR2 or SPORTx_RCR2). A smaller value could cause an external device to abort the current operation or have other unpredictable results. If a SPORT is not being used, the TFSDIV (or RFSDIV) divisor can be used as a counter for dividing an external clock or for generating a periodic pulse or periodic interrupt. The SPORT must be enabled for this mode of operation to work.

Maximum Clock Rate Restrictions

Externally generated late transmit frame syncs also experience a delay from arrival to data output, and this can limit the maximum serial clock speed. See the product data sheet for exact timing specifications.

Word Length

Each SPORT channel (transmit and receive) independently handles word lengths of 3 to 32 bits. The data is right-justified in the SPORT data registers if it is fewer than 32 bits long, residing in the LSB positions. The value of the serial word length (SLEN) field in the SPORTx_TCR2 and SPORTx_RCR2 registers of each SPORT determines the word length according to this formula:

```
Serial Word Length = SLEN + 1
```

The SLEN value should not be set to 0 or 1; values from 2 to 31 are allowed. Continuous operation (when the last bit of the current word is immediately followed by the first bit of the next word) is restricted to word sizes of 4 or longer (so SLEN ≥ 3).

Bit Order

Bit order determines whether the serial word is transmitted MSB first or LSB first. Bit order is selected by the RLSBIT and TLSBIT bits in the SPORTx_RCR1 and SPORTx_TCR1 registers. When RLSBIT (or TLSBIT) = 0, serial words are received (or transmitted) MSB first. When RLSBIT (or TLSBIT) = 1, serial words are received (or transmitted) LSB first.

Data Type

The TDTYPE field of the SPORTx_TCR1 register and the RDTYPE field of the SPORTx_RCR1 register specify one of four data formats for both single and multichannel operation. See Table 12-4.

Table 12-4. TDTYPE, RDTYPE, and Data Formatting

TDTYPE or RDTYPE	SPORTx_TCR1 Data Formatting	SPORTx_RCR1 Data Formatting
00	Normal operation	Zero fill
01	Reserved	Sign extend
10	Compand using μ-law	Compand using μ-law
11	Compand using A-law	Compand using A-law

These formats are applied to serial data words loaded into the SPORTx_RX and SPORTx_TX buffers. SPORTx_TX data words are not actually zero filled or sign extended, because only the significant bits are transmitted.

Companding

Companding (a contraction of COMpressing and exPANDing) is the process of logarithmically encoding and decoding data to minimize the number of bits that must be sent. The SPORTs support the two most widely used companding algorithms, μ-law and A-law. The processor compands data according to the CCITT G.711 specification. The type of companding can be selected independently for each SPORT.

When companding is enabled, valid data in the SPORTx_RX register is the right-justified, expanded value of the eight LSBs received and sign extended to 16 bits. A write to SPORTx_TX causes the 16-bit value to be compressed to eight LSBs (sign extended to the width of the transmit word) and written to the internal transmit register. Although the companding standards support only 13-bit (A-law) or 14-bit (μ-law) maximum word lengths, up to 16-bit word lengths can be used. If the magnitude of the word value is greater than the maximum allowed, the value is automatically compressed to the maximum positive or negative value.

Lengths greater than 16 bits are not supported for companding operation.

Clock Signal Options

Each SPORT has a transmit clock signal (TSCLK) and a receive clock signal (RSCLK). The clock signals are configured by the TCKFE and RCKFE bits of the SPORTx_TCR1 and SPORTx_RCR1 registers. Serial clock frequency is configured in the SPORTx_TCLKDIV and SPORTx_RCLKDIV registers.

The receive clock pin may be tied to the transmit clock if a single clock is desired for both receive and transmit.

Both transmit and receive clocks can be independently generated internally or input from an external source. The ITCLK bit of the SPORTx_TCR1 configuration register and the IRCLK bit in the SPORTx_RCR1 configuration register determines the clock source.

When IRCLK or ITCLK = 1, the clock signal is generated internally by the processor, and the TSCLK or RSCLK pin is an output. The clock frequency is determined by the value of the serial clock divisor in the SPORTx_RCLKDIV register.

When IRCLK or ITCLK = 0, the clock signal is accepted as an input on the TSCLK or RSCLK pins, and the serial clock divisors in the SPORTx_TCLKDIV/SPORTx_RCLKDIV registers are ignored. The externally generated serial clocks do not need to be synchronous with the system clock or with each other. The system clock must have a higher frequency than RSCLK and TSCLK.

Frame Sync Options

Framing signals indicate the beginning of each serial word transfer. The framing signals for each SPORT are TFS (transmit frame sync) and RFS (receive frame sync). A variety of framing options are available; these options are configured in the SPORT configuration registers (SPORTx_TCR1, SPORTx_TCR2, SPORTx_RCR1 and SPORTx_RCR2). The TFS and RFS signals of a SPORT are independent and are separately configured in the control registers.

Framed Versus Unframed

The use of multiple frame sync signals is optional in SPORT communications. The TFSR (transmit frame sync required select) and RFSR (receive frame sync required select) control bits determine whether frame sync signals are required. These bits are located in the SPORTx_TCR1 and SPORTx_RCR1 registers.

When TFSR = 1 or RFSR = 1, a frame sync signal is required for every data word. To allow continuous transmitting by the SPORT, each new data word must be loaded into the SPORTx_TX hold register before the previous word is shifted out and transmitted.

When TFSR = 0 or RFSR = 0, the corresponding frame sync signal is not required. A single frame sync is needed to initiate communications but is ignored after the first bit is transferred. Data words are then transferred continuously, unframed.

With frame syncs not required, interrupt or DMA requests may not be serviced frequently enough to guarantee continuous unframed data flow. Monitor status bits or check for a SPORT Error interrupt to detect underflow or overflow of data.

Figure 12-9 illustrates framed serial transfers, which have these characteristics:

- TFSR and RFSR bits in the SPORTx_TCR1 and SPORTx_RCR1 registers determine framed or unframed mode.
- Framed mode requires a framing signal for every word. Unframed mode ignores a framing signal after the first word.
- Unframed mode is appropriate for continuous reception.
- Active low or active high frame syncs are selected with the LTFS and LRFS bits of the SPORTx_TCR1 and SPORTx_RCR1 registers.

See "Timing Examples" on page 12-39 for more timing examples.

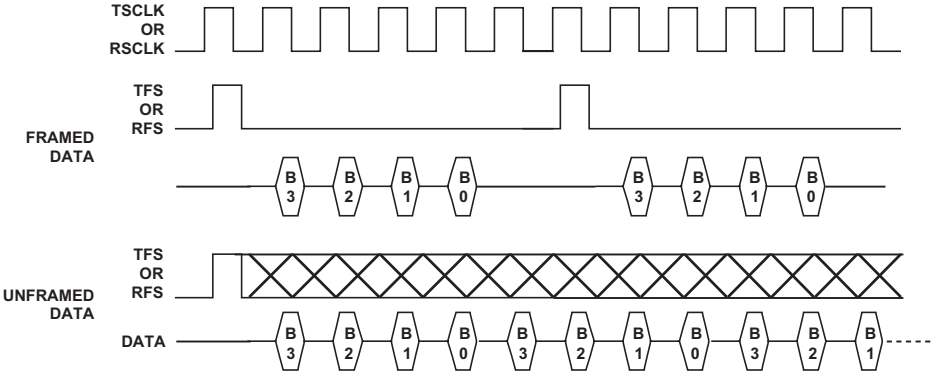


Figure 12-9. Framed Versus Unframed Data

Internal Versus External Frame Syncs

Both transmit and receive frame syncs can be independently generated internally or can be input from an external source. The `ITFS` and `IRFS` bits of the `SPORTx_TCR1` and `SPORTx_RCR1` registers determine the frame sync source.

When `ITFS = 1` or `IRFS = 1`, the corresponding frame sync signal is generated internally by the SPORT, and the `TFS` pin or `RFS` pin is an output. The frequency of the frame sync signal is determined by the value of the frame sync divisor in the `SPORTx_TFSDIV` or `SPORTx_RFSDIV` register.

When `ITFS = 0` or `IRFS = 0`, the corresponding frame sync signal is accepted as an input on the `TFS` pin or `RFS` pin, and the frame sync divisors in the `SPORTx_TFSDIV`/`SPORTx_RFSDIV` registers are ignored.

All of the frame sync options are available whether the signal is generated internally or externally.

Active Low Versus Active High Frame Syncs

Frame sync signals may be either active high or active low (in other words, inverted). The LTFS and LRFS bits of the SPORTx_TCR1 and SPORTx_RCR1 registers determine frame sync logic levels:

- When LTFS = 0 or LRFS = 0, the corresponding frame sync signal is active high.
- When LTFS = 1 or LRFS = 1, the corresponding frame sync signal is active low.

Active high frame syncs are the default. The LTFS and LRFS bits are initialized to 0 after a processor reset.

Sampling Edge for Data and Frame Syncs

Data and frame syncs can be sampled on either the rising or falling edges of the SPORT clock signals. The TCKFE and RCKFE bits of the SPORTx_TCR1 and SPORTx_RCR1 registers select the driving and sampling edges of the serial data and frame syncs.

For the SPORT transmitter, setting TCKFE = 1 in the SPORTx_TCR1 register selects the falling edge of TSCLKx to drive data and internally generated frame syncs and selects the rising edge of TSCLKx to sample externally generated frame syncs. Setting TCKFE = 0 selects the rising edge of TSCLKx to drive data and internally generated frame syncs and selects the falling edge of TSCLKx to sample externally generated frame syncs.

For the SPORT receiver, setting RCKFE = 1 in the SPORTx_RCR1 register selects the falling edge of RSCLKx to drive internally generated frame syncs and selects the rising edge of RSCLKx to sample data and externally generated frame syncs. Setting RCKFE = 0 selects the rising edge of RSCLKx to drive internally generated frame syncs and selects the falling edge of RSCLKx to sample data and externally generated frame syncs.

Note externally generated data and frame sync signals should change state on the opposite edge than that selected for sampling. For example, for an externally generated frame sync to be sampled on the rising edge of the clock (TCKFE = 1 in the SPORTx_TCR1 register), the frame sync must be driven on the falling edge of the clock.

The transmit and receive functions of two SPORTs connected together should always select the same value for TCKFE in the transmitter and RCKFE in the receiver, so that the transmitter drives the data on one edge and the receiver samples the data on the opposite edge.

In Figure 12-10, TCKFE = RCKFE = 0 and transmit and receive are connected together to share the same clock and frame syncs.

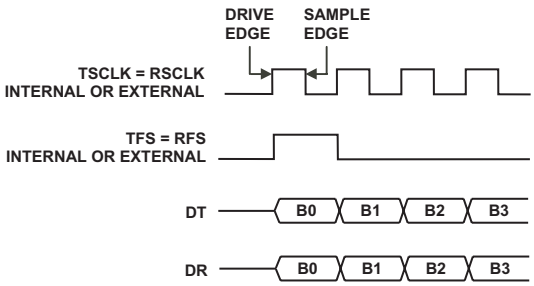


Figure 12-10. Example of TCKFE = RCKFE = 0, Transmit and Receive Connected

In Figure 12-11, TCKFE = RCKFE = 1 and transmit and receive are connected together to share the same clock and frame syncs.

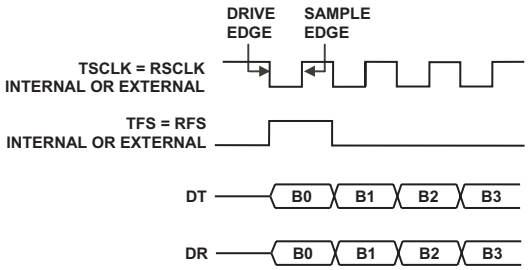


Figure 12-11. Example of TCKFE = RCKFE = 1, Transmit and Receive Connected

Early Versus Late Frame Syncs (Normal Versus Alternate Timing)

Frame sync signals can occur during the first bit of each data word (late) or during the serial clock cycle immediately preceding the first bit (early). The LATFS and LARFS bits of the SPORTx_TCR1 and SPORTx_RCR1 registers configure this option.

When LATFS = 0 or LARFS = 0, early frame syncs are configured; this is the normal mode of operation. In this mode, the first bit of the transmit data word is available and the first bit of the receive data word is sampled in the serial clock cycle after the frame sync is asserted, and the frame sync is not checked again until the entire word has been transmitted or received. In multichannel operation, this corresponds to the case when multichannel frame delay is 1.

If data transmission is continuous in early framing mode (in other words, the last bit of each word is immediately followed by the first bit of the next word), then the frame sync signal occurs during the last bit of each word. Internally generated frame syncs are asserted for one clock cycle in early framing mode. Continuous operation is restricted to word sizes of 4 or longer (SLEN ≥ 3).

When LATFS = 1 or LARFS = 1, late frame syncs are configured; this is the alternate mode of operation. In this mode, the first bit of the transmit data word is available and the first bit of the receive data word is sampled in the same serial clock cycle that the frame sync is asserted. In multichannel operation, this is the case when frame delay is 0. Receive data bits are sampled by serial clock edges, but the frame sync signal is only checked during the first bit of each word. Internally generated frame syncs remain asserted for the entire length of the data word in late framing mode. Externally generated frame syncs are only checked during the first bit.

Figure 12-12 illustrates the two modes of frame signal timing. In summary:

- For the LATFS or LARFS bits of the SPORTx_TCR1 or SPORTx_RCR1 registers: LATFS = 0 or LARFS = 0 for early frame syncs, LATFS = 1 or LARFS = 1 for late frame syncs.
- For early framing, the frame sync precedes data by one cycle. For late framing, the frame sync is checked on the first bit only.
- Data is transmitted MSB first (TLSBIT = 0 or RLSBIT = 0) or LSB first (TLSBIT = 1 or RLSBIT = 1).
- Frame sync and clock are generated internally or externally.

See "Timing Examples" on page 12-39 for more examples.

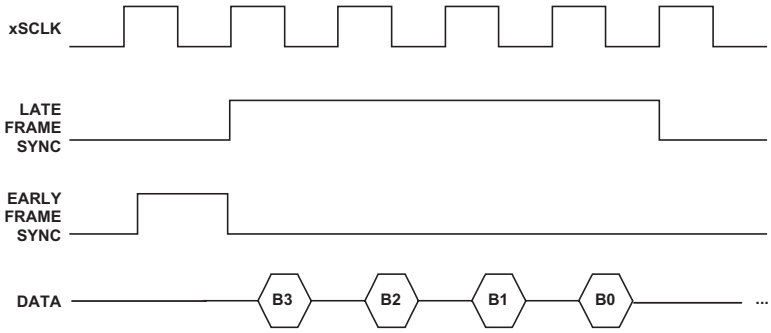


Figure 12-12. Normal Versus Alternate Framing

Data Independent Transmit Frame Sync

Normally the internally generated transmit frame sync signal (TFS) is output only when the SPORTx_TX buffer has data ready to transmit. The data-independent transmit frame sync select bit (DITFS) allows the continuous generation of the TFS signal, with or without new data. The DITFS bit of the SPORTx_TCR1 register configures this option.

When DITFS = 0, the internally generated TFS is only output when a new data word has been loaded into the SPORTx_TX buffer. The next TFS is generated once data is loaded into SPORTx_TX. This mode of operation allows data to be transmitted only when it is available.

When DITFS = 1, the internally generated TFS is output at its programmed interval regardless of whether new data is available in the SPORTx_TX buffer. Whatever data is present in SPORTx_TX is transmitted again with each assertion of TFS. The TUVF (transmit underflow status) bit in the SPORTx_STAT register is set when this occurs and old data is retransmitted. The TUVF status bit is also set if the SPORTx_TX buffer does not have new data when an externally generated TFS occurs. Note that in this mode of operation, data is transmitted only at specified times.

If the internally generated TFS is used, a single write to the SPORTx_TX data register is required to start the transfer.

Moving Data Between SPORTs and Memory

Transmit and receive data can be transferred between the SPORTs and on-chip memory in one of two ways: with single word transfers or with DMA block transfers.

If no SPORT DMA channel is enabled, the SPORT generates an interrupt every time it has received a data word or needs a data word to transmit. SPORT DMA provides a mechanism for receiving or transmitting an entire block or multiple blocks of serial data before the interrupt is generated. The SPORT's DMA controller handles the DMA transfer, allowing the processor core to continue running until the entire block of data is transmitted or received. Interrupt service routines (ISRs) can then operate on the block of data rather than on single words, significantly reducing overhead.

For information about DMA, see Chapter 5, "Direct Memory Access".

SPORT RX, TX, and Error Interrupts

The SPORT RX interrupt is asserted when RSPEN is enabled and any words are present in the RX FIFO. If RX DMA is enabled, the SPORT RX interrupt is turned off and DMA services the RX FIFO.

The SPORT TX interrupt is asserted when TSPEN is enabled and the TX FIFO has room for words. If TX DMA is enabled, the SPORT TX interrupt is turned off and DMA services the TX FIFO.

The SPORT error interrupt is asserted when any of the sticky status bits (ROVF, RUVF, TOVF, TUVF) are set. The ROVF and RUVF bits are cleared by writing 0 to RSPEN. The TOVF and TUVF bits are cleared by writing 0 to TSPEN.

PAB Errors

The SPORT generates a PAB error for illegal register read or write operations. Examples include:

- Reading a write-only register (for example, SPORT_TX)
- Writing a read-only register (for example, SPORT_RX)
- Writing or reading a register with the wrong size (for example, 32-bit read of a 16-bit register)
- Accessing reserved register locations

Timing Examples

Several timing examples are included within the text of this chapter (in the sections "Framed Versus Unframed" on page 12-31, "Early Versus Late Frame Syncs (Normal Versus Alternate Timing)" on page 12-35, and "Frame Syncs in Multichannel Mode" on page 12-19). This section contains additional examples to illustrate other possible combinations of the framing options.

These timing examples show the relationships between the signals but are not scaled to show the actual timing parameters of the processor. Consult the product data sheet for actual timing parameters and values.

These examples assume a word length of four bits (SLEN = 3). Framing signals are active high (LRFS = 0 and LTFS = 0).

Figure 12-13 through Figure 12-18 show framing for receiving data.

In Figure 12-13 and Figure 12-14, the normal framing mode is shown for non-continuous data (any number of TSCLK or RSCLK cycles between words) and continuous data (no TSCLK or SCLK cycles between words).

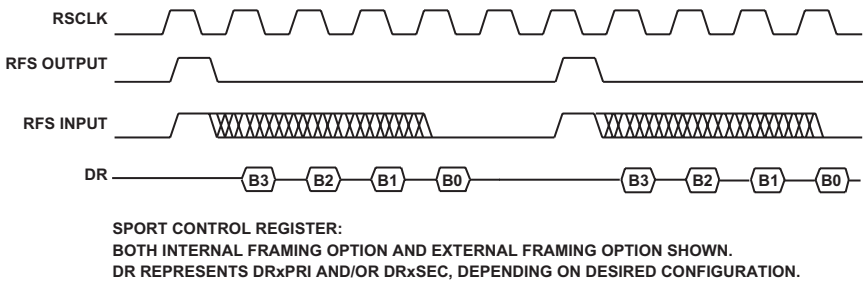


Figure 12-13. SPORT Receive, Normal Framing

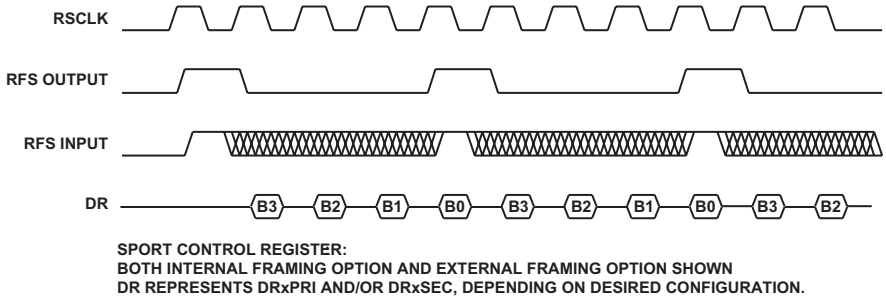


Figure 12-14. SPORT Continuous Receive, Normal Framing

Figure 12-15 and Figure 12-16 show non-continuous and continuous receiving in the alternate framing mode. These four figures show the input timing requirement for an externally generated frame sync and also the output timing characteristic of an internally generated frame sync. Note

the output meets the input timing requirement; therefore, with two SPORT channels used, one SPORT channel could provide RFS for the other SPORT channel.

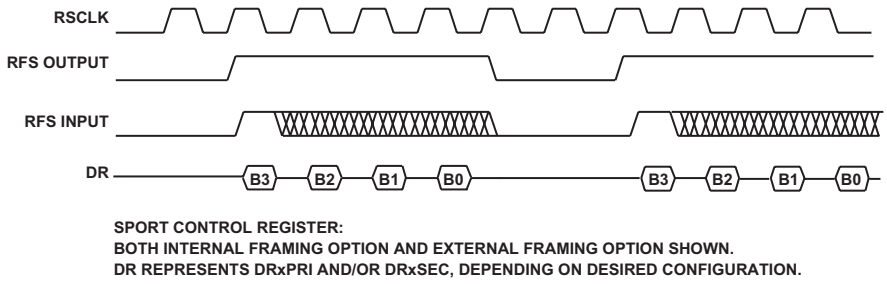


Figure 12-15. SPORT Receive, Alternate Framing

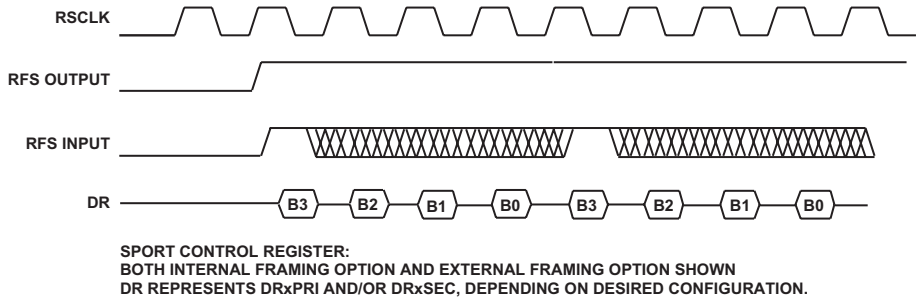


Figure 12-16. SPORT Continuous Receive, Alternate Framing

Figure 12-17 and Figure 12-18 show the receive operation with normal framing and alternate framing, respectively, in the unframed mode. A single frame sync signal occurs only at the start of the first word, either one

RSCLK before the first bit (in normal mode) or at the same time as the first bit (in alternate mode). This mode is appropriate for multiword bursts (continuous reception).

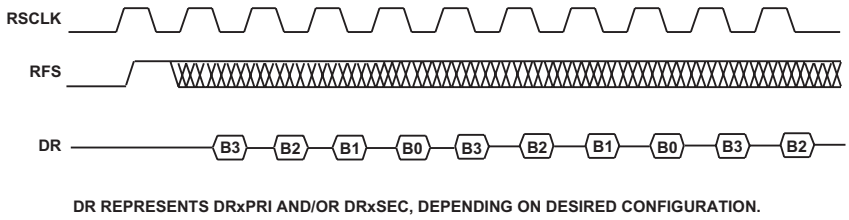


Figure 12-17. SPORT Receive, Unframed Mode, Normal Framing

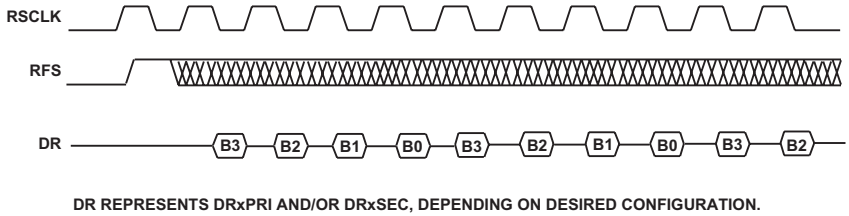


Figure 12-18. SPORT Receive, Unframed Mode, Alternate Framing

Figure 12-19 through Figure 12-24 show framing for transmitting data and are very similar to Figure 12-13 through Figure 12-18.

In Figure 12-19 and Figure 12-20, the normal framing mode is shown for non-continuous data (any number of TSCLK cycles between words) and continuous data (no TSCLK cycles between words). Figure 12-21 and

Figure 12-22 show non-continuous and continuous transmission in the alternate framing mode. As noted previously for the receive timing diagrams, the RFS output meets the RFS input timing requirement.

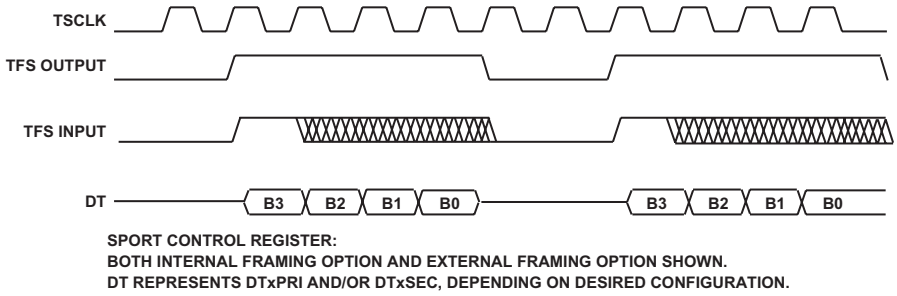


Figure 12-19. SPORT Transmit, Normal Framing

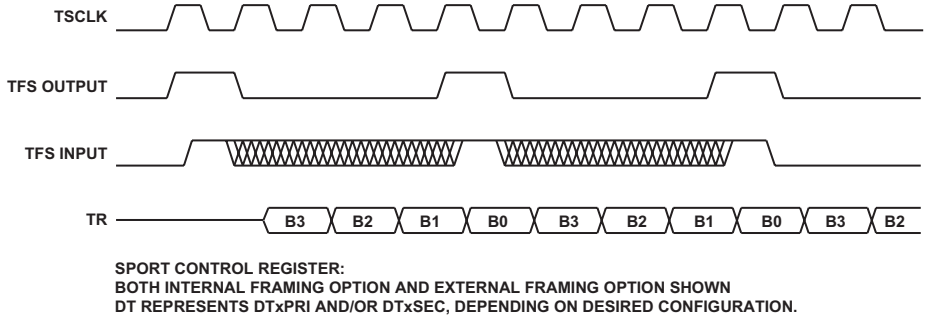


Figure 12-20. SPORT Continuous Transmit, Normal Framing

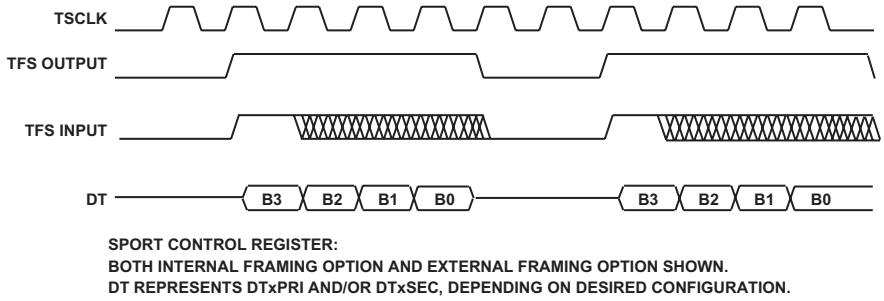


Figure 12-21. SPORT Transmit, Alternate Framing

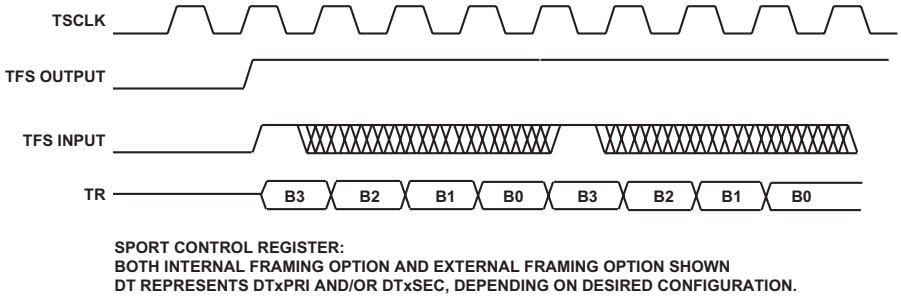


Figure 12-22. SPORT Continuous Transmit, Alternate Framing

Figure 12-23 and Figure 12-24 show the transmit operation with normal framing and alternate framing, respectively, in the unframed mode. A single frame sync signal occurs only at the start of the first word, either one TSCLK before the first bit (in normal mode) or at the same time as the first bit (in alternate mode).

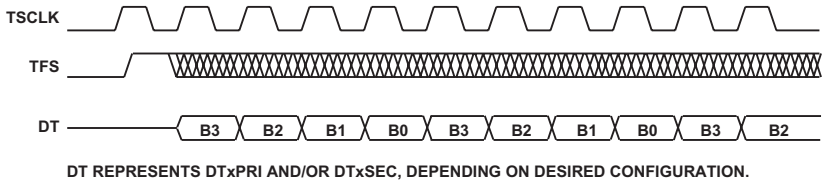


Figure 12-23. SPORT Transmit, Unframed Mode, Normal Framing

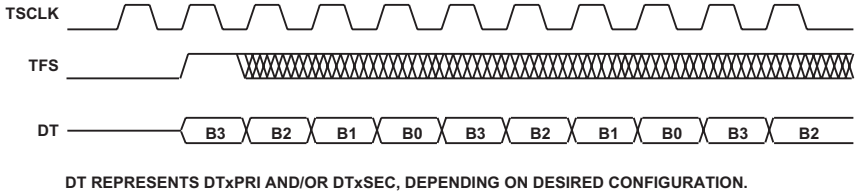


Figure 12-24. SPORT Transmit, Unframed Mode, Alternate Framing

SPORT Registers

The following sections describe the SPORT registers. Table 12-5 provides an overview of the available control registers.

Table 12-5. SPORT Register Mapping

Register Name	Function	Notes
SPORTx_TCR1	Primary transmit configuration register	Bits [15:1] can only be written if bit 0 = 0
SPORTx_TCR2	Secondary transmit configuration register	
SPORTx_TCLK_DIV	Transmit clock divider register	Ignored if external SPORT clock mode is selected

Table 12-5. SPORT Register Mapping (Cont'd)

Register Name	Function	Notes
SPORTx_TFSDIV	Transmit frame sync divider register	Ignored if external frame sync mode is selected
SPORTx_TX	SPORT transmit data register	See description of FIFO buffering at "SPORTx_TX Register" on page 12-58
SPORTx_RCR1	Primary receive configuration register	Bits [15:1] can only be written if bit 0 = 0
SPORTx_RCR2	Secondary receive configuration register	
SPORTx_RCLK_DIV	Receive clock divider register	Ignored if external SPORT clock mode is selected
SPORTx_RFSDIV	Receive frame sync divider register	Ignored if external frame sync mode is selected
SPORTx_RX	SPORT receive data register	See description of FIFO buffering at "SPORTx_RX Register" on page 12-60
SPORTx_STAT	Receive and transmit status	
SPORTx_MCM1	Primary multichannel mode configuration register	Configure this register before enabling the SPORT
SPORTx_MCM2	Secondary multichannel mode configuration register	Configure this register before enabling the SPORT
SPORTx_MRCSn	Receive channel selection registers	Select or deselect channels in a multichannel frame
SPORTx_MTCSn	Transmit channel selection registers	Select or deselect channels in a multichannel frame
SPORTx_CHNL	Currently serviced channel in a multichannel frame	

Register Writes and Effective Latency

When the SPORT is disabled (TSPEN and RSPEN cleared), SPORT register writes are internally completed at the end of the SCLK cycle in which they occurred, and the register reads back the newly-written value on the next cycle.

When the SPORT is enabled to transmit (TSPEN set) or receive (RSPEN set), corresponding SPORT configuration register writes are disabled (except for SPORTx_RCLKDIV, SPORTx_TCLKDIV, and multichannel mode channel select registers). The SPORTx_TX register writes are always enabled; SPORTx_RX, SPORTx_CHNL, and SPORTx_STAT are read-only registers.

After a write to a SPORT register, while the SPORT is disabled, any changes to the control and mode bits generally take effect when the SPORT is re-enabled.

Most configuration registers can only be changed while the SPORT is disabled (TSPEN/RSPEN = 0). Changes take effect after the SPORT is re-enabled. The only exceptions to this rule are the TCLKDIV/RCLKDIV registers and multichannel select registers.

SPORTx_TCR1 and SPORTx_TCR2 Registers

The main control registers for the transmit portion of each SPORT are the transmit configuration registers, SPORTx_TCR1 and SPORTx_TCR2, shown in Figure 12-25 and Figure 12-26.

A SPORT is enabled for transmit if bit 0 (TSPEN) of the transmit configuration 1 register is set to 1. This bit is cleared during either a hard reset or a soft reset, disabling all SPORT transmission.

When the SPORT is enabled to transmit (TSPEN set), corresponding SPORT configuration register writes are not allowed except for SPORTx_TCLKDIV and multichannel mode channel select registers. Writes to disallowed registers have no effect. While the SPORT is enabled, SPORTx_TCR1 is not written except for bit 0 (TSPEN). For example:

```
write (SPORTx_TCR1, 0x0001) ;   /* SPORT TX Enabled */
write (SPORTx_TCR1, 0xFF01) ;   /* ignored, no effect */
write(SPORTx_TCR1,0xFFF0);   /*SPORTdisabled,SPORTx_TCR1
still equal to 0x0000 */
```

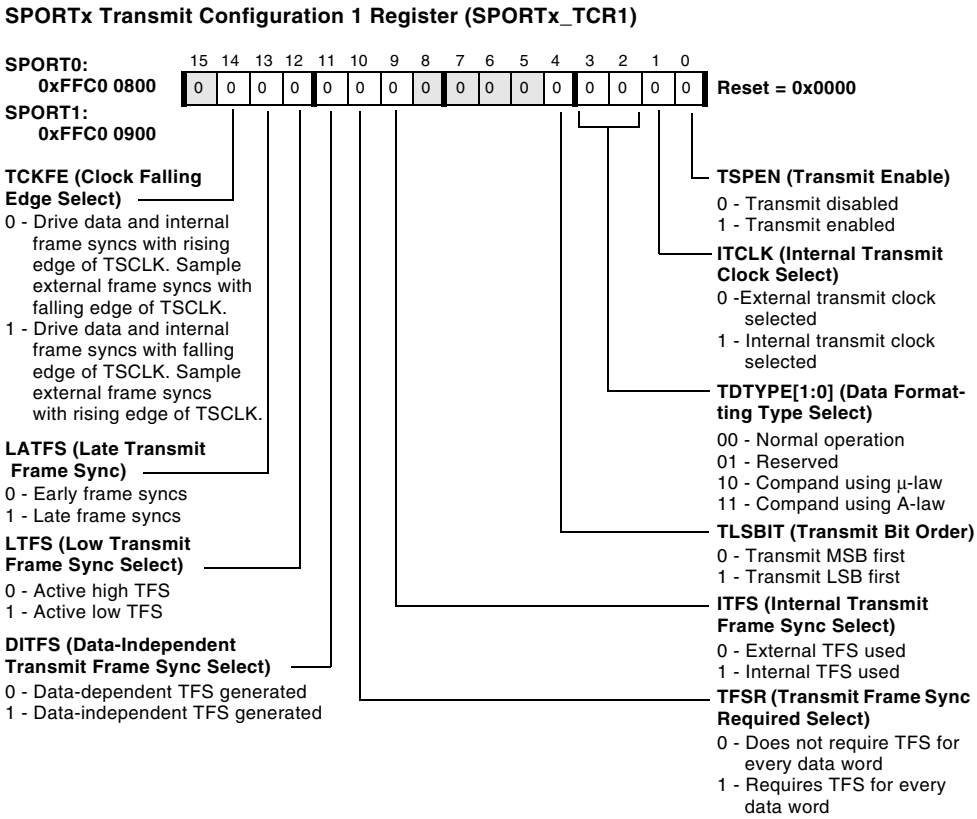


Figure 12-25. SPORTx Transmit Configuration 1 Register

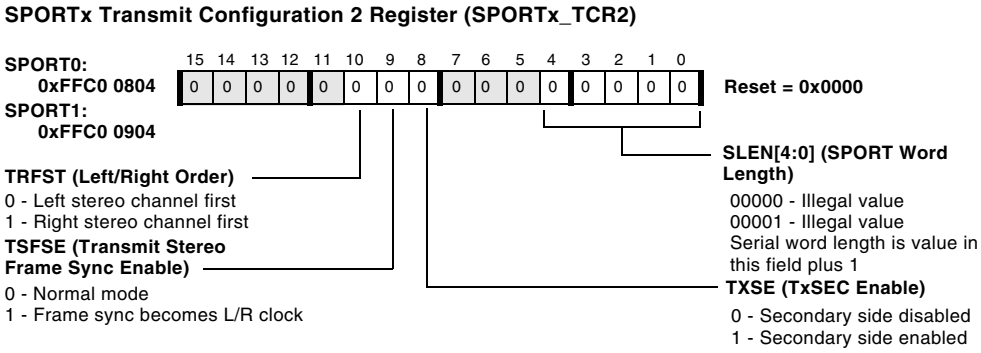


Figure 12-26. SPORTx Transmit Configuration 2 Register

Additional information for the `SPORTx_TCR1 and SPORTx_TCR2` transmit configuration register bits includes:

- **Transmit enable** (`TSPEN`). This bit selects whether the SPORT is enabled to transmit (if set) or disabled (if cleared).

  Setting `TSPEN` causes an immediate assertion of a SPORT TX interrupt, indicating that the TX data register is empty and needs to be filled. This is normally desirable because it allows centralization of the transmit data write code in the TX interrupt service routine (ISR). For this reason, the code should initialize the ISR and be ready to service TX interrupts before setting `TSPEN`.

  Similarly, if DMA transfers are used, DMA control should be configured correctly before setting `TSPEN`. Set all DMA control registers before setting `TSPEN`.

Clearing TSPEN causes the SPORT to stop driving data, TSCLK, and frame sync pins; it also shuts down the internal SPORT circuitry. In low power applications, battery life can be extended by clearing TSPEN whenever the SPORT is not in use.

All SPORT control registers should be programmed before TSPEN is set. Typical SPORT initialization code first writes all control registers, including DMA control if applicable. The last step in the code is to write SPORTx_TCR1 with all of the necessary bits, including TSPEN.

- **Internal transmit clock select**. (ITCLK). This bit selects the internal transmit clock (if set) or the external transmit clock on the TSCLK pin (if cleared). The TCLKDIV MMR value is not used when an external clock is selected.
- **Data formatting type select**. The two TDTYPE bits specify data formats used for single and multichannel operation.
- **Bit order select**. (TLSBIT). The TLSBIT bit selects the bit order of the data words transmitted over the SPORT.
- **Serial word length select**. (SLEN). The serial word length (the number of bits in each word transmitted over the SPORTs) is calculated by adding 1 to the value of the SLEN field:

```
Serial Word Length = SLEN + 1;
```

The SLEN field can be set to a value of 2 to 31; 0 and 1 are illegal values for this field. Three common settings for the SLEN field are 15, to transmit a full 16-bit word; 7, to transmit an 8-bit byte; and 23, to transmit a 24-bit word. The processor can load 16- or 32-bit values into the transmit buffer via DMA or an MMR write

instruction; the SLEN field tells the SPORT how many of those bits to shift out of the register over the serial link. The SPORT always transfers the SLEN+1 lower bits from the transmit buffer.

The frame sync signal is controlled by the SPORTx_TFSDIV and SPORTx_RFSDIV registers, not by SLEN. To produce a frame sync pulse on each byte or word transmitted, the proper frame sync divider must be programmed into the frame sync divider register; setting SLEN to 7 does not produce a frame sync pulse on each byte transmitted.

- **Internal transmit frame sync select**. (ITFS). This bit selects whether the SPORT uses an internal TFS (if set) or an external TFS (if cleared).
- **Transmit frame sync required select**. (TFSR). This bit selects whether the SPORT requires (if set) or does not require (if cleared) a transmit frame sync for every data word.

The TFSR bit is normally set during SPORT configuration. A frame sync pulse is used to mark the beginning of each word or data packet, and most systems need a frame sync to function properly.

- **Data-Independent transmit frame sync select**. (DITFS). This bit selects whether the SPORT generates a data-independent TFS (sync at selected interval) or a data-dependent TFS (sync when data is present in SPORTx_TX) for the case of internal frame sync select (ITFS = 1). The DITFS bit is ignored when external frame syncs are selected.

  The frame sync pulse marks the beginning of the data word. If DITFS is set, the frame sync pulse is issued on time, whether the SPORTx_TX register has been loaded or not; if DITFS is cleared, the frame sync pulse is only generated if the SPORTx_TX data register has been loaded. If the receiver demands regular frame sync pulses, DITFS should be set, and the processor should keep loading the

SPORTx_TX register on time. If the receiver can tolerate occasional late frame sync pulses, DITFS should be cleared to prevent the SPORT from transmitting old data twice or transmitting garbled data if the processor is late in loading the SPORTx_TX register.

- **Low transmit frame sync select**. (LTFS). This bit selects an active low TFS (if set) or active high TFS (if cleared).
- **Late transmit frame sync**. (LATFS). This bit configures late frame syncs (if set) or early frame syncs (if cleared).
- **Clock drive/sample edge select**. (TCKFE). This bit selects which edge of the TCLKx signal the SPORT uses for driving data, for driving internally generated frame syncs, and for sampling externally generated frame syncs. If set, data and internally generated frame syncs are driven on the falling edge, and externally generated frame syncs are sampled on the rising edge. If cleared, data and internally generated frame syncs are driven on the rising edge, and externally generated frame syncs are sampled on the falling edge.
- **TxSec enable**. (TXSE). This bit enables the transmit secondary side of the SPORT (if set).
- **Stereo serial enable**. (TSFSE). This bit enables the stereo serial operating mode of the SPORT (if set). By default this bit is cleared, enabling normal clocking and frame sync.
- **Left/Right order**. (TRFST). If this bit is set, the right channel is transmitted first in stereo serial operating mode. By default this bit is cleared, and the left channel is transmitted first.

SPORTx_RCR1 and SPORTx_RCR2 Registers

The main control registers for the receive portion of each SPORT are the receive configuration registers, SPORTx_RCR1 and SPORTx_RCR2, shown in Figure 12-27 and Figure 12-28.

A SPORT is enabled for receive if bit 0 (RSPEN) of the receive configuration 1 register is set to 1. This bit is cleared during either a hard reset or a soft reset, disabling all SPORT reception.

When the SPORT is enabled to receive (RSPEN set), corresponding SPORT configuration register writes are not allowed except for SPORTx_RCLKDIV and multichannel mode channel select registers. Writes to disallowed registers have no effect. While the SPORT is enabled, SPORTx_RCR1 is not written except for bit 0 (RSPEN). For example:

```
write (SPORTx_RCR1, 0x0001) ;  /* SPORT RX Enabled */
write (SPORTx_RCR1, 0xFF01) ;  /* ignored, no effect */
write (SPORTx_RCR1, 0xFFF0) ;  /* SPORT disabled, SPORTx_RCR1
still equal to 0x0000 */
```

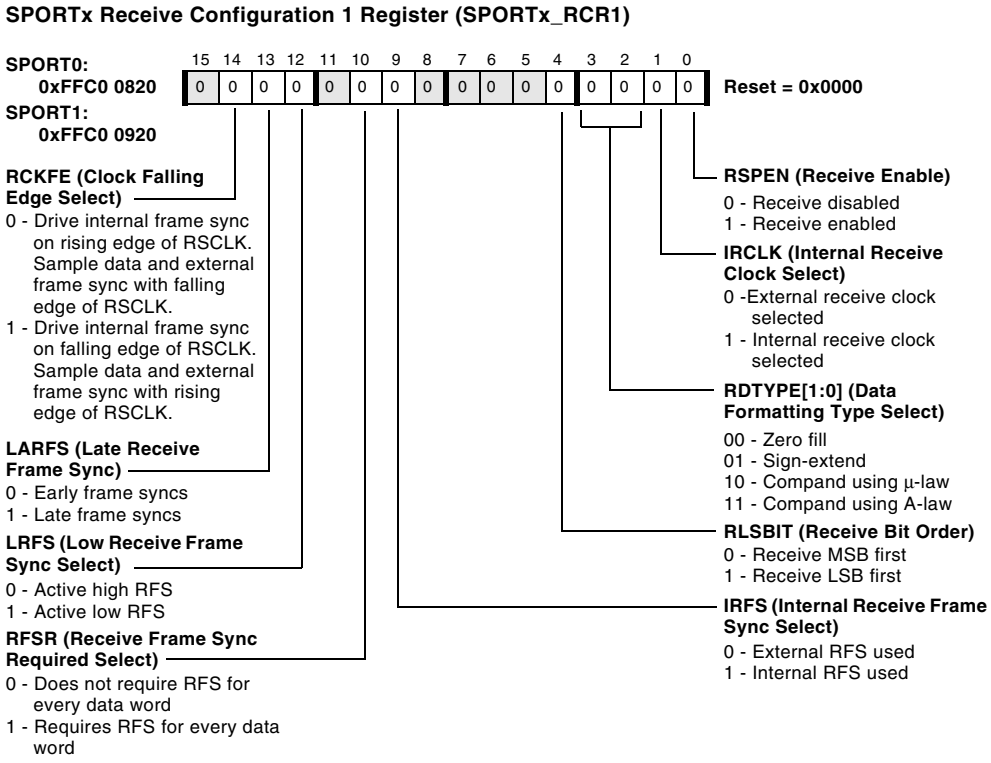


Figure 12-27. SPORTx Receive Configuration 1 Register

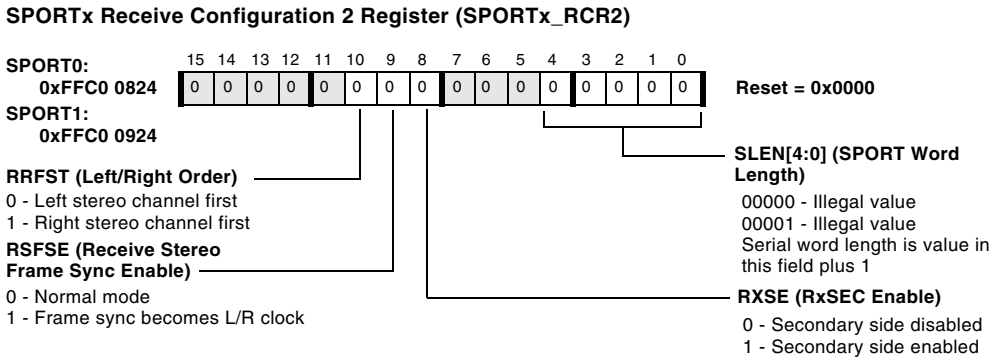


Figure 12-28. SPORTx Receive Configuration 2 Register

Additional information for the SPORTx_RCR1 and SPORTxRCR2 receive configuration register bits:

- **Receive enable**. (RSPEN). This bit selects whether the SPORT is enabled to receive (if set) or disabled (if cleared). Setting the RSPEN bit turns on the SPORT and causes it to sample data from the data receive pins as well as the receive bit clock and receive frame sync pins if so programmed.

  Setting RSPEN enables the SPORTx receiver, which can generate a SPORTx RX interrupt. For this reason, the code should initialize the ISR and the DMA control registers, and should be ready to service RX interrupts before setting RSPEN. Setting RSPEN also generates DMA requests if DMA is enabled and data is received. Set all DMA control registers before setting RSPEN.

  Clearing RSPEN causes the SPORT to stop receiving data; it also shuts down the internal SPORT receive circuitry. In low power applications, battery life can be extended by clearing RSPEN whenever the SPORT is not in use.

All SPORT control registers should be programmed before RSPEN is set. Typical SPORT initialization code first writes all control registers, including DMA control if applicable. The last step in the code is to write SPORTx_RCR1 with all of the necessary bits, including RSPEN.

- **Internal receive clock select**. (IRCLK). This bit selects the internal receive clock (if set) or external receive clock (if cleared). The RCLKDIV MMR value is not used when an external clock is selected.
- **Data formatting type select**. (RDTYPE). The two RDTYPE bits specify one of four data formats used for single and multichannel operation.
- **Bit order select**. (RLSBIT). The RLSBIT bit selects the bit order of the data words received over the SPORTs.
- **Serial word length select**. (SLEN). The serial word length (the number of bits in each word received over the SPORTs) is calculated by adding 1 to the value of the SLEN field. The SLEN field can be set to a value of 2 to 31; 0 and 1 are illegal values for this field.

The frame sync signal is controlled by the SPORTx_TFSDIV and SPORTx_RFSDIV registers, not by SLEN. To produce a frame sync pulse on each byte or word transmitted, the proper frame sync divider must be programmed into the frame sync divider register; setting SLEN to 7 does not produce a frame sync pulse on each byte transmitted.

- **Internal receive frame sync select**. (IRFS). This bit selects whether the SPORT uses an internal RFS (if set) or an external RFS (if cleared).
- **Receive frame sync required select**. (RFSR). This bit selects whether the SPORT requires (if set) or does not require (if cleared) a receive frame sync for every data word.

- **Low receive frame sync select**. (LRFS). This bit selects an active low RFS (if set) or active high RFS (if cleared).
- **Late receive frame sync**. (LARFS). This bit configures late frame syncs (if set) or early frame syncs (if cleared).
- **Clock drive/sample edge select**. (RCKFE). This bit selects which edge of the RSCLK clock signal the SPORT uses for sampling data, for sampling externally generated frame syncs, and for driving internally generated frame syncs. If set, internally generated frame syncs are driven on the falling edge, and data and externally generated frame syncs are sampled on the rising edge. If cleared, internally generated frame syncs are driven on the rising edge, and data and externally generated frame syncs are sampled on the falling edge.
- **RxSec enable**. (RXSE). This bit enables the receive secondary side of the SPORT (if set).
- **Stereo serial enable**. (RSFSE). This bit enables the stereo serial operating mode of the SPORT (if set). By default this bit is cleared, enabling normal clocking and frame sync.
- **Left/Right order**. (RRFST). If this bit is set, the right channel is received first in stereo serial operating mode. By default this bit is cleared, and the left channel is received first.

Data Word Formats

The format of the data words transferred over the SPORTs is configured by the combination of transmit SLEN and receive SLEN; RDTYPE; TDTYPE; RLSBIT; and TLSBIT bits of the SPORTx_TCR1, SPORTx_TCR2, SPORTx_RCR1, and SPORTx_RCR2 registers.

SPORTx_TX Register

The SPORTx transmit data register (SPORTx_TX) is a write-only register. Reads produce a Peripheral Access Bus (PAB) error. Writes to this register cause writes into the transmitter FIFO. The 16-bit wide FIFO is 8 deep for word length <= 16 and 4 deep for word length > 16. The FIFO is common to both primary and secondary data and stores data for both. Data ordering in the FIFO is shown in the Figure 12-29. The SPORTx_TX register is shown in Figure 12-30.

ONLY PRIMARY ENABLED
DATA LENGTH <= 16 BITS

8 WORDS OF
PRIMARY DATA
IN FIFO

15	0
PRIMARY	W7
PRIMARY	W6
PRIMARY	W5
PRIMARY	W4
PRIMARY	W3
PRIMARY	W2
PRIMARY	W1
PRIMARY	W0

PRIMARY AND
SECONDARY ENABLED
DATA LENGTH <= 16 BITS

4 WORDS OF
PRIMARY DATA AND
4 WORDS OF
SECONDARY DATA
IN FIFO

15	0
SECONDARY	W3
PRIMARY	W3
SECONDARY	W2
PRIMARY	W2
SECONDARY	W1
PRIMARY	W1
SECONDARY	W0
PRIMARY	W0

ONLY PRIMARY ENABLED
DATA LENGTH > 16 BITS

4 WORDS OF
PRIMARY DATA
IN FIFO

15	0
PRIMARY	W3 LOW
PRIMARY	W3 HIGH
PRIMARY	W2 LOW
PRIMARY	W2 HIGH
PRIMARY	W1 LOW
PRIMARY	W1 HIGH
PRIMARY	W0 LOW
PRIMARY	W0 HIGH

PRIMARY AND
SECONDARY ENABLED
DATA LENGTH > 16 BITS

2 WORDS OF
PRIMARY DATA AND
2 WORDS OF
SECONDARY DATA
IN FIFO

15	0
SECONDARY	W1 LOW
SECONDARY	W1 HIGH
PRIMARY	W1 LOW
PRIMARY	W1 HIGH
SECONDARY	W0 LOW
SECONDARY	W0 HIGH
PRIMARY	W0 LOW
PRIMARY	W0 HIGH

Figure 12-29. SPORT Transmit FIFO Data Ordering

It is important to keep the interleaving of primary and secondary data in the FIFO as shown. This means that PAB/DMA writes to the FIFO must follow an order of primary first, and then secondary, if secondary is enabled. DAB/PAB writes must match their size to the data word length. For word length up to and including 16 bits, use a 16-bit write. Use a 32-bit write for word length greater than 16 bits.

When transmit is enabled, data from the FIFO is assembled in the TX Hold register based on TXSE and SLEN, and then shifted into the primary and secondary shift registers. From here, the data is shifted out serially on the DTPRI and DTSEC pins.

The SPORT TX interrupt is asserted when TSPEN = 1 and the TX FIFO has room for additional words. This interrupt does not occur if SPORT DMA is enabled. For DMA operation, see Chapter 5, "Direct Memory Access".

The transmit underflow status bit (TUVF) is set in the SPORT status register when a transmit frame sync occurs and no new data has been loaded into the serial shift register. In multichannel mode (MCM), TUVF is set whenever the serial shift register is not loaded, and transmission begins on the current enabled channel. The TUVF status bit is a sticky write-1-to-clear (W1C) bit and is also cleared by disabling the SPORT (writing TXEN = 0).

If software causes the core processor to attempt a write to a full TX FIFO with a SPORTx_TX write, the new data is lost and no overwrites occur to data in the FIFO. The TOVF status bit is set and a SPORT error interrupt is asserted. The TOVF bit is a sticky bit; it is only cleared by disabling the SPORT TX. To find out whether the core processor can access the SPORTx_TX register without causing this type of error, read the register's status first. The TXF bit in the SPORT status register is 0 if space is available for another word in the FIFO.

The TXF and TOVF status bits in the SPORTx status register are updated upon writes from the core processor, even when the SPORT is disabled.

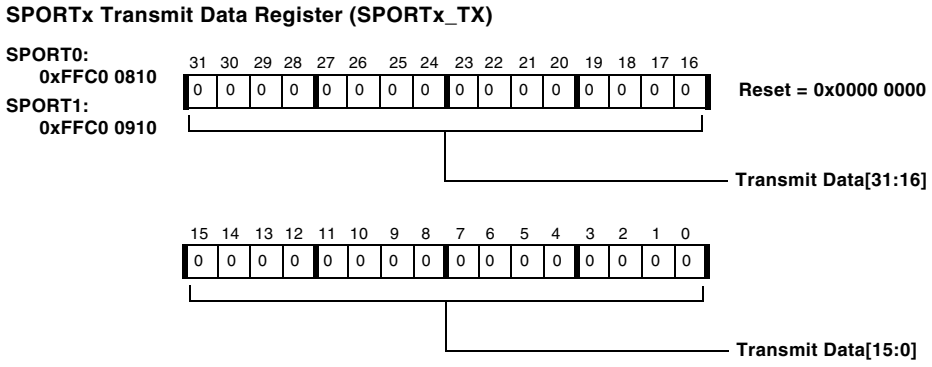


Figure 12-30. SPORTx Transmit Data Register

SPORTx_RX Register

The SPORTx receive data register (SPORTx_RX) is a read-only register. Writes produce a PAB error. The same location is read for both primary and secondary data. Reading from this register space causes reading of the receive FIFO. This 16-bit FIFO is 8 deep for receive word length <= 16 and 4 deep for length > 16 bits. The FIFO is shared by both primary and secondary receive data. The order for reading using PAB/DMA reads is important since data is stored in differently depending on the setting of the SLEN and RXSE configuration bits.

Data storage and data ordering in the FIFO are shown in Figure 12-31. The SPORTx_RX register is shown in Figure 12-32.

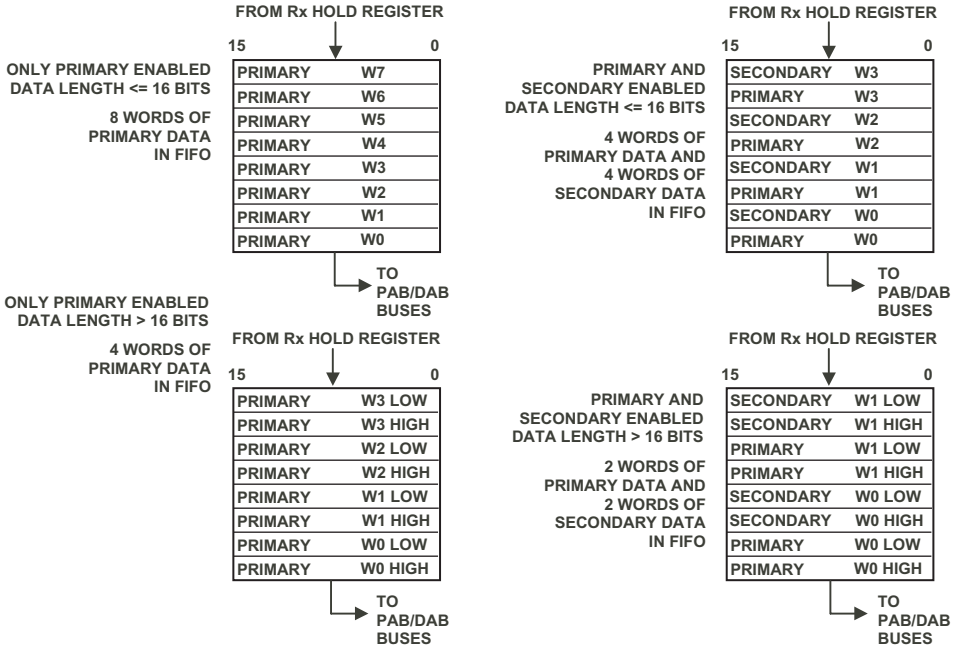


Figure 12-31. SPORT Receive FIFO Data Ordering

When reading from the FIFO for both primary and secondary data, read primary first, followed by secondary. DAB/PAB reads must match their size to the data word length. For word length up to and including 16 bits, use a 16-bit read. Use a 32-bit read for word length greater than 16 bits.

When receiving is enabled, data from the DRPRI pin is loaded into the RX primary shift register, while data from the DRSEC pin is loaded into the RX secondary shift register. At transfer completion of a word, data is shifted

into the RX hold registers for primary and secondary data, respectively. Data from the hold registers is moved into the FIFO based on RXSE and SLEN.

The SPORT RX interrupt is generated when RSPEN = 1 and the RX FIFO has received words in it. When the core processor has read all the words in the FIFO, the RX interrupt is cleared. The SPORT RX interrupt is set only if SPORT RX DMA is disabled; otherwise, the FIFO is read by DMA reads.

If the program causes the core processor to attempt a read from an empty RX FIFO, old data is read, the RUVF flag is set in the SPORTx_STAT register, and the SPORT error interrupt is asserted. The RUVF bit is a sticky bit and is cleared only when the SPORT is disabled. To determine if the core can access the RX registers without causing this error, first read the RX FIFO status (RXNE in the SPORTx status register). The RUVF status bit is updated even when the SPORT is disabled.

The ROVF status bit is set in the SPORTx_STAT register when a new word is assembled in the RX shift register and the RX hold register has not moved the data to the FIFO. The previously written word in the hold register is overwritten. The ROVF bit is a sticky bit; it is only cleared by disabling the SPORT RX.

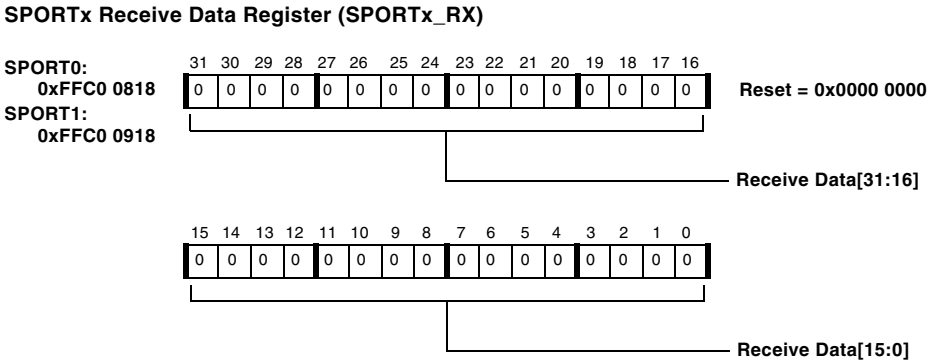


Figure 12-32. SPORTx Receive Data Register

SPORTx_STAT Register

The SPORT status register (SPORTx_STAT) is used to determine if the access to a SPORT RX or TX FIFO can be made by determining their full or empty status. This register is shown in Figure 12-33.

The TXF bit in the SPORT status register indicates whether there is room in the TX FIFO. The RXNE status bit indicates whether there are words in the RX FIFO. The TXHRE bit indicates if the TX hold register is empty.

The transmit underflow status bit (TUVF) is set whenever the TFS signal occurs (from either an external or internal source) while the TX shift register is empty. The internally generated TFS may be suppressed whenever SPORTx_TX is empty by clearing the DITFS control bit in the SPORT configuration register. The TUVF status bit is a sticky write-1-to-clear (W1C) bit and is also cleared by disabling the SPORT (writing TXEN = 0).

For continuous transmission (TFSR = 0), TUVF is set at the end of a transmitted word if no new word is available in the TX hold register.

The TOVF bit is set when a word is written to the TX FIFO when it is full. It is a sticky W1C bit and is also cleared by writing TXEN = 0. Both TXF and TOVF are updated even when the SPORT is disabled.

When the SPORT RX hold register is full, and a new receive word is received in the shift register, the receive overflow status bit (ROVF) is set in the SPORT status register. It is a sticky W1C bit and is also cleared by disabling the SPORT (writing RXEN = 0).

The RUVF bit is set when a read is attempted from the RX FIFO and it is empty. It is a sticky W1C bit and is also cleared by writing RXEN = 0. The RUVF bit is updated even when the SPORT is disabled.

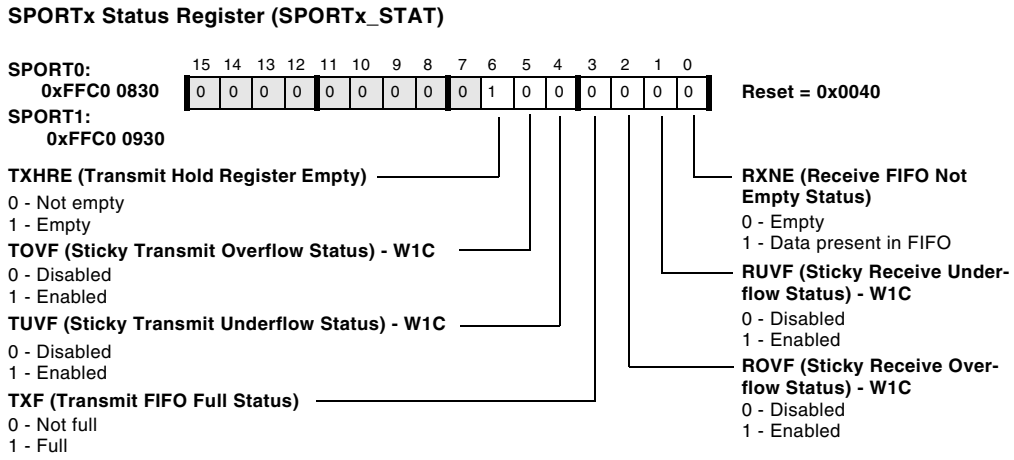


Figure 12-33. SPORTx Status Register

SPORTx_TCLKDIV and SPORTx_RCLKDIV Registers

The frequency of an internally generated clock is a function of the system clock frequency (as seen at the SCLK pin) and the value of the 16-bit serial clock divide modulus registers (the SPORTx transmit serial clock divider register, SPORTx_TCLKDIV, shown in Figure 12-34, and the SPORTx receive serial clock divider register, SPORTx_RCLKDIV, shown in Figure 12-35).

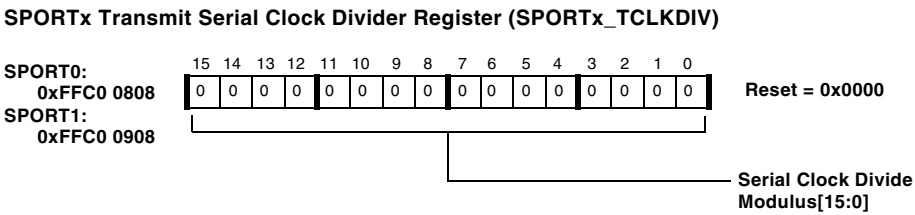


Figure 12-34. SPORTx Transmit Serial Clock Divider Register

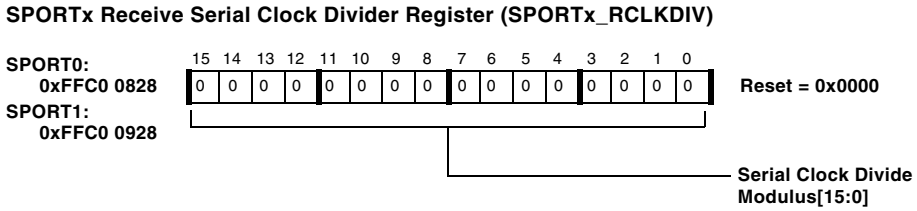


Figure 12-35. SPORTx Receive Serial Clock Divider Register

SPORTx_TFSDIV and SPORTx_RFSDIV Register

The 16-bit SPORTx transmit frame sync divider register (`SPORTx_TFSDIV`) and the SPORTx receive frame sync divider register (`SPORTx_RFSDIV`) specify how many transmit or receive clock cycles are counted before generating a `TFS` or `RFS` pulse when the frame sync is internally generated. In this way, a frame sync can be used to initiate periodic transfers. The counting of serial clock cycles applies to either internally or externally generated serial clocks. These registers are shown in Figure 12-36 and Figure 12-37.

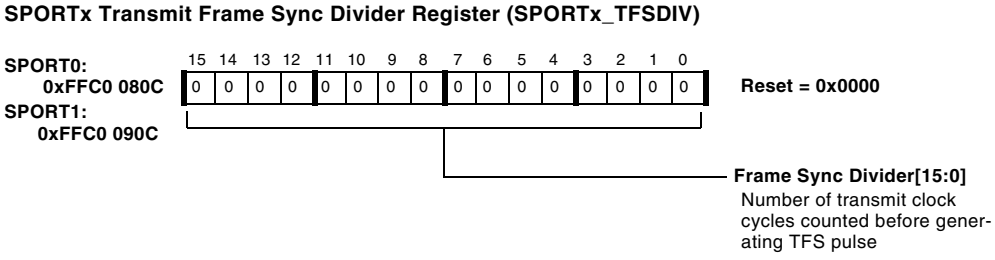


Figure 12-36. SPORTx Transmit Frame Sync Divider Register

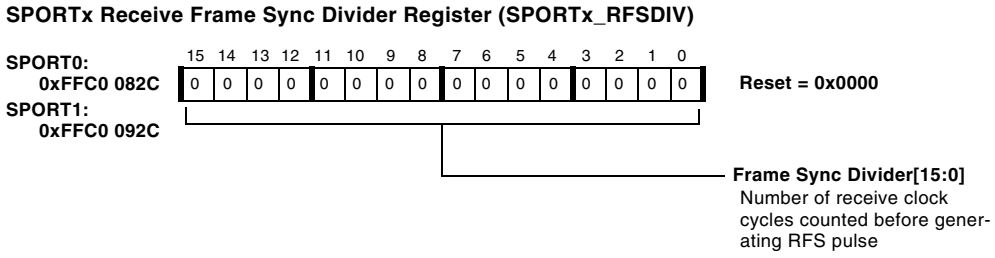


Figure 12-37. SPORTx Receive Frame Sync Divider Register

SPORTx_MCMCn Registers

There are two SPORTx multichannel configuration registers (`SPORTx_MCMCn`) for each SPORT, shown in Figure 12-38 and Figure 12-39. The `SPORTx_MCMCn` registers are used to configure the multichannel operation of the SPORT. The two control registers are shown below.

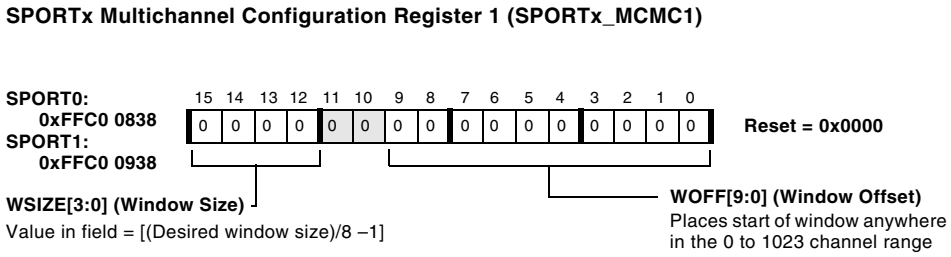


Figure 12-38. SPORTx Multichannel Configuration Register 1

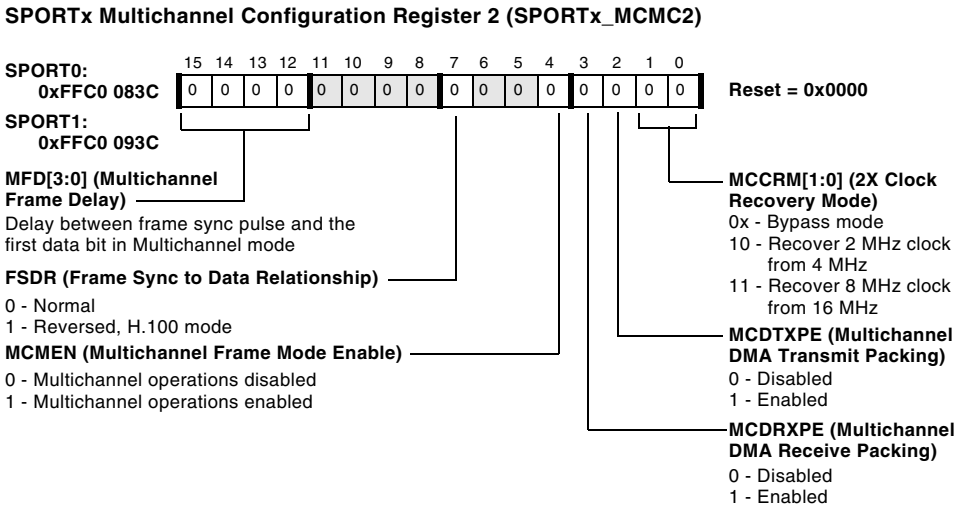


Figure 12-39. SPORTx Multichannel Configuration Register 2

SPORTx_CHNL Register

The 10-bit `CHNL` field in the SPORTx current channel register (`SPORTx_CHNL`) indicates which channel is currently being serviced during multichannel operation. This field is a read-only status indicator. The `CHNL[9:0]` field increments by one as each channel is serviced. The counter stops at the upper end of the defined window. The channel select register restarts at 0 at each frame sync. As an example, for a window size of 8 and an offset of 148, the counter displays a value between 0 and 156.

Once the window size has completed, the channel counter resets to 0 in preparation for the next frame. Because there are synchronization delays between RSCLK and the processor clock, the channel register value is approximate. It is never ahead of the channel being served, but it may lag behind. See Figure 12-40.

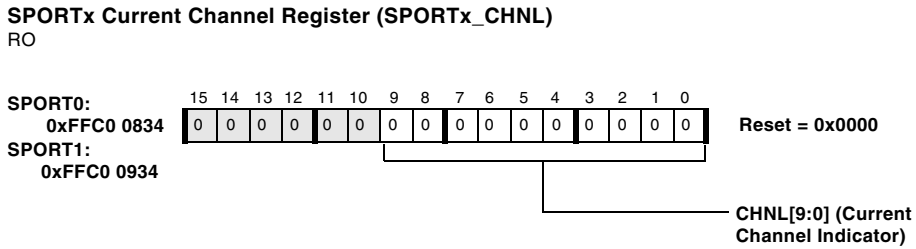


Figure 12-40. SPORTx Current Channel Register

SPORTx_MRCSn Registers

The multichannel selection registers are used to enable and disable individual channels. The SPORTx multichannel receive select registers (SPORTx_MRCSn, shown in Figure 12-41) specify the active receive channels. There are four registers, each with 32 bits, corresponding to the 128 channels. Setting a bit enables that channel so that the SPORT selects that word for receive from the multiple word block of data. For example, setting bit 0 selects word 0, setting bit 12 selects word 12, and so on.

Setting a particular bit in the SPORTx_MRCSn register causes the SPORT to receive the word in that channel's position of the datastream; the received word is loaded into the RX buffer. When the secondary receive side is

enabled by the RXSE bit, both inputs are processed on enabled channels. Clearing the bit in the SPORTx_MRCSn register causes the SPORT to ignore the data on either channel.

SPORTx Multichannel Receive Select Registers (SPORTx_MRCSn)

For all bits, 0 - Channel disabled, 1 - Channel enabled, so SPORT selects that word from multiple word block of data.

For Memory-mapped addresses, see Table 12-6.

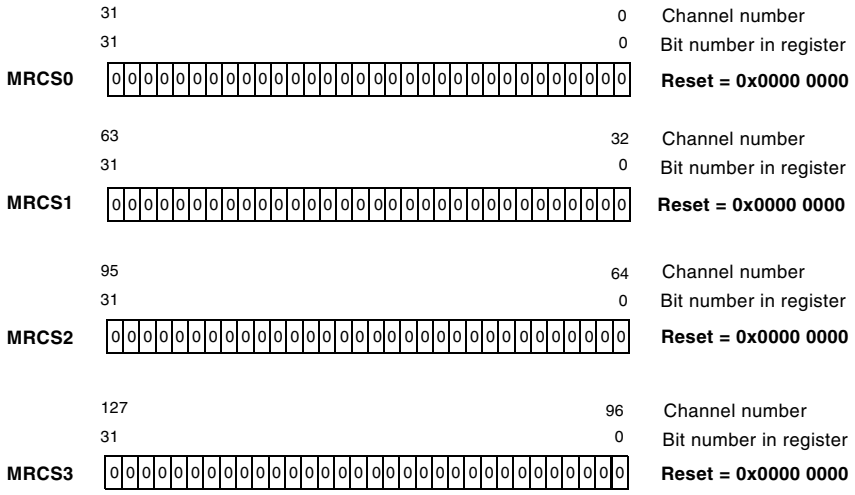


Figure 12-41. SPORTx Multichannel Receive Select Registers

Table 12-6. SPORTx Multichannel Receive Select Register Memory-mapped Addresses

Register Name	Memory-mapped Address
SPORT0_MRCS0	0xFFC0 0850
SPORT0_MRCS1	0xFFC0 0854
SPORT0_MRCS2	0xFFC0 0858
SPORT0_MRCS3	0xFFC0 085C
SPORT1_MRCS0	0xFFC0 0950
SPORT1_MRCS1	0xFFC0 0954
SPORT1_MRCS2	0xFFC0 0958
SPORT1_MRCS3	0xFFC0 095C

SPORTx_MTCSn Registers

The multichannel selection registers are used to enable and disable individual channels. The four SPORTx multichannel transmit select registers (SPORTx_MTCSn, Figure 12-42) specify the active transmit channels. There are four registers, each with 32 bits, corresponding to the 128 channels. Setting a bit enables that channel so that the SPORT selects that word for transmit from the multiple word block of data. For example, setting bit 0 selects word 0, setting bit 12 selects word 12, and so on.

Setting a particular bit in a SPORTx_MTCSn register causes the SPORT to transmit the word in that channel's position of the datastream. When the secondary transmit side is enabled by the TXSE bit, both sides transmit a

word on the enabled channel. Clearing the bit in the `SPORTx_MTCSn` register causes both SPORT controllers' data transmit pins to three-state during the time slot of that channel.

SPORTx Multichannel Transmit Select Registers (SPORTx_MTCSn)

For all bits, 0 - Channel disabled, 1 - Channel enabled, so SPORT selects that word from multiple word block of data.

For Memory-mapped addresses, see Table 12-7.

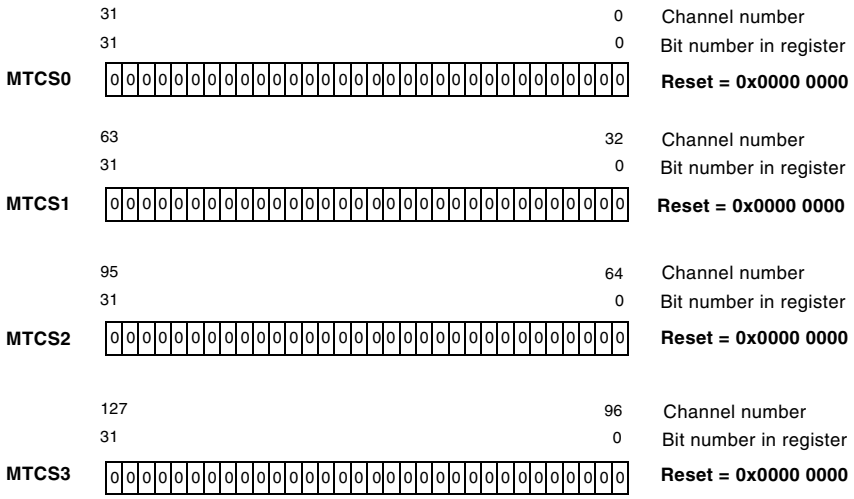


Figure 12-42. SPORTx Multichannel Transmit Select Registers

Table 12-7. SPORTx Multichannel Transmit Select Register Memory-mapped Addresses

Register Name	Memory-mapped Address
SPORT0_MTCS0	0xFFC0 0840
SPORT0_MTCS1	0xFFC0 0844
SPORT0_MTCS2	0xFFC0 0848
SPORT0_MTCS3	0xFFC0 084C
SPORT1_MTCS0	0xFFC0 0940
SPORT1_MTCS1	0xFFC0 0944
SPORT1_MTCS2	0xFFC0 0948
SPORT1_MTCS3	0xFFC0 094C

Programming Examples

This section shows an example of typical usage of the SPORT peripheral in conjunction with the DMA controller. See Listing 12-1 through Listing 12-4 on page 12-78.

The SPORT is usually employed for high-speed, continuous serial transfers. The example reflects this, in that the SPORT is set-up for auto-buffered, repeated DMA transfers.

Because of the many possible configurations, the example uses generic labels for the content of the SPORT's configuration registers (`SPORTx_RCRx` and `SPORTx_TCRx`) and the DMA configuration. An example value is given in the comments, but for the meaning of the individual bits the user is referred to the detailed explanation in this chapter.

The example configures both the receive and the transmit section. Since they are completely independent, the code uses separate labels.

SPORT Initialization Sequence

The SPORT's receiver and transmitter are configured, but they are not enabled yet.

Listing 12-1. SPORT Initialization

```
Program_SPORT_TRANSMITTER_Registers:
  /* Set P0 to SPORT0 Base Address */
  P0.h = hi(SPORT0_TCR1);
  P0.l = lo(SPORT0_TCR1);

/* Configure Clock speeds */
  R1 = SPORT_TCLK_CONFIG;  /* Divider SCLK/TCLK (value 0 to
65535) */
  W[P0 + (SPORT0_TCLKDIV - SPORT0_TCR1)] = R1;   /*  TCK divider
register */
  /* number of Bitclocks between FrameSyncs -1  (value SPORT_SLEN
to 65535) */
  R1 = SPORT_TFSDIV_CONFIG;
  W[P0 + (SPORT0_TFSDIV - SPORT0_TCR1)] = R1;   /*  TFSDIV regis-
ter */

/* Transmit configuration */
  /*  Configuration register 2   (for instance 0x000E for 16-bit
wordlength) */
  R1 = SPORT_TRANSMIT_CONF_2;
  W[P0 + (SPORT0_TCR2 - SPORT0_TCR1)] = R1;
  /*  Configuration register 1   (for instance 0x4E12 for inter-
nally generated clk and framesync) */
  R1 = SPORT_TRANSMIT_CONF_1;
  W[P0] = R1;
  ssync;  /*  NOTE: SPORT0 TX NOT enabled yet (bit 0 of TCR1 must
be zero) */
```

```
Program_SPORT_RECEIVER_Registers:
  /* Set P0 to SPORT0 Base Address */
  P0.h = hi(SPORT0_RCR1);
  P0.l = lo(SPORT0_RCR1);

/* Configure Clock speeds */
  R1 = SPORT_RCLK_CONFIG;  /* Divider SCLK/RCLK (value 0 to
65535) */
  W[P0 + (SPORT0_RCLKDIV - SPORT0_RCR1)] = R1;   /*  RCK divider
register */
  /* number of Bitclock between FrameSyncs -1  (value SPORT_SLEN
to 65535) */
  R1 = SPORT_RFSDIV_CONFIG;
  W[P0 + (SPORT0_RFSDIV - SPORT0_RCR1)] = R1;  /*  RFSDIV regis-
ter */

/* Receive configuration */
  /*  Configuration register 2   (for instance 0x000E for 16-bit
wordlength) */
  R1 = SPORT_RECEIVE_CONF_2;
  W[P0 + (SPORT0_RCR2 - SPORT0_RCR1)] = R1;
  /*  Configuration register 1   (for instance 0x4410 for exter-
nal clk and framesync) */
  R1 = SPORT_RECEIVE_CONF_1;
  W[P0] = R1;
  ssync;  /*  NOTE: SPORT0 RX NOT enabled yet (bit 0 of RCR1 must
be zero) */
```

DMA Initialization Sequence

Next the DMA channels for receive (channel3) and for transmit (channel4) are set up for auto-buffered, one-dimensional, 32-bit transfers. Again, there are other possibilities, so generic labels have been used, with a particular value shown in the comments. See Chapter 5, "Direct Memory Access", for a detailed explanation of the bits.

Note that the DMA channels can be enabled at the end of the configuration since the SPORT is not enabled yet. However, if preferred, the user can enable the DMA later, immediately before enabling the SPORT. The only requirement is that the DMA channel be enabled before the associated peripheral is enabled to start the transfer.

Listing 12-2. DMA Initialization

```
Program_DMA_Controller:

/* Receiver (DMA channel 3) */
  /* Set P0 to DMA Base Address */
  P0.l = lo(DMA3_CONFIG);
  P0.h = hi(DMA3_CONFIG);

  /* Configuration (for instance 0x108A for Autobuffer, 32-bit
wide transfers) */
  R0 = DMA_RECEIVE_CONF(z);
  W[P0] = R0;  /* configuration register */

  /* rx_buf = Buffer in Data memory (divide count by four because
of 32-bit DMA transfers) */
  R1 = (length(rx_buf)/4)(z);
  W[P0 + (DMA3_X_COUNT - DMA3_CONFIG)] = R1;  /* X_count register
*/
  R1 = 4(z);  /* 4 bytes in a 32-bit transfer */
```

```
  W[P0 + (DMA3_X_MODIFY - DMA3_CONFIG)] = R1;  /* X_modify regis-
ter */

  /* start_address register points to memory buffer to be filled
*/
  R1.l = rx_buf;
  R1.h = rx_buf;
  [P0 + (DMA3_START_ADDR - DMA3_CONFIG)] = R1;

  BITSET(R0,0);  /* R0 still contains value of CONFIG register -
set bit 0 */
  W[P0] = R0;  /* enable DMA channel (SPORT not enabled yet) */

/* Transmitter (DMA channel 4) */
  /* Set P0 to DMA Base Address */
  P0.l = lo(DMA4_CONFIG);
  P0.h = hi(DMA4_CONFIG);
  /* Configuration (for instance 0x1088 for Autobuffer, 32-bit
wide transfers) */
  R0 = DMA_TRANSMIT_CONF(z);
  W[P0] = R0;  /* configuration register */

  /* tx_buf = Buffer in Data memory (divide count by four because
of 32-bit DMA transfers) */
  R1 = (length(tx_buf)/4)(z);
  W[P0 + (DMA4_X_COUNT - DMA4_CONFIG)] = R1;  /* X_count register
*/
  R1 = 4(z);  /* 4 bytes in a 32-bit transfer */
  W[P0 + (DMA4_X_MODIFY - DMA4_CONFIG)] = R1;  /* X_modify regis-
ter */

  /* start_address register points to memory buffer to be trans-
mitted from */
  R1.l = tx_buf;
```

```
  R1.h = tx_buf;
  [P0 + (DMA4_START_ADDR - DMA4_CONFIG)] = R1;

  BITSET(R0,0);  /* R0 still contains value of CONFIG register -
set bit 0 */
  W[P0] = R0;  /* enable DMA channel (SPORT not enabled yet) */
```

Interrupt Servicing

The receive channel and the transmit channel will each generate an interrupt request if so programmed. The following code fragments show the minimum actions that must be taken. Not shown is the programming of the core and system event controllers.

Listing 12-3. Servicing an Interrupt

```
RECEIVE_ISR:
  [--SP] = RETI;  /* nesting of interrupts */

  /* clear DMA interrupt request */
  P0.h = hi(DMA3_IRQ_STATUS);
  P0.l = lo(DMA3_IRQ_STATUS);
  R1   = 1;
  W[P0] = R1.l;  /* write one to clear  */

  RETI = [SP++];
  rti;

TRANSMIT_ISR:
  [--SP] = RETI;  /* nesting of interrupts */

  /* clear DMA interrupt request */
  P0.h = hi(DMA4_IRQ_STATUS);
  P0.l = lo(DMA4_IRQ_STATUS);
```

```
  R1   = 1;
  W[P0] = R1.l;  /* write one to clear  */

  RETI = [SP++];
  rti;
```

Starting a Transfer

After the initialization procedure outlined in the previous sections, the receiver and transmitter are enabled. The core may just wait for interrupts.

Listing 12-4. Starting a Transfer

```
/* Enable Sport0 RX and TX */
  P0.h = hi(SPORT0_RCR1);
  P0.l = lo(SPORT0_RCR1);
  R1 = W[P0](Z);
  BITSET(R1,0);
  W[P0] = R1;
  ssync;  /*  Enable Receiver (set bit 0) */

  P0.h = hi(SPORT0_TCR1);
  P0.l = lo(SPORT0_TCR1);
  R1 = W[P0](Z);
  BITSET(R1,0);
  W[P0] = R1;
  ssync;  /*  Enable Transmitter (set bit 0) */


/* dummy wait loop (do nothing but waiting for interrupts) */
wait_forever:
  jump wait_forever;
```

13 UART PORT CONTROLLERS

This chapter describes the Universal Asynchronous Receiver/Transmitter (UART) modules. Following an overview and a list of key features is a description of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

Overview

The processor features two separate and identical UART modules.

The UART modules are full-duplex peripherals compatible with PC-style industry-standard UARTs, sometimes called Serial Controller Interfaces (SCI). The UARTs convert data between serial and parallel formats. The serial communication follows an asynchronous protocol that supports various word length, stop bits, bit rate, and parity generation options.

Features

Each UART includes these features:

- 5 – 8 data bits
- 1 or 2 stop bits (1 1/2 in 5-bit mode)
- Even, odd, and sticky parity bit options
- 3 interrupt outputs for reception, transmission, and status
- Independent DMA operation for receive and transmit

- SIR IrDA operation mode
- Internal loop back

The UARTs are logically compliant to EIA-232E, EIA-422, EIA-485 and LIN standards, but usually require external transceiver devices to meet electrical requirements. In IrDA® (Infrared Data Association) mode, the UARTs meet the half-duplex IrDA SIR (9.6/115.2 Kbps rate) protocol.

Interface Overview

Figure 13-1 shows a simplified block diagram of one UARTx module and how it interconnects to the Blackfin architecture and to the outside world.

External Interface

Each UART features an RX and a TX pin available through port F. These two pins usually connect to an external transceiver device that meets the electrical requirements of full duplex (for example, EIA-232, EIA-422, 4-wire EIA-485) or half duplex (for example, 2-wire EIA-485, LIN) standards. While the UART0 signals are multiplexed with DMA request inputs, the UART1 signals compete with timer 6 and timer 7. To connect UART signals to the respective pins, clear the `PFDE` and `PFTE` bits in the `PORT_MUX` register, and enable the alternate function in the `PORTF_FER` register. If only the receive or only the transmit channel of a UART module is used, the unused pin can still function as a General-Purpose I/O (GPIO).

Modem status and control functionality is not supported by the UART modules, but may be implemented using GPIO pins.

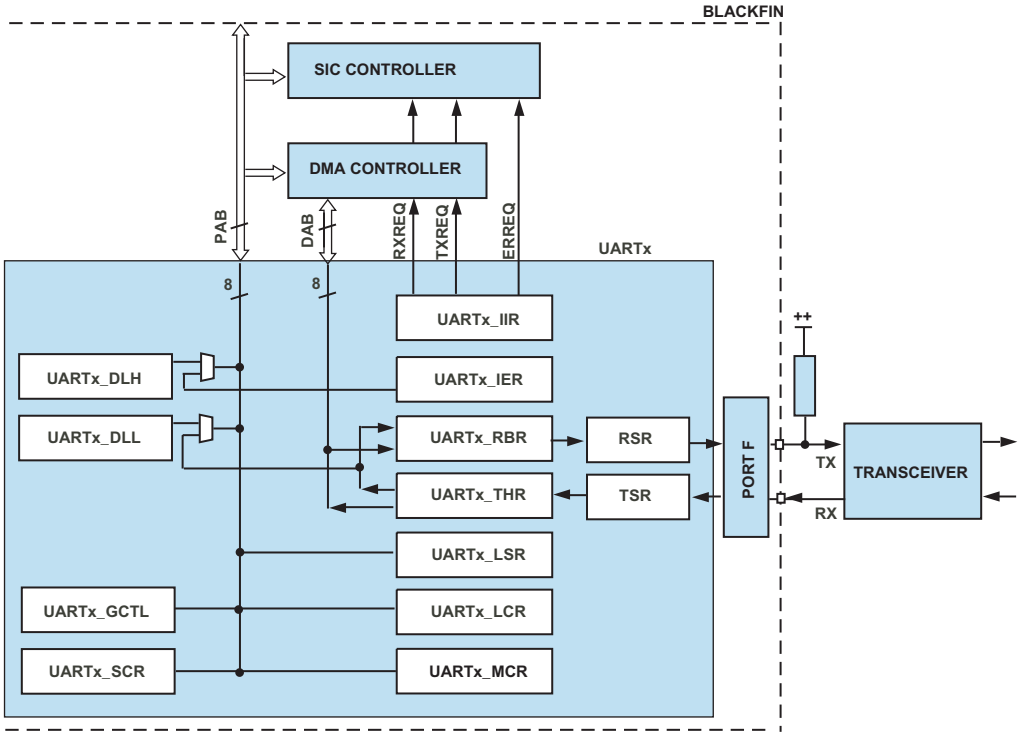


Figure 13-1. UART Block Diagram

Internal Interface

The UARTs are DMA-capable peripherals with support for separate TX and RX DMA master channels. They can be used in either DMA or programmed non-DMA mode of operation. The non-DMA mode requires software management of the data flow using either interrupts or polling. The DMA method requires minimal software intervention as the DMA engine itself moves the data. Each UART has its own separate transmit and receive DMA channels. For more information on DMA, see Chapter 5, “Direct Memory Access”.

All UART registers are 8 bits wide. They connect to the PAB bus. However, some registers share their address as controlled by the DLAB bit in the UARTx_LCR register. The UARTx_RBR and UARTx_THR registers also connect to the DAB bus

Timer 1 provides a hardware assisted autobaud detection mechanism for use with UART 0. Similarly, timer 6 supports UART 1. See Chapter 15, "General-Purpose Timers", for more information.

Description of Operation

The following sections describe the operation of the UART.

UART Transfer Protocol

UART communication follows an asynchronous serial protocol, consisting of individual data words. A word has 5 to 8 data bits.

All data words require a start bit and at least one stop bit. With the optional parity bit, this creates a 7- to 12-bit range for each word. The format of received and transmitted character frames is controlled by the line control register (UARTx_LCR). Data is always transmitted and received least significant bit (LSB) first.

Figure 13-2 shows a typical physical bitstream measured on one of the TX pins.

Aside from the standard UART functionality, the UART also supports half-duplex serial data communication via infrared signals, according to the recommendations of the Infrared Data Association (IrDA). The physical layer known as IrDA SIR (9.6/115.2 Kbps rate) is based on return-to-zero-inverted (RZI) modulation. Pulse position modulation is not supported.

Using the 16x data rate clock, RZI modulation is achieved by inverting and modulating the non-return-to-zero (NRZ) code normally transmitted by the UART. On the receive side, the 16x clock is used to determine an IrDA pulse sample window, from which the RZI-modulated NRZ code is recovered.

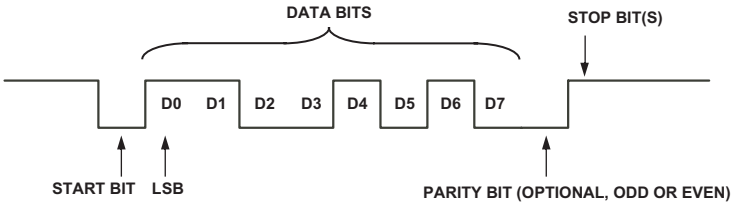


Figure 13-2. Bitstream on a TX Pin Transmitting an “S” Character (0x53)

IrDA support is enabled by setting the `IREN` bit in the `UARTx_GCTL` register. The IrDA application requires external transceivers.

UART Transmit Operation

Receive and transmit paths operate completely independently except that the bit rate and the frame format are identical for both transfer directions.

Transmission is initiated by writes to the `UARTx_THR` register. If no former operation is pending, the data is immediately passed from the `UARTx_THR` register to the internal `TSR` register where it is shifted out at a bit rate equal to `SCLK/(16 × Divisor)` with start, stop, and parity bits appended as defined the `UARTx_LCR` register. The least significant bit (LSB) is always transmitted first. This is bit 0 of the value written to `UARTx_THR`.

Writes to the `UARTx_THR` register clear the `THRE` flag. Transfers of data from `UARTx_THR` to the transmit shift registers (`TSR`) set this status flag in `UARTx_LSR` again.

When enabled by the ETBEI bit in the UARTx_IER register, a 0 to 1 transition of the THRE flag requests an interrupt on the dedicated TXREQ output. This signal is routed through the DMA controller. If the associated DMA channel is enabled, the TXREQ signal functions as a DMA request, otherwise the DMA controller simply forwards it to the SIC interrupt controller.

The UARTx_THR register and the internal TSR register can be seen as a two-stage transmit buffer. When data is pending in either one of these registers, the TEMT flag is low. As soon as the data has left the TSR register, the TEMT bit goes high again and indicates that all pending transmit operation has finished. At that time it is safe to disable the UCEN bit or to three-state off-chip line drivers.

UART Receive Operation

The receive operation uses the same data format as the transmit configuration, except that one valid stop bit is always sufficient, that is, the STB bit has no impact to the receiver.

After detection of the start bit, the received word is shifted into the internal shift register (RSR) at a bit rate of SCLK/(16 x Divisor). Once the appropriate number of bits (including one stop bit) is received, the content of the RSR register is transferred to the UARTx_RBR registers, shown in Figure 13-11. Finally, the data ready (DR) bit and the status flags are updated in the UARTx_LSR register, to signal data reception, parity, and also error conditions, if required.

The RSR and the UARTx_RBR registers can be seen as a two-stage receive buffer. If more than 2 bytes are received before software reads the UARTx_RBR register, an overrun error is reported and old data is overwritten.

If enabled by the ERBFI bit in the UARTx_IER register, a 0 to 1 transition of the DR flag requests an interrupt on the dedicated RXREQ output. This signal is routed through the DMA controller. If the associated DMA channel is enabled, the RXREQ signal functions as a DMA request, otherwise the DMA controller simply forwards it to the SIC interrupt controller.

If errors are detected during reception, an interrupt can be requested to a separate error interrupt output. This error request goes directly to the SIC interrupt controller. However, it is hard-wired with the error requests of other modules. The error handler routine may need to interrogate multiple modules as to whether they requested the event. Error requests must be enabled by the ELSI bit in the UARTx_IER register. The following error situations are detected. Every error has an indicating bit in the UARTx_LSR register.

- Overrun error (OE bit)
- Parity error (PE bit)
- Framing error/Invalid stop bit (FE bit)
- Break indicator (BI bit)

Reception is started when a falling edge is detected on the RX input pin. The receiver attempts to see a start bit. For better immunity against noise and hazards on the line, the receiver oversamples every bit 16 times and does a majority decision based on the mid three samples. The data is shifted immediately into the internal RSR register. After the 9th sample of the first stop bit is processed, the received data is copied to the UARTx_RBR register and the receiver recovers itself for further data.

The sampling clock equal to 16 times the bit rate samples the data bits close to their midpoint. Because the receiver clock is usually asynchronous to the transmitter's data rate, the sampling point may drift relative to the center of the data bits. The sampling point is synchronized again with

each start bit, so the error accumulates only over the length of a single word. A receive filter removes spurious pulses of less than two times the sampling clock period.

IrDA Transmit Operation

To generate the IrDA pulse transmitted by the UART, the normal NRZ output of the transmitter is first inverted if the `TPOLC` bit is cleared, so a 0 is transmitted as a high pulse of 16 UART clock periods and a 1 is transmitted as a low pulse for 16 UART clock periods. The leading edge of the pulse is then delayed by six UART clock periods. Similarly, the trailing edge of the pulse is truncated by eight UART clock periods. This results in the final representation of the original 0 as a high pulse of only 3/16 clock periods in a 16-cycle UART clock period. The pulse is centered around the middle of the bit time, as shown in Figure 13-3. The final IrDA pulse is fed to the off-chip infrared driver.

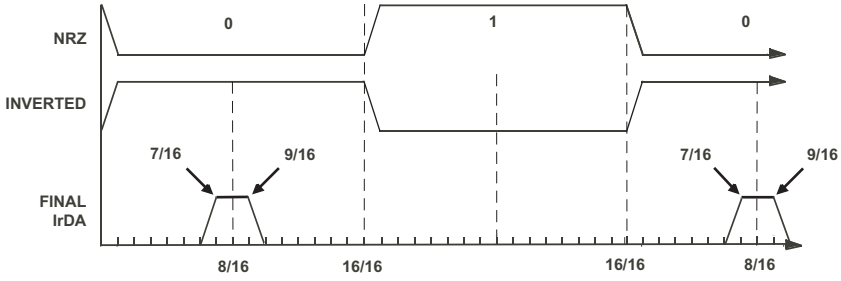


Figure 13-3. IrDA Transmit Pulse

This modulation approach ensures a pulse width output from the UART of three cycles high out of every 16 UART clock cycles. As shown in Table 13-1 on page 13-12, the error terms associated with the bit rate generator are very small and well within the tolerance of most infrared transceiver specifications.

IrDA Receive Operation

The IrDA receiver function is more complex than the transmit function. The receiver must discriminate the IrDA pulse and reject noise. To do this, the receiver looks for the IrDA pulse in a narrow window centered around the middle of the expected pulse.

Glitch filtering is accomplished by counting 16 system clocks from the time an initial pulse is seen. If the pulse is absent when the counter expires, it is considered a glitch. Otherwise, it is interpreted as a 0. This is acceptable because glitches originating from on-chip capacitive cross-coupling typically do not last for more than a fraction of the system clock period. Sources outside of the chip and not part of the transmitter can be avoided by appropriate shielding. The only other source of a glitch is the transmitter itself. The processor relies on the transmitter to perform within specification. If the transmitter violates the specification, unpredictable results may occur. The 4-bit counter adds an extra level of protection at a minimal cost. Note because the system clock can change across systems, the longest glitch tolerated is inversely proportional to the system clock frequency.

The receive sampling window is determined by a counter that is clocked at the 16x bit-time sample clock. The sampling window is re-synchronized with each start bit by centering the sampling window around the start bit.

The polarity of receive data is selectable, using the `IRPOL` bit. Figure 13-4 gives examples of each polarity type.

- `IRPOL` = 0 assumes that the receive data input idles 0 and each active 1 transition corresponds to a UART NRZ value of 0.
- `IRPOL` = 1 assumes that the receive data input idles 1 and each active 0 transition corresponds to a UART NRZ value of 0.

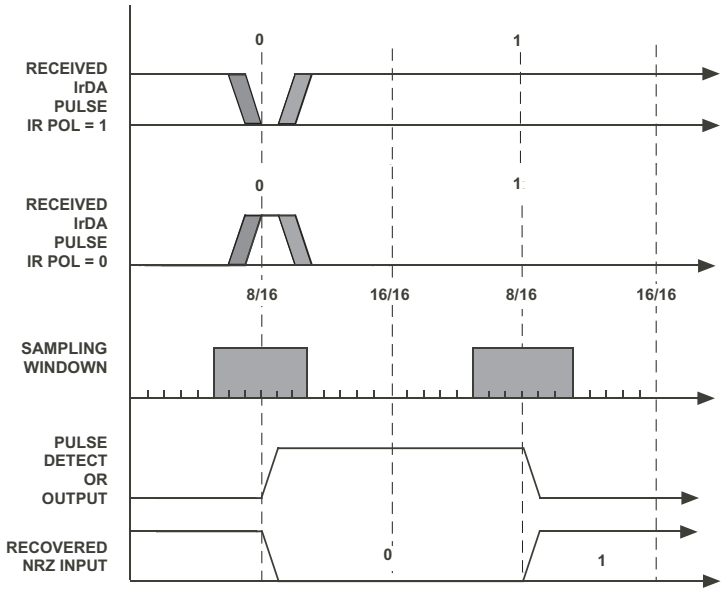


Figure 13-4. IrDA Receiver Pulse Detection

Interrupt Processing

Each UART module has three interrupt outputs. One is dedicated for transmission, one for reception, and the third is used to report line status. As shown in Figure 13-1 on page 13-3, the transmit and receive requests are routed through the DMA controller. The status request goes directly to the SIC controller after being ORed with interrupt signals from other modules.

If the associated DMA channel is enabled, the request functions as a DMA request. If the DMA channel is disabled, it simply forwards the request to the SIC interrupt controller. Note that a DMA channel must be associated with the UART module to enable TX and RX interrupts. Otherwise, the

transmit and receive requests cannot be forwarded. Refer to the description of the peripheral map registers on page 5-71 in the "Direct Memory Access" chapter.

Transmit interrupts are enabled by the ETBEI bit in the UARTx_IER register. If set, the transmit request is asserted when the THRE bit in the UART_LSR register transitions from 0 to 1, indicating that the TX buffer is ready for new data.

Note that the THRE bit resets to 1. When the ETBEI bit is set in the UARTx_IER register, the UART module immediately issues an interrupt or DMA request. This way, no special handling of the first character is required when transmission of a string is initiated. Simply set the ETBEI bit and let the interrupt service routine load the first character from memory and write it to the UARTx_THR register in the normal manner. Accordingly, the ETBEI bit can be cleared if the string transmission has completed. For more information, see "DMA Mode" on page 13-17.

The THRE bit is cleared by hardware when new data is written to the UARTx_THR register. These writes also clear the TX interrupt request. However, they also initiate further transmission. If software doesn't want to continue transmission, the TX request can alternatively be cleared by either clearing the ETBEI bit or by reading the UARTx_IIR register.

Receive interrupts are enabled by the ERBFI bit in the UARTx_IER register. If set, the receive request is asserted when the DR bit in the UART_LSR register transitions from 0 to 1, indicating that new data is available in the UARTx_RBR register. When software reads the UARTx_RBR, hardware clears the DR bit again. Reading UARTx_RBR also clears the RX interrupt request.

Status interrupts are enabled by the ERBFI bit in the UARTx_IER register. If set, the status interrupt request is asserted when any error bit in the UART_LSR register transitions from 0 to 1. Refer to "UARTx_LSR Registers" on page 13-24 for details. Reading the UARTx_LSR register clears the error bits destructively. These reads also clear the status interrupt request.

For legacy reasons, the UARTx_IIR registers still reflect the UART interrupt status. Legacy operation may require bundling all UART interrupt sources to a single interrupt channel and servicing them all by the same software routine. This can be established by globally assigning all UART interrupts to the same interrupt priority, by using the System Interrupt Controller (SIC).

If either the line status interrupt or the receive data interrupt has been assigned a lower interrupt priority by the SIC, a deadlock condition can occur. To avoid this, always assign the lowest priority of the enabled UART interrupts to the UARTx_THR empty event.

Bit Rate Generation

The UART clock is enabled by the UCEN bit in the UARTx_GCTL register.

The bit rate is characterized by the system clock (SCLK) and the 16-bit divisor. The divisor is split into the UARTx_DLL and the UARTx_DLH registers. These registers form a 16-bit divisor. The bit clock is divided by 16 so that:

BIT RATE = SCLK/(16 x Divisor)

Divisor = 65,536 when UARTx_DLL = UARTx_DLH = 0

Table 13-1 provides example divide factors required to support most standard baud rates.

Table 13-1. UART Bit Rate Examples With 100 MHz SCLK

Bit Rate	DL	Actual	% Error
2400	2604	2400.15	.006
4800	1302	4800.31	.007
9600	651	9600.61	.006
19200	326	19171.78	.147

Table 13-1. UART Bit Rate Examples With 100 MHz SCLK (Cont'd)

Bit Rate	DL	Actual	% Error
38400	163	38343.56	.147
57600	109	57339.45	.452
115200	54	115740.74	.469
921600	7	892857.14	3.119
6250000	1	6250000	-

Careful selection of SCLK frequencies, that is, even multiples of desired bit rates, can result in lower error percentages.

Note that the UART module is clocked 16 times faster than the bit clock. This is required to oversample bits on reception and to generate RZI code in IrDA mode.

Autobaud Detection

At the chip level, the UART RX pins are routed to the alternate capture inputs (TACIx) of the general purpose timers. When working in WDTH_CAP mode these timers can be used to automatically detect the bit rate applied to the RX pin by an external device. For more information, see Chapter 15, "General-Purpose Timers".

The capture capabilities of the timers are often used to supervise the bit rate at runtime. If the Blackfin UART was talking to any device supplied by a weak clock oscillator that drifts over time, the Blackfin can re-adjust its UART bit rate dynamically as required.

Often, autobaud detection is used for initial bit rate negotiations. There, the Blackfin processor is most likely a slave device waiting for the host to send a predefined autobaud character as discussed below. This is exactly the scenario used for UART booting. In this scenario, it is recommended that the UART clock enable bit UCEN is not enabled while autobaud

detection is performed to prevent the UART from starting reception with incorrect bit rate matching. Alternatively, the UART can be disconnected from the RX pin by setting the LOOP bit.

A software routine can detect the pulse widths of serial stream bit cells. Because the sample base of the timers is synchronous with the UART operation—all derived from SCLK—the pulse widths can be used to calculate the baud rate divider for the UART.

```
DIVISOR = (TIMERx_WIDTH) / (16 x Number of captured UART bits)
```

In order to increase the number of timer counts and therefore the resolution of the captured signal, it is recommended not to measure just the pulse width of a single bit, but to enlarge the pulse of interest over more bits. Traditionally, a NULL character (ASCII 0x00) was used in autobaud detection, as shown in Figure 13-5.

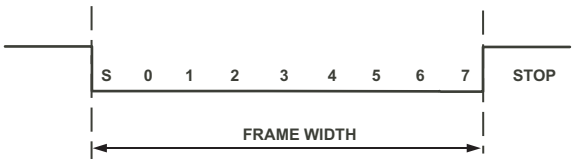


Figure 13-5. Autobaud Detection Character 0x00

Because the example frame in Figure 13-5 encloses 8 data bits and 1 start bit, apply the formula:

```
DIVISOR = TIMERx_WIDTH/(16 x 9)
```

Real UART RX signals often have asymmetrical falling and rising edges, and the sampling logic level is not exactly in the middle of the signal voltage range. At higher bit rates, such pulse width-based autobaud detection might not return adequate results without additional analog signal conditioning. Measuring signal periods works around this issue and is strongly recommended.

For example, predefine ASCII character “@” (0x40) as the autobaud detection character and measure the period between two subsequent falling edges. As shown in Figure 13-6, measure the period between the falling edge of the start bit and the falling edge after bit 6. Since this period encloses 8 bits, apply the formula:

```
DIVISOR = TIMERx_PERIOD >> 7
```

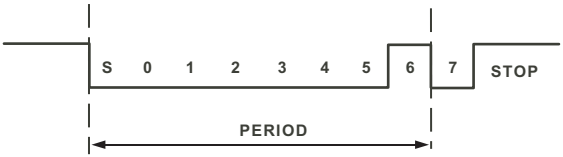


Figure 13-6. Autobaud Detection Character 0x40

An example is provided in Listing 13-2 on page 13-33.

Programming Model

The following sections describe a programming model for the UARTs.

Non-DMA Mode

In non-DMA mode, data is moved to and from the UART by the processor core. To transmit a character, load it into UARTx_THR. Received data can be read from UARTx_RBR. The processor must write and read one character at time.

To prevent any loss of data and misalignments of the serial datastream, the UARTx_LSR register provides two status flags for handshaking—THRE and DR.

The THRE flag is set when UARTx_THR is ready for new data and cleared when the processor loads new data into UARTx_THR. Writing UARTx_THR when it is not empty overwrites the register with the new value and the previous character is never transmitted.

The DR flag signals when new data is available in UARTx_RBR. This flag is cleared automatically when the processor reads from UARTx_RBR. Reading UARTx_RBR when it is not full returns the previously received value. When UARTx_RBR is not read in time, newly received data overwrites UARTx_RBR and the OE flag is set.

With interrupts disabled, these status flags can be polled to determine when data is ready to move. Note that because polling is processor intensive, it is not typically used in real-time signal processing environments. Be careful if transmit and receive are served by different software threads, because read operations on UART_LSR and UART_IIR registers are destructive. Polling the SIC_ISR register without enabling the interrupts by SIC_MASK is an alternate method of operation to consider. Software can write up to two words into the UARTx_THR register before enabling the UART clock. As soon as the UCEN bit is set, those two words are sent.

Alternatively, UART writes and reads can be accomplished by interrupt service routines (ISRs). Separate interrupt lines are provided for UART TX, UART RX, and UART error. The independent interrupts can be enabled individually by the UARTx_IER register.

The ISRs can evaluate the status bit field within the UARTx_IIR register to determine the signalling interrupt source. If more than one source is signalling, the status field displays the one with the highest priority. Interrupts also must be assigned and unmasked by the processor's interrupt controller. The ISRs must clear the interrupt latches explicitly. See Figure 13-13 on page 13-29.

DMA Mode

In this mode, separate receive (RX) and transmit (TX) DMA channels move data between the UART and memory. The software does not have to move data, it just has to set up the appropriate transfers either through the descriptor mechanism or through autobuffer mode.

DMA channels provide a 4-deep FIFO, resulting in total buffer capabilities of 6 words at both the transmit and receive sides. In DMA mode, the latency is determined by the bus activity and arbitration mechanism and not by the processor loading and interrupt priorities. For more information, see Chapter 5, “Direct Memory Access”.

DMA interrupt routines must explicitly write 1s to the corresponding `DMAx_IRQ_STATUS` registers to clear the latched request of the pending interrupt.

The UART’s DMA is enabled by first setting up the system DMA control registers and then enabling the UART `ERBFI` and/or `ETBEI` interrupts in the `UARTx_IER` register. This is because the interrupt request lines double as DMA request lines. Depending on whether DMA is enabled or not, upon receiving these requests, the DMA control unit either generates a direct memory access or passes the UART interrupt on to the system interrupt handling unit. The UART’s error interrupt goes directly to the system interrupt handling unit, bypassing the DMA unit completely.

For transmit DMA, it is recommended to set the `SYNC` bit in the `DMAx_CONFIG` register. With this bit set, the interrupt generation is delayed until the entire DMA FIFO has been drained to the UART module. The UART TX DMA interrupt service routine is allowed to start another DMA sequence or to clear the `ETBEI` control bit only when the `SYNC` bit is set.

If another DMA is started while data is still pending in the UART transmitter, there is no need to pulse the ETBEI bit to initiate the second DMA. If, however, the recent byte has already been loaded into the TSR registers (that is, the THRE bit is set), then the ETBEI bit must be cleared and set again to let the second DMA start.

In DMA transmit mode, the ETBEI bit enables the peripheral request to the DMA FIFO. The strobe on the memory side is still enabled by the DMAEN bit. If the DMA count is less than the DMA FIFO depth, which is 4, then the DMA interrupt might be requested already before the ETBEI bit is set. If this is not wanted, set the SYNC bit in the DMAx_CONFIG register.

Regardless of the SYNC setting, the DMA stream has not left the UART transmitter completely at the time the interrupt is generated. Transmission may abort in the middle of the stream, causing data loss, if the UART clock was disabled without additional polling of the TEMT bit.

The UART's DMA supports 8-bit and 16-bit operation, but not 32-bit operation. Sign extension is not supported.

Mixing Modes

Especially on the transmit side, switching from DMA mode to non-DMA operation on the fly requires some thought. By default, the interrupt timing of the DMA is synchronized with the memory side of the DMA FIFOs. The TX DMA completion interrupt is generated after the last byte has been copied from the memory into the DMA FIFO. The TX DMA interrupt service routine is not yet permitted to start other DMA sequences or to switch to non-DMA transmission. The interrupt is requested by the time the DMA_DONE bit is set. The DMA_RUN bit, however, remains set until the data has completely left the TX DMA FIFO.

Therefore, when planning to switch from DMA to non-DMA of operation, always set the SYNC bit in the DMAx_CONFIG word of the last descriptor or work unit before handing over control to non-DMA mode. Then, after

the interrupt occurs, software can write new data into the UARTx_THR register as soon as the THRE bit permits. If the SYNC bit cannot be set, software can poll the DMA_RUN bit instead.

When switching from non-DMA to DMA operation, take care that the very first DMA request is issued properly. If the DMA is enabled while the UART is still transmitting, no precaution is required. If, however, the DMA is enabled after the TEMT bit became high, the ETBEI bit should be pulsed to initiate DMA transmission.

UART Registers

The processor provides a set of PC-style industry-standard control and status registers for each UART. These Memory-mapped Registers (MMRs) are byte-wide registers that are mapped as half words with the most significant byte zero filled. Table 13-2 provides an overview of the UART registers.

Consistent with industry-standard devices, multiple registers are mapped to the same address location. The UARTx_DLH and UARTx_DLL registers share their addresses with the UARTx_THR registers, the UARTx_RBR registers, and the UARTx_IER registers. The DLAB bit in the UARTx_LCR registers controls which set of registers is accessible at a given time. Software must use 16-bit word load/store instructions to access these registers.

Transmit and receive channels are both buffered. The UARTx_THR registers buffer the transmit shift register (TSR) and the UARTx_RBR registers buffer the receive shift register (LSR). The shift registers are not directly accessible by software.

Table 13-2. UART Register Overview

Name	Address Offset	DLAB	Operation	Reset Value	Function
UARTx_RBR	0x0000	0	R	0x00	Receive buffer register
UARTx_THR	0x0000	0	W	0x00	Transmit hold register
UARTx_DLL	0x0000	1	R/W	0x01	Divisor latch low byte
UARTx_IER	0x0004	0	R/W	0x00	Interrupt enable register
UARTx_DLH	0x0004	1	R/W	0x00	Divisor latch high byte
UARTx_IIR	0x0008	X	R Read operations are destructive	0x01	Interrupt identification register
UARTx_LCR	0x000C	X	R/W	0x00	Line control register
UARTx_MCR	0x0010	X	R/W	0x00	Modem control register
UARTx_LSR	0x0014	X	R Read operations are destructive	0x60	Line status register
UARTx_SCR	0x001C	X	R/W	0x00	Scratch register
UARTx_GCTL	0x0024	X	R/W	0x00	Global control register

UARTx_LCR Registers

The UARTx_LCR registers, shown in Figure 13-7, control the format of received and transmitted character frames.

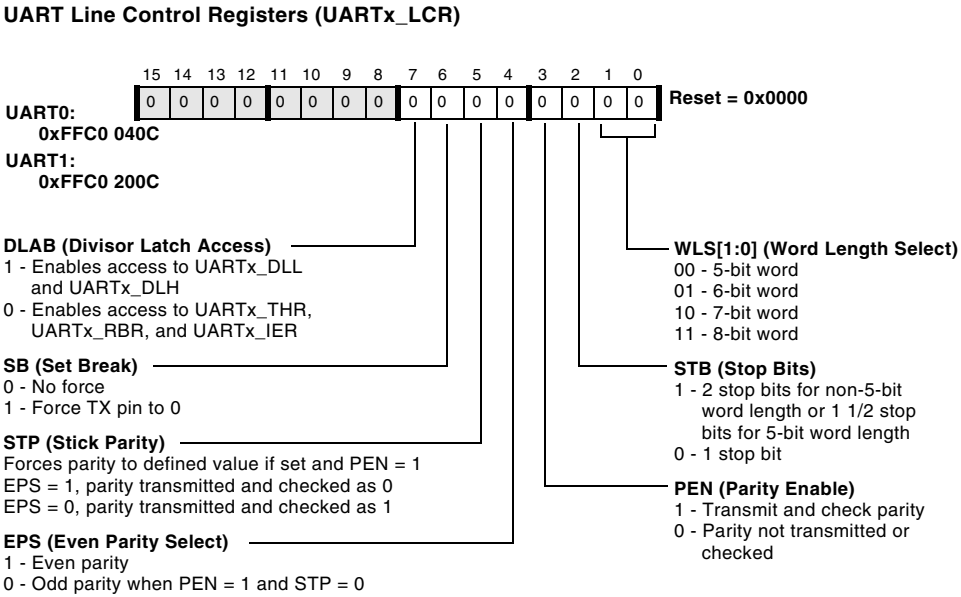


Figure 13-7. UART Line Control Registers

The 2-bit WLS field determines whether the transmitted and received UART word consists of 5, 6, 7 or 8 data bits.

The STB bit controls how many stop bits are appended to transmitted data. When STB=0, one stop bit is transmitted. If WLS is non zero, STB=1 instructs the transmitter to add one additional stop bit, two stop bits in total. If WLS=0 and 5-bit operation is chosen, STB=1 forces the transmitter to append one additional half bit, 1 1/2 stop bits in total. Note that this bit does not impact data reception—the receiver is always satisfied with one stop bit.

The PEN bit inserts one additional bit between the most significant data bit and the first stop bit. The polarity of this so-called parity bit depends on data and the STP and EPS control bits. Both transmitter and receiver calculate the parity value. The receiver compares the received parity bit with the expected value and issues a parity error if they don't match. If PEN is cleared, the STP and the EPS bits are ignored.

The STP bit controls whether the parity is generated by hardware based on the data bits or whether it is set to a fixed value. If STP=0 the hardware calculates the parity bit value based on the data bits. Then, the EPS bit determines whether odd or even parity mode is chosen. If EPS=0, odd parity is used. That means that the total count of logical-1 data bits including the parity bit must be an odd value. Even parity is chosen by STP=0 and EPS=1. Then, the count of logical-1 bits must be a even value. If the STP bit is set, then hardware parity calculation is disabled. In this case, the sent and received parity equals the inverted EPS bit. The example in Table 13-3 summarizes polarity behavior assuming 8-bit data words (WLS=3).

Table 13-3. UART Priority

PEN	STP	EPS	Data (hex)	Data (binary, LSB first)	Parity
0	x	x	x	x	None
1	0	0	0x60	0000 0110	1
1	0	0	0x57	1110 1010	0
1	0	1	0x60	0000 0110	0
1	0	1	0x57	1110 1010	1
1	1	0	x	x	1
1	1	0	x	x	1
1	1	1	x	x	0
1	1	1	x	x	0

If set, the SB bit forces the TX pin to low asynchronously, regardless of whether or not data is currently transmitted. It functions even when the UART clock is disabled. Since the TX pin normally drives high, it can be used as a flag output pin, if the UART is not used.

The DLAB bit controls whether the UARTx_RBR, UARTx_THR and UARTx_IER registers are accessible by the PAB bus (DLAB=0) or the divisor latch registers DLH and DLL alternatively (DLAB=1).

UARTx_MCR Registers

The UARTx_MCR registers control the UART port, as shown in Figure 13-8. Even if modem functionality is not supported, the UART modem control registers are available in order to support the loopback mode.

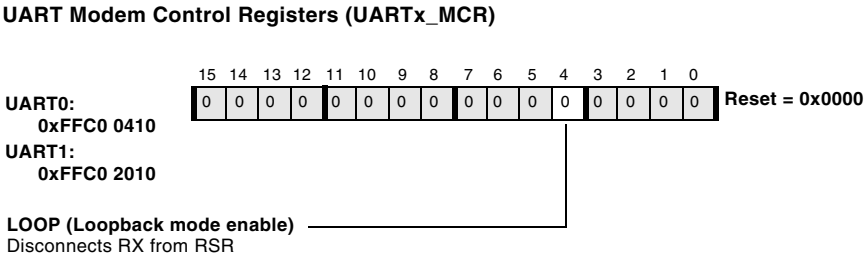


Figure 13-8. UART Modem Control Registers

Loopback mode disconnects the receiver's input from the RX pin, but redirects it to the transmit output internally.

UARTx_LSR Registers

The UARTx_LSR registers contain UART status information as shown in Figure 13-9.

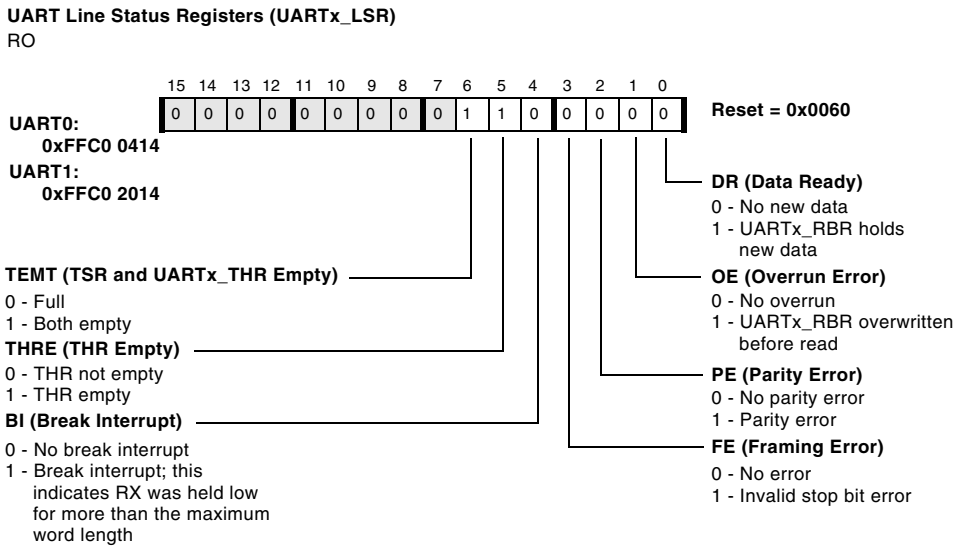


Figure 13-9. UART Line Status Registers

The DR bit indicates that data is available in the receiver and can be read from the UARTx_RBR register. The bit is set by hardware when the receiver detects the first valid stop bit. It is cleared by hardware when the UARTx_RBR register is read.

The OE bit indicates that a start bit condition has been detected, but the internal receive shift register (RSR) and the receive buffer (UARTx_RBR) already contain data. New data overwrites the content of the buffers. To avoid overruns read the UARTx_RBR register in time. The OE bit cleared when the LSR register is read.

The PE bit indicates that the received parity bit does not match the expected value. The PE bit is set simultaneously with the DR bit. The PE bit cleared when the LSR register is read. Invalid parity bits can be simulated by setting the FPE bit in the UARTx_GCTL register.

The FE bit indicates that the first stop bit has been sampled low. It is cleared by hardware when the UARTx_RBR register is read. Invalid stop bits can be simulated by setting the FFE bit in the UARTx_GCTL register.

The BI bit indicates that the first stop bit has been sampled low and the entire data word, including parity bit, consists of low bits only. It is cleared by hardware when the UARTx_RBR register is read.

Because of the destructive nature of these read operations, special care should be taken. For more information, see the Memory chapter of the *ADSP-BF53x/BF56x Blackfin Processor Programming Reference*.

The THRE bit indicates that the UART transmit channel is ready for new data and software can write to UARTx_THR. Writes to UARTx_THR clear the THRE bit. It is set again when data is passed from UARTx_THR to the internal TSR register.

The TEMT bit indicates that both the UARTx_THR register and the internal TSR register are empty. In this case the program is permitted to write to the UARTx_THR register twice without losing data. The TEMT bit can also be used as indicator that pending UART transmission has been completed. At that time it is safe to disable the UCEN bit or to three-state the off-chip line driver.

UARTx_THR Registers

The write-only UARTx_THR registers, shown in Figure 13-10, are mapped to the same address as the read-only UARTx_RBR and UARTx_DLL registers. To access UARTx_THR, the DLAB bit in UARTx_LCR must be cleared. When the DLAB bit is cleared, writes to this address target the UARTx_THR register, and reads from this address return the UARTx_RBR register.

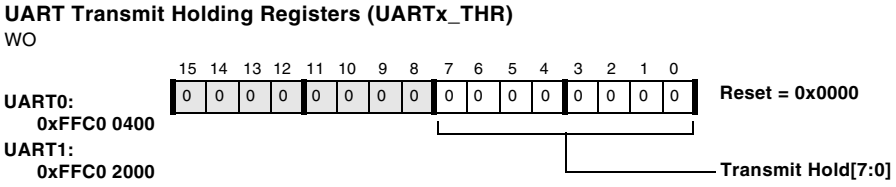


Figure 13-10. UART Transmit Holding Registers

UARTx_RBR Registers

The read-only UARTx_RBR registers, shown in Figure 13-11, are mapped to the same address as the write-only UARTx_THR and UARTx_DLL registers. To access UARTx_RBR, the DLAB bit in UARTx_LCR must be cleared. When the DLAB bit is cleared, writes to this address target the UARTx_THR register, while reads from this address return the UARTx_RBR register.

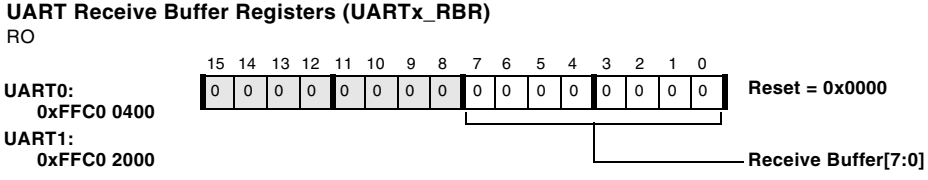


Figure 13-11. UART Receive Buffer Registers

UARTx_IER Registers

The UARTx_IER registers, shown in Figure 13-12, are used to enable requests for system handling of empty or full states of UART data registers. Unless polling is used as a means of action, the ERBFI and/or ETBEI bits in this register are normally set.

Setting this register without enabling system DMA causes the UART to notify the processor of data inventory state by means of interrupts. For proper operation in this mode, system interrupts must be enabled, and appropriate interrupt handling routines must be present. For backward compatibility, the UARTx_IIR still reflects the correct interrupt status.

Each UART features three separate interrupt channels to handle data transmit, data receive, and line status events independently, regardless whether DMA is enabled or not. At system level the two UART status interrupt channels are ORed prior being connected to the SIC controller.

With system DMA enabled, the UART uses DMA to transfer data to or from the processor. Dedicated DMA channels are available to receive and transmit operation. Line error handling can be configured completely independently from the receive/transmit setup.

The UARTx_IER registers are mapped to the same address as the UARTx_DLH registers. To access UARTx_IER, the DLAB bit in UARTx_LCR must be cleared.

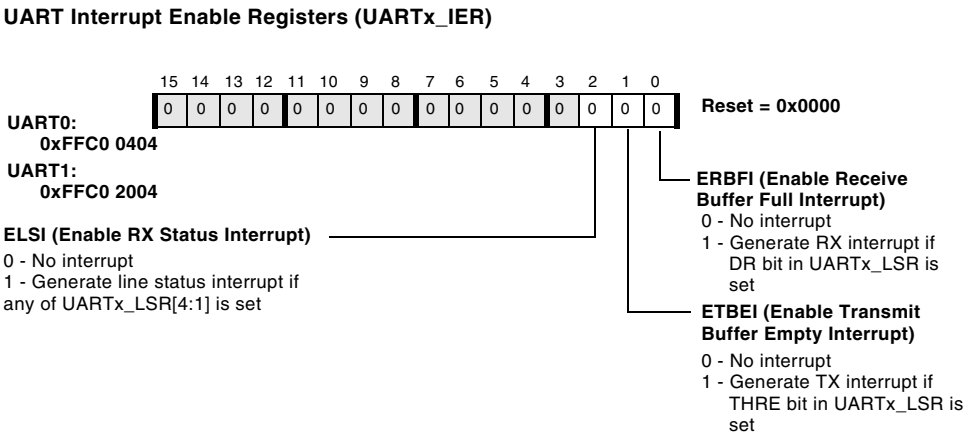


Figure 13-12. UART Interrupt Enable Registers

The UART's DMA is enabled by first setting up the system DMA control registers and then enabling the UART `ERBFI` and/or `ETBEI` interrupts in the `UARTx_IER` register. This is because the interrupt request lines double as DMA request lines. Depending on whether DMA is enabled or not, upon receiving these requests, the DMA control unit either generates a direct memory access or passes the UART interrupt on to the system interrupt handling unit. However, UART's error interrupt goes directly to the system interrupt handling unit, bypassing the DMA unit completely.

The `ELSI` bit enables interrupt generation on an independent interrupt channel when any of the following conditions are raised by the respective bit in the `UARTx_LSR` register:

- Receive overrun error (`OE`)
- Receive parity error (`PE`)
- Receive framing error (`FE`)
- Break interrupt (`BI`)

UARTx_IIR Registers

When cleared, the NINT bit signals that an interrupt is pending. The STATUS field indicates the highest priority pending interrupt. The receive line status has the highest priority; the UARTx_THR empty interrupt has the lowest priority. In the case where both interrupts are signalling, the UARTx_IIR reads 0x06.

When a UART interrupt is pending, the interrupt service routine (ISR) needs to clear the interrupt latch explicitly. Figure 13-13 shows how to clear any of the three latches.

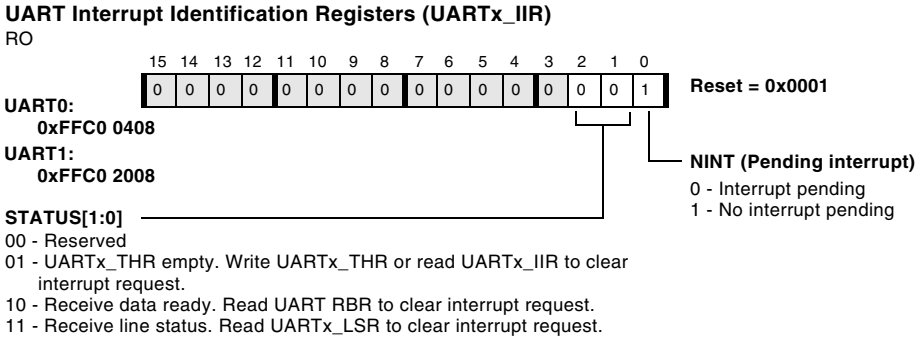


Figure 13-13. UART Interrupt Identification Registers

The TX interrupt request is cleared by writing new data to the UARTx_THR register or by reading the UARTx_IIR register. Please note the special role of the UARTx_IIR register read in the case where the service routine does not want to transmit further data.

If software stops transmission, it must read the UARTx_IIR register to reset the interrupt request. As long as the UARTx_IIR register reads 0x04 or 0x06 (indicating that another interrupt of higher priority is pending), the UARTx_THR empty latch cannot be cleared by reading UARTx_IIR.

Because of the destructive nature of these read operations, special care should be taken. For more information, see the Memory chapter of the *ADSP-BF53x/BF56x Blackfin Processor Programming Reference*.

UARTx_DLL and UARTx_DLH Registers

The UARTx_DLL registers are mapped to the same addresses as the UARTx_THR and UARTx_RBR registers. The UARTx_DLH registers are mapped to the same addresses as the UARTx_IER registers. The DLAB bit in UARTx_LCR must be set before the UARTx_DLL and UARTx_DLH registers, shown in Figure 13-14, can be accessed.

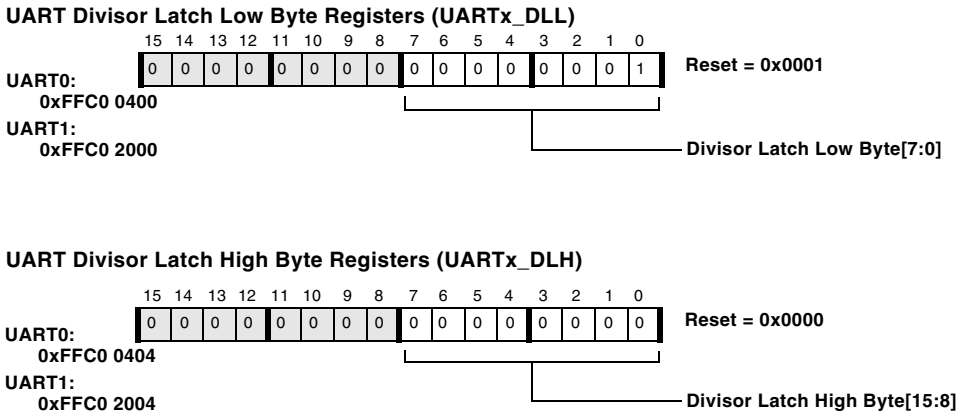


Figure 13-14. UART Divisor Latch Registers

Note the 16-bit divisor formed by UARTx_DLH and UARTx_DLL resets to 0x0001, resulting in the highest possible clock frequency by default. If the UART is not used, disabling the UART clock saves power. The UARTx_DLH and UARTx_DLL registers can be programmed by software before or after setting the UCEN bit.

UARTx_SCR Registers

The contents of the 8-bit UARTx_SCR registers, shown in Figure 13-15, are reset to 0x00. They are used for general-purpose data storage and do not control the UART hardware in any way.

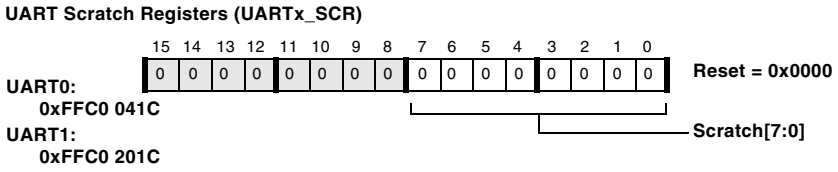


Figure 13-15. UART Scratch Registers

UARTx_GCTL Registers

The UARTx_GCTL registers, shown in Figure 13-16, contain the enable bit for internal UART clocks and for the IrDA mode of operation of the UARTs.

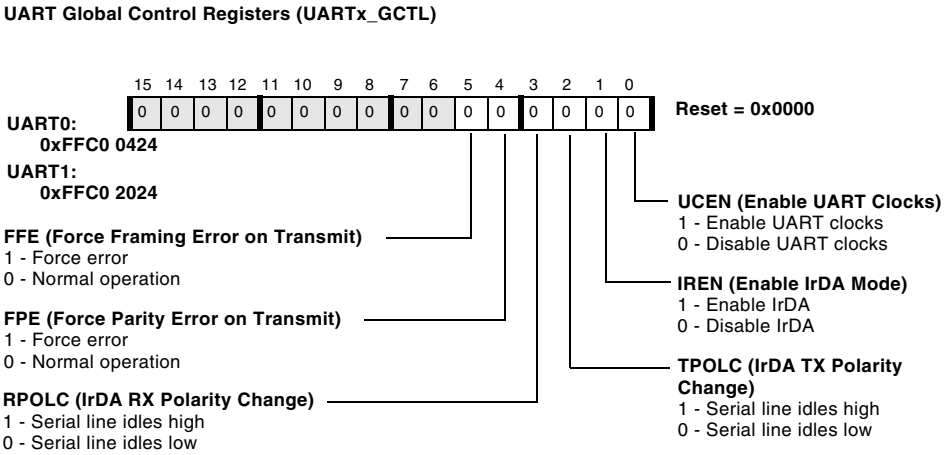


Figure 13-16. UART Global Control Registers

The UCEN bit enables the UART clocks. It also resets the state machine and control registers when cleared.

Note that the UCEN bit was not present in previous UART implementations. It has been introduced to save power if the UART is not used. When porting code, be sure to enable this bit.

The IrDA TX polarity change bit and the IrDA RX polarity change bit are effective only in IrDA mode. The two force error bits, FPE and FFE, are intended for test purposes. They are useful for debugging software, especially in loopback mode.

Programming Examples

The subroutine in Listing 13-1 shows a typical UART initialization sequence.

Listing 13-1. UART Initialization

```
/*************************************************************
 *  Configures UART in 8 data bits, no parity, 1 stop bit mode.
 *  Input parameters: r0 holds divisor latch value to be
 *                    written into
 *                    DLH:DLL registers.
 *                    p0 contains the UARTx_GCTL register address
 *  Return values:    none
 *************************************************************/
uart_init:
    [--sp] = r7;
    r7 = UCEN (z);  /* First of all, enable UART clock */
    w[p0+UART0_GCTL-UART0_GCTL] = r7;

    r7 = DLAB (z);                    /* to set bit rate    */
    w[p0+UART0_LCR-UART0_GCTL] = r7;  /* set DLAB bit first */
```

```
    w[p0+UART0_DLL-UART0_GCTL] = r0;  /* write lower byte to DLL
*/
    r7 = r0 >> 8;
    w[p0+UART0_DLH-UART0_GCTL] = r7; /* write upper byte to DLH
*/

    r7 = STB | WLS(8) (z);            /* clear DLAB again and con-
fig to */
   w[p0+UART0_LCR-UART0_GCTL] = r7;  /* 8 bits, no parity, 2 stop
bits */

    r7 = [sp++];
    rts;
uart_init.end:
```

The subroutine in Listing 13-2 performs autobaud detection similarly to UART boot.

Listing 13-2. UART Autobaud Detection Subroutine

```
/*************************************************************
 *  Assuming 8 data bits, this functions expects a '@'
 *  (ASCII 0x40) character
 *  on the UARTx RX pin. A Timer performs the autobaud detection.
 *  Input parameters: p0 contains the UARTx_GCTL register address
 *                    p1 contains the TIMERx_CONFIG register
 *  address
 *  Return values:    r0 holds timer period value (equals 8 bits)
*************************************************************/
uart_autobaud:
    [--sp] = (r7:5,p5:5);
    r5.h = hi(TIMER0_CONFIG);  /* for generic timer use calculate
*/
    r5.l = lo(TIMER0_CONFIG);  /* specific bits first */
    r7 = p1;
```

```
    r7 = r7 - r5;
    r7 >>= 4;  /* r7 holds the 'x' of TIMERx_CONFIG now */
    r5 = TIMEN0 (z);
    r5 <<= r7;  /* r5 holds TIMENx/TIMDISx now */
    r6 = TRUN0 | TOVL_ERR0 | TIMIL0 (z);
    r6 <<= r7;
    CC = r7 <= 3;
    r7 = r6 << 12;
    if !CC r6 = r7;  /* r6 holds TRUNx | TOVL_ERRx | TIMILx */

    p5.h = hi(TIMER_STATUS);
    p5.l = lo(TIMER_STATUS);
    w[p5 + TIMER_DISABLE - TIMER_STATUS] = r5; /* disable Timer x
*/
    [p5 + TIMER_STATUS - TIMER_STATUS] = r6; /* clear pending
latches */
    /* period capture, falling edge to falling edge */
    r7 = TIN_SEL | IRQ_ENA | PERIOD_CNT | WDTH_CAP (z);
    w[p1 + TIMER0_CONFIG - TIMER0_CONFIG] = r7;
    w[p5+TIMER_ENABLE-TIMER_STATUS] = r5;

uart_autobaud.wait:  /* wait for timer event */
        r7 = w[p5 + TIMER_STATUS - TIMER_STATUS] (z);
        r7 = r7 & r5;
        CC = r7 == 0;
        if CC jump uart_autobaud.wait;
    w[p5 + TIMER_DISABLE - TIMER_STATUS] = r5; /* disable Timer x
*/
    [p5 + TIMER_STATUS - TIMER_STATUS] = r6; /* clear pending
latches */
    /* Save period value to R0 */
    r0 = [p1 + TIMER0_PERIOD - TIMER0_CONFIG];

    /* delay processing as autobaud character is still ongoing */
```

```
    r7 = OUT_DIS | IRQ_ENA | PERIOD_CNT | PWM_OUT (z);
    w[p1 + TIMER0_CONFIG - TIMER0_CONFIG] = r7;
    w[p5 + TIMER_ENABLE - TIMER_STATUS] = r5;

uart_autobaud.delay:
        r7 = w[p5 + TIMER_STATUS - TIMER_STATUS] (z);
        r7 = r7 & r5;
        CC = r7 == 0;
        if CC jump uart_autobaud.delay;
    w[p5 + TIMER_DISABLE - TIMER_STATUS] = r5;
    [p5 + TIMER_STATUS - TIMER_STATUS] = r6;
    (r7:5,p5:5) = [sp++];
    rts;
uart_autobaud.end:
```

The parent routine in Listing 13-3 performs autobaud detection using UART0 and TIMER1.

Listing 13-3. UART Autobaud Detection Parent Routine

```
    p0.l = lo(PORTF_FER);  /* function enable on UART0 pins PF0
and PF1 */
    p0.h = hi(PORTF_FER);  /* by default PORT_MUX register is all
set */
    r0 = PF1 | PF0 (z)
    w[p0] = r0;
    p0.l = lo(UART0_GCTL); /* select UART 0 */
    p0.h = hi(UART0_GCTL);
    p1.l = lo(TIMER1_CONFIG); /* select TIMER 1 */
    p1.h = hi(TIMER1_CONFIG);
    call uart_autobaud;
    r0 >>= 7;       /* divide PERIOD value by (16 x 8) */
    call uart_init;
    ...
```

The subroutine in Listing 13-4 transmits a character by polling operation.

Listing 13-4. UART Character Transmission

```
/*****************************************************
 *  Transmit a single byte by polling the THRE bit.
 *  Input parameters: r0 holds the character to be transmitted
 *                    p0 contains UARTx_GCTL register address
 *  Return values: none
*****************************************************/
uart_putc:
    [--sp] = r7;
uart_putc.wait:
        r7 = w[p0+UART0_LSR-UART0_GCTL] (z);
        CC = bittst(r7, bitpos(THRE));
        if !CC jump uart_putc.wait;
    w[p0+UART0_THR-UART0_GCTL] = r0; /* write initiates transfer
*/
    r7 = [sp++];
    rts;
uart_putc.end:
```

Use the routine shown in Listing 13-5 to transmit a C-style string that is terminated by a null character.

Listing 13-5. UART String Transmission

```
/*************************************************************
 *  Transmit a null-terminated string.
 *  Input parameters: p1 points to the string
 *                    p0 contains UARTx_GCTL register address
 *  Return values: none
*************************************************************/
uart_puts:
```

```
    [--sp] = rets;
    [--sp] = r0;
uart_puts.loop:
        r0 = b[p1++] (z);
        CC = r0 == 0;
        if CC jump uart_puts.exit;
        call uart_putc;
        jump uart_puts.loop;
uart_puts.exit:
    r0 = [sp++];
    rets = [sp++];
    rts;
uart_puts.end:
```

Note that polling the UART0_LSR register for transmit purposes may clear the receive error latch bits. It is, therefore, not recommended to poll UART0_LSR for transmission this way while data is received. In case, write a polling loop that reads UARTx_LSR once and then evaluates *all* status bits of interest, as shown in Listing 13-6.

Listing 13-6. UART Polling Loop

```
uart_loop:
    r7 = w[p0+UART0_LSR-UART0_GCTL] (z);
    CC = bittst(r7, bitpos(DR));
    if !CC jump uart_loop.transmit;
    r6 = w[p0+UART0_RBR-UART0_GCTL] (z);
    r5 = BI | OE | FE | PE (z);
    r5 = r5 & r7;
    CC = r5 == 0;
    if !CC jump uart_loop.error;
    b[p1++] = r6;          /* store byte */
uart_loop.transmit:
    CC = bittst(r7, bitpos(THRE));
    if !CC jump uart_loop;
```

```
    r5 = b[p2++] (z);     /* load next byte */
    w[p0+UART0_THR-UART0_GCTL] = r5;
    jump uart_loop;
uart_loop.error:
    ...
    jump uart_loop;
```

In non-DMA interrupt operation, the three UART interrupt request lines may or may not be ORed together in the SIC controller. If they had three different service routines, they may look as shown in Listing 13-7.

Listing 13-7. UART Non-DMA Interrupt Operation

```
isr_uart_rx:
    [--sp] = astat;
    [--sp] = r7;
    r7 = w[p0+UART0_RBR-UART0_GCTL] (z);
    b[p4++] = r7;
    ssync;
    r7 = [sp++];
    astat = [sp++];
    rti;
isr_uart_rx.end:

isr_uart_tx:
    [--sp] = astat;
    [--sp] = r7;
    r7 = b[p3++] (z);
    CC = r7 == 0;
    if CC jump isr_uart_tx.final;
    w[p0+UART0_THR-UART0_GCTL] = r7;
    r7 = [sp++];
    astat = [sp++];
    ssync;
    rti;
```

```
isr_uart_tx.final:
    r7 = w[p0+UART0_IER-UART0_GCTL] (z);  /* clear TX interrupt
enable */
    bitclr(r7, bitpos(ETBEI));  /* ensure this sequence is not */
    w[p0+UART0_IER-UART0_GCTL] = r7;      /* interrupted by other
IER accesses */
    ssync;
    r7 = [sp++];
    astat = [sp++];
    rti;
isr_uart_tx.end:

isr_uart_error:
    [--sp] = astat;
    [--sp] = r7;
    r7 = w[p0+UART0_LSR-UART0_GCTL] (z);  /* read clears inter-
rupt request */
    /* do something with the error */
    r7 = [sp++];
    astat = [sp++];
    ssync;
    rti;
isr_uart_error.end:
```

Listing 13-8 transmits a string by DMA operation, waits until DMA completes and sends an additional string by polling. Note the importance of the SYNC bit.

Listing 13-8. UART Transmission SYNC Bit Use

```
.section data;
.byte sHello[] = 'Hello Blackfin User',13,10,0;
.byte sWorld[] = 'How is life?',13,10,0;

.section program;
```

```
...
p1.l = lo(IMASK);
p1.h = hi(IMASK);
r0.l = lo(isr_uart_tx);     /* register service routine */
r0.h = hi(isr_uart_tx);     /* UART0 TX defaults to IVG10 */
r0 = [p1 + IMASK - IMASK];  /* unmask interrupt in CEC */
bitset(r0, bitpos(EVT_IVG10));
[p1] = r0;
p1.l = lo(SIC_IMASK);
p1.h = hi(SIC_IMASK);       /* unmask interrupt in SIC */
r0.l = 0x1000;
r0.h = 0x0000;
[p1] = r0;
[--sp] = reti;        /* enable nesting of interrupts */

p5.l = lo(DMA9_CONFIG);     /* setup DMA in STOP mode */
p5.h = hi(DMA9_CONFIG);
r7.l = lo(sHello);
r7.h = hi(sHello);
[p5+DMA9_START_ADDR-DMA9_CONFIG] = r7;
r7 = length(sHello) (z);
r7+= -1;    /* don't send trailing null character */
w[p5+DMA9_X_COUNT-DMA9_CONFIG] = r7;
r7 = 1;
w[p5+DMA9_X_MODIFY-DMA9_CONFIG] = r7;
r7 = FLOW_STOP | WDSIZE_8 | DI_EN | SYNC | DMAEN (z);
w[p5] = r7;

p0.l = lo(UART0_GCTL); /* select UART 0 */
p0.h = hi(UART0_GCTL);
r0 = ETBEI (z);         /* enable and issue first request */
w[p0+UART0_IER-UART0_GCTL] = r0;
```

```
wait4dma: /* just one way to synchronize with the service routine
*/
    r0 = w[p5+DMA9_IRQ_STATUS-DMA9_CONFIG] (z);
    CC = bittst(r0,bitpos(DMA_RUN));
    if CC jump wait4dma;
    p1.l=lo(sWorld);
    p1.h=hi(sWorld);
    call uart_puts;

forever: jump forever;

isr_uart_tx:
    [--sp] = astat;
    [--sp] = r7;
    r7 = DMA_DONE (z);    /* W1C interrupt request */
    w[p5+DMA9_IRQ_STATUS-DMA9_CONFIG] = r7;
    r7 = 0;         /* pulse ETBEI for general case */
    w[p0+UART0_IER-UART0_GCTL] = r7;
    ssync;
    r7 = [sp++];
    astat = [sp++];
    rti;
isr_uart_tx.end:
```

14 GENERAL-PURPOSE PORTS

This chapter describes the general-purpose ports. Following an overview and a list of key features is a block diagram of the interface and a description of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

Overview

The ADSP-BF534, ADSP-BF536, and ADSP-BF537 Blackfin processors feature a rich set of peripherals, which through a powerful pin multiplexing scheme, provides great flexibility to the external application space.

Table 14-1shows all the peripheral signals that can be accessed off chip. In total, there are signal count of 124 signals on the ADSP-BF534 or 142 signals on the ADSP-BF536 and ADSP-BF537. The ADSP-BF534, ADSP-BF536, and ADSP-BF537 processors feature 60 pins for peripheral purposes from which the rich peripheral set signals are multiplexed through.

Table 14-1. General-Purpose and Special Function Signals

Peripheral	Signals
10/100 Ethernet MAC[1]	MII interface (18) or RMII (11)
CAN 2.0 B Controller	Data (2)
TWI Controller	Data (1), clock (1)
PPI Interface	Data (16), frame sync (3), clock (1)
SPI Interface	Data (2), clock (1), slave select (1), slave enable (7)

Table 14-1. General-Purpose and Special Function Signals (Cont'd)

Peripheral	Signals
SPORTs	Data (8), clock (4), frame sync (4)
UARTs	Data (4)
Timers	PWM/capture/clock (8), alternate clock input (8), alternate capture input (3)
General-Purpose I/O	GPIO (48)
Handshake MemDMA	MemDMA request (2)

1 ADSP-BF536 and ADSP-BF537 only.

Features

The peripheral pins are functionally organized into general-purpose ports designated port F, port G, port H, and port J.

Port F provides 16 pins:

- UART0 and UART1 signals
- Primary timer signals
- Primary SPI signals
- Handshake memDMA request signals
- GPIOs

Port G provides 16 pins:

- SPORT1 signals
- PPI data signals
- GPIOs

Port H provides 16 pins:

- MII/RMII signals (ADSP-BF536 and ADSP-BF537 only)
- GPIOs

Port J provides 12 pins:

- SPORT0 signals
- TWI signals
- CAN signals
- Some alternate timer inputs
- Additional SPI slave select signals
- Two additional MII/RMII pins (ADSP-BF536 and ADSP-BF537 only—on the ADSP-BF534, `PJ0` should be left "No Connect" and `PJ1` should be "Connect to Ground")

Interface Overview

By default all pins of port F, port G, and port H are in general-purpose I/O (GPIO) mode. Port J does not provide GPIO functionality. In this mode, a pin can function as either digital input, digital output, or interrupt input. See "General-Purpose I/O Modules" on page 14-10 for details. Peripheral functionality must be explicitly enabled by the function enable

registers (PORTF_FER, PORTG_FER, and PORTH_FER). The competing peripherals on port F, port G, and port H are controlled by the multiplexer control register (PORT_MUX).

In this chapter, the naming convention for registers and bits uses a lower case x to represent F, G, or H. For example, the name PORTx_FER represents PORTF_FER, PORTG_FER, and PORTH_FER. The bit name PxO represents PF0, PG0, and PH0. This convention is used to discuss registers common to these three ports.

External Interface

The external interface of the general-purpose ports are described in the following sections.

Port F Structure

Figure 14-1 shows the multiplexer scheme for port F. Port F is controlled by the PORT_MUX and the PORTF_FER registers.

Port F provides both UART ports. If only one UART is required in the target application, the user has the option to enable either the handshake memDMA request pins (see "Handshaked Memory DMA Operation" on page 5-40) or two additional timers. Other timers are competing with additional SPI slave select signals and with the rarely used PPI frame sync FS3.

Special attention is required for the use of the timers: with PPI enabled, timer 0 and timer 1 are typically used for PPI frame sync generation. The alternate timer clock input TACLK0 needs to be enabled by the timer 0 module only. It is not gated by any function enable or multiplexer register.

With UART0 enabled by PFDE = 0 in the PORT_MUX register, the UART0 RX pin can also be simultaneously captured by timer 1 through its alternative capture input TACI1 (see Chapter 15, "General-Purpose Timers").

This allows for unique applications like autobaud detection. Similarly, timer 6 can be used to capture UART1 RX through TACI6 if PFTE is cleared. If the PFTE bit is set, timer 6 can still capture the same pin through the regular TMR6 input.

Any GPIO can be enabled individually and overrides the peripheral function if the respective bit in the PORTF_FER is cleared.

The eight pins PF7 - PF0 can drive higher current than all other pins of the processor, regardless whether in GPIO or peripheral function mode. The high current option does not need to be enabled. It is always on. Refer to the part-specific data sheet for further details.

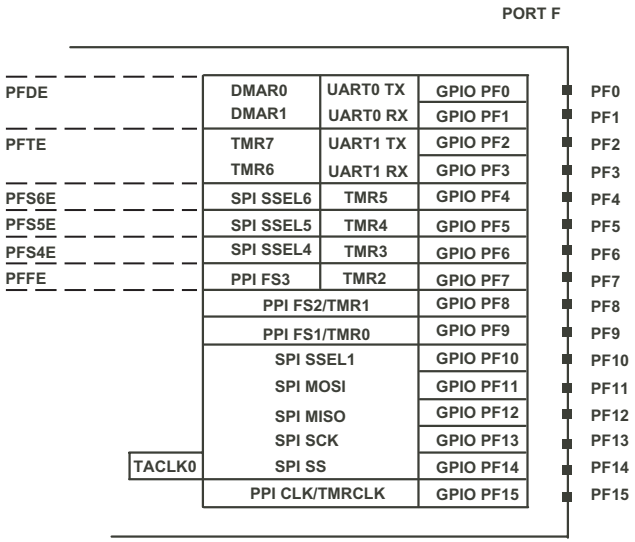


Figure 14-1. Port F Multiplexing Scheme

Port G Structure

Figure 14-2 shows the multiplexer scheme for port G. It is controlled by the PORT_MUX and the PORTG_FER registers.

SPORT1 signals are multiplexed with PPI data signals PPID15-8. Thus, with an 8-bit PPI configuration, no restrictions apply to SPORT1. With a 10-bit PPI configuration, the secondary SPORT1 data signals are not available. However, in a 12-bit PPI configuration, only the SPORT1 transmit channel remains functional.

Any GPIO can be enabled individually and overrides the PPI or SPORT function.

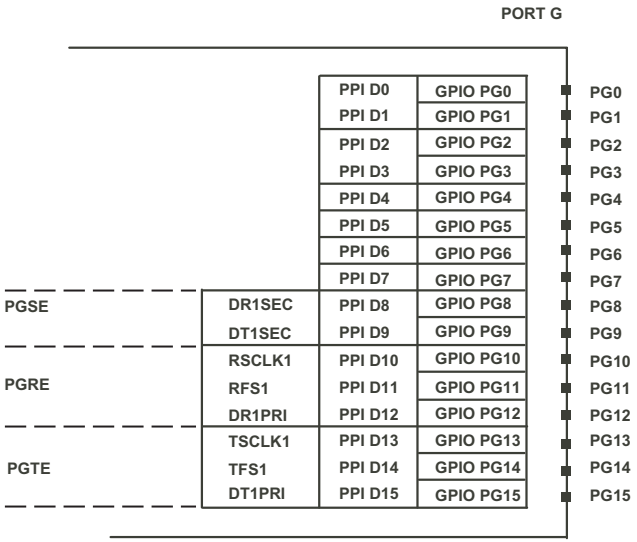


Figure 14-2. Port G Multiplexing Scheme

Port H Structure

Figure 14-3 shows the multiplexer scheme for port H. It is controlled by the PORT_MUX and the PORTH_FER registers. On the ADSP-BF534 processor, port H functions as a GPIO port only.

Port H provides most of the signals of the ADSP-BF536 and ADSP-BF537 MII or alternate RMII interface. Refer to Chapter 8, "Ethernet MAC", for information about how to configure MII versus RMII mode. The three alternate timer capture inputs are not gated by the function enable or multiplexer control registers.

For MII operation, all bits of the PORTH_FER register must be set. For RMII mode, 7 pins can be used in GPIO mode, and PORTH_FER should be written with a value of 0xC373.

ADSP-BF534 programmers should not set any bit of the PORTH_FER register. The PORTH_FER register is reserved on the ADSP-BF534.

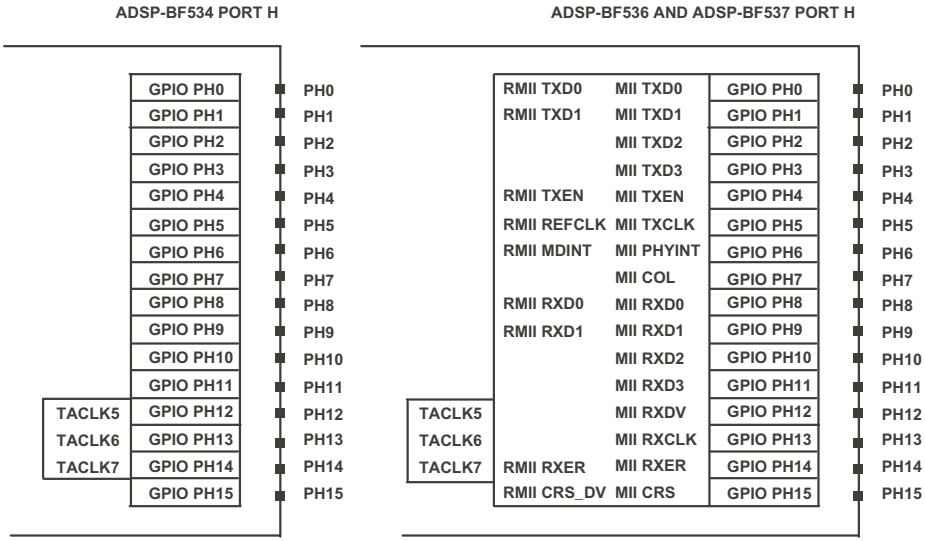


Figure 14-3. Port H Multiplexing Scheme

Port J Structure

Figure 14-4 shows the multiplexer scheme for port J. It is controlled by the PORT_MUX register.

Port J does not provide GPIO functionality. The CAN pins share functionality with the secondary SPORT0 data pins, as well as with SPI slave select SSEL7. With the CAN port active, the SPORT0 can still be used in 6-pin mode. Even with all SPI slave selects enabled, the receive channel SPORT0 is fully functional. The four alternate timer clock inputs TACLK1 to TACLK4 are always functional regardless of any multiplexer control or function enable bit. The two MII pins and the TWI pins are not multiplexed at all.

When the CAN interface is selected by the PJCE bit field and in the PORT_MUX register, timer 0 can capture the CAN RX pin through its TACIO input for autobaud detection.

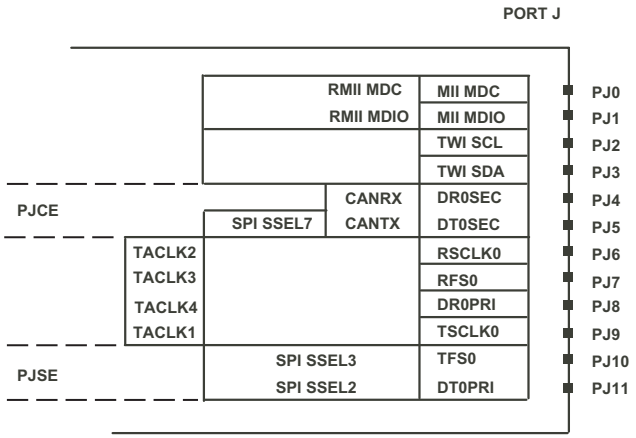


Figure 14-4. Port J Multiplexing Scheme

Internal Interfaces

Port control and GPIO registers are part of the system memory-mapped registers (MMRs). The addresses of the GPIO module MMRs appear in Appendix B. Core access to the GPIO configuration registers is through the system bus.

The PORT_MUX register controls the muxing schemes of port F, port G and port J.

The function enable register (PORTF_FER, PORTG_FER, PORTH_FER) enables the peripheral functionality for each individual pin of port x.

Performance/Throughput

The PFx, PGx, and PHx pins are synchronized to the system clock (SCLK). When configured as outputs, the GPIOs can transition once every system clock cycle.

When configured as inputs, the overall system design should take into account the potential latency between the core and system clocks. Changes in the state of port pins have a latency of 3 SCLK cycles before being detectable by the processor. When configured for level-sensitive interrupt generation, there is a minimum latency of 4 SCLK cycles between the time the signal is asserted on the pin and the time that program flow is interrupted. When configured for edge-sensitive interrupt generation, an additional SCLK cycle of latency is introduced, giving a total latency of 5 SCLK cycles between the time the edge is asserted and the time that the core program flow is interrupted.

Description of Operation

The operation of the general-purpose ports is described in the following sections.

Operation

The GPIO pins on port F, port G, and port H can be controlled individually by the function enable registers (PORTx_FER). With a control bit in these registers cleared, the peripheral function is fully decoupled from the pin. It functions as a GPIO pin only. To drive the pin in GPIO output

mode, set the respective direction bit in the PORTxIO_DIR register. To make the pin a digital input or interrupt input, enable its input driver in the PORTxIO_INEN register.

By default all peripheral pins are configured as inputs after reset. port F, port G, and port H pins are in GPIO mode. However, GPIO input drivers are disabled to minimize power consumption and any need of external pulling resistors.

When the control bit in the function enable registers (PORTx_FER) is set, the pin is set to its peripheral functionality and is no longer controlled by the GPIO module. However, the GPIO module can still sense the state of the pin. When using a particular peripheral interface, pins required for the peripheral must be individually enabled. Keep the related function enable bit cleared if a signal provided by the peripheral is not required by your application. This allows it to be used in GPIO mode.

General-Purpose I/O Modules

The processor supports 48 bidirectional or general-purpose I/O (GPIO) signals. These 48 GPIOs are managed by three different GPIO modules, which are functionally identical. One is associated with port F, one with port G, and one with port H. Every module controls 16 GPIOs available through the pins PF15-0, PG15-0, and PH15-0.

Each GPIO can be individually configured as either an input or an output by using the GPIO direction registers (PORTxIO_DIR).

When configured as output, the GPIO data registers (PORTFIO, PORTGIO, and PORTHIO) can be directly written to specify the state of the GPIOs.

The GPIO direction registers are read-write registers with each bit position corresponding to a particular GPIO. A logic 1 configures a GPIO as an output, driving the state contained in the GPIO data register if the peripheral function is not enabled by the function enable registers. A logic 0 configures a GPIO as an input.

Note when using the GPIO as an input, the corresponding bit should also be set in the GPIO input enable register. Otherwise, changes at the input pins will not be recognized by the processor.

The GPIO input enable registers (PORTFIO_INEN, PORTGIO_INEN, and PORTHIO_INEN) are used to enable the input buffers on any GPIO that is being used as an input. Leaving the input buffer disabled eliminates the need for pull-ups and pull-downs when a particular PFx, PGx, or PHx pin is not used in the system. By default, the input buffers are disabled.

Once the input driver of a GPIO pin is enabled, the GPIO is not allowed to operate as an output anymore. Never enable the input driver (by setting PORTxIO_INEN bits) and the output driver (by setting PORTxIO_DIR bits) for the same GPIO.

A write operation to any of the GPIO data registers sets the value of all GPIOs in this port that are configured as outputs. GPIOs configured as inputs ignore the written value. A read operation returns the state of the GPIOs defined as outputs and the sense of the inputs, based on the polarity and sensitivity settings, if their input buffers are enabled. Table 14-2 helps to interpret read values in GPIO mode, based on the settings of the PORTxIO_POLAR, PORTxIO_EDGE, and PORTxIO_BOTH registers.

Table 14-2. GPIO Value Register Pin Interpretation

POLAR	EDGE	BOTH	Effect of MMR Settings
0	0	X	Pin that is high reads as 1; pin that is low reads as 0
0	1	0	If rising edge occurred, pin reads as 1; otherwise, pin reads as 0
1	0	X	Pin that is low reads as 1; pin that is high reads as 0
1	1	0	If falling edge occurred, pin reads as 1; otherwise, pin reads as 0
X	1	1	If any edge occurred, pin reads as 1; otherwise, pin reads as 0

For GPIOs configured as edge-sensitive, a readback of 1 from one of these registers is sticky. That is, once it is set it remains set until cleared by user code. For level-sensitive GPIOs, the pin state is checked every cycle, so the readback value will change when the original level on the pin changes.

The state of the output is reflected on the associated pin only if the function enable bit in the `PORTx_FER` register is cleared.

Write operations to the GPIO data registers modify the state of all GPIOs of a port. In cases where only one or a few GPIOs need to be changed, the user may write to the GPIO set registers, `PORTxIO_SET`, the GPIO clear registers, `PORTxIO_CLEAR`, or to the GPIO toggle registers, `PORTxIO_TOGGLE` instead.

While a direct write to a GPIO data register alters all bits in the register, writes to a GPIO set register can be used to set a single or a few bits only. No read-modify-write operations are required. The GPIO set registers are write-1-to-set registers. All 1s contained in the value written to a GPIO set register sets the respective bits in the GPIO data register. The 0s have no effect. For example, assume that `PF0` is configured as an output. Writing

0x0001 to the GPIO set register drives a logic 1 on the PF0 pin without affecting the state of any other PFx pins. The GPIO set registers are typically also used to generate GPIO interrupts by software. Read operations from the GPIO set registers return the content of the GPIO data registers.

The GPIO clear registers provide an alternative port to manipulate the GPIO data registers. While a direct write to a GPIO data register alters all bits in the register, writes to a GPIO clear register can be used to clear individual bits only. No read-modify-write operations are required. The clear registers are write-1-to-clear registers. All 1s contained in the value written to the GPIO clear register clears the respective bits in the GPIO data register. The 0s have no effect. For example, assume that PF4 and PF5 are configured as outputs. Writing 0x0030 to the PORTFIO_C clear register drives a logic 0 on the PF4 and PF5 pins without affecting the state of any other PFx pins.

If an edge-sensitive pin generates an interrupt request, the service routine must acknowledge the request by clearing the respective GPIO latch. This is usually performed through the clear registers.

Read operations from the GPIO clear registers return the content of the GPIO data registers.

The GPIO toggle registers provide an alternative port to manipulate the GPIO data registers. While a direct write to a GPIO data register alters all bits in the register, writes to a toggle register can be used to toggle individual bits. No read-modify-write operations are required. The GPIO toggle registers are write-1-to-toggle registers. All 1s contained in the value written to a GPIO toggle register toggle the respective bits in the GPIO data register. The 0s have no effect. For example, assume that PG1 is configured as an output. Writing 0x0002 to the PORTGIO_T toggle register changes the pin state (from logic 0 to logic 1, or from logic 1 to logic 0) on the PG1 pin without affecting the state of any other PGx pins. Read operations from the GPIO toggle registers return the content of the GPIO data registers.

The state of the GPIOs can be read through any of these data, set, clear, or toggle registers. However, the returned value reflects the state of the input pin only if the proper input enable bit in the PORTxIO_INEN register is set. Note that GPIOs can still sense the state of the pin when the function enable bits in the PORTx_FER registers are set.

Since function enable registers and GPIO input enable registers reset to zero, no external pull-ups or pull-downs are required on the unused pins of port F, port G, and port H.

GPIO Interrupt Processing

Each GPIO can be configured to generate an interrupt. The processor can sense up to 48 asynchronous off-chip signals, requesting interrupts through five interrupt channels. To make a pin function as an interrupt pin, the associated input enable bit in the PORTxIO_INEN register must be set. The function enable bit in the PORTx_FER register is typically cleared. Then, an interrupt request can be generated according to the state of the pin (either high or low), an edge transition (low to high or high to low), or on both edge transitions (low to high and high to low). Input sensitivity is defined on a per-bit basis by the GPIO polarity registers (PORTFIO_POLAR, PORTGIO_POLAR, and PORTHIO_POLAR), and the GPIO interrupt sensitivity registers (PORTFIO_EDGE, PORTGIO_EDGE, and PORTHIO_EDGE). If configured for edge sensitivity, the GPIO set on both edges registers (PORTFIO_BOTH, PORTGIO_BOTH, and PORTHIO_BOTH) let the interrupt request generate on both edges.

The GPIO polarity registers are used to configure the polarity of the GPIO input source. To select active high or rising edge, set the bits in the GPIO polarity register to 0. To select active low or falling edge, set the bits in the GPIO polarity register to 1. This register has no effect on GPIOs that are defined as outputs. The contents of the GPIO polarity registers are cleared at reset, defaulting to active high polarity.

The GPIO interrupt sensitivity registers are used to configure each of the inputs as either a level-sensitive or an edge-sensitive source. When using an edge-sensitive mode, an edge detection circuit is used to prevent a situation where a short event is missed because of the system clock rate. The GPIO interrupt sensitivity register has no effect on GPIOs that are defined as outputs. The contents of the GPIO interrupt sensitivity registers are cleared at reset, defaulting to level sensitivity.

The GPIO set on both edges registers are used to enable interrupt generation on both rising and falling edges. When a given GPIO has been set to edge-sensitive in the GPIO interrupt sensitivity register, setting the respective bit in the GPIO set on both edges register to both edges results in an interrupt being generated on both the rising and falling edges. This register has no effect on GPIOs that are defined as level-sensitive or as outputs. See Table 14-2 on page 14-12 for information on how the GPIO set on both edges register interacts with the GPIO polarity and GPIO interrupt sensitivity registers.

When the GPIO's input drivers are enabled while the GPIO direction registers configure it as an output, software can trigger a GPIO interrupt by writing to the data/set/toggle registers. The interrupt service routine should clear the GPIO to acknowledge the request.

Each of the three GPIO modules provides two independent interrupt channels. Identical in functionality, these are called interrupt A and interrupt B. Both interrupt channels have their own mask register which lets you assign the individual GPIOs to none, either, or both interrupt channels.

Since all mask registers reset to zero, none of the GPIOs is assigned any interrupt by default. Each GPIO represents a bit in each of these registers. Setting a bit means enabling the interrupt on this channel.

Interrupt A and interrupt B operate independently. For example, writing 1 to a bit in the mask interrupt A register does not affect interrupt channel B. This facility allows GPIOs to generate GPIO interrupt A, GPIO interrupt B, both GPIO interrupts A and B, or neither.

A GPIO interrupt is generated by a logical OR of all unmasked GPIOs for that interrupt. For example, if PF0 and PF1 are both unmasked for GPIO interrupt channel A, GPIO interrupt A will be generated when triggered by PF0 or PF1. The interrupt service routine must evaluate the GPIO data register to determine the signaling interrupt source. Note that interrupt channel A of port F and interrupt channel A of port G are ORed at system level as shown in Figure 14-5.

When using either rising or falling edge-triggered interrupts, the interrupt condition must be cleared each time a corresponding interrupt is serviced by writing 1 to the appropriate bit in the GPIO clear register.

At reset, all interrupts are masked and disabled.

Similarly to the GPIOs themselves, the mask register can either be written through the GPIO mask data registers (PORTxIO_MASKA, PORTxIO_MASKB) or be controlled by the mask A/mask B set, clear and toggle registers.

The GPIO mask interrupt set registers (PORTxIO_MASKA_SET, PORTxIO_MASKB_SET) provide an alternative port to manipulate the GPIO mask interrupt registers. While a direct write to a mask interrupt register alters all bits in the register, writes to a mask interrupt set register can be used to set a single or a few bits only. No read-modify-write operations are required.

The mask interrupt set registers are write-1-to-set registers. All ones contained in the value written to the mask interrupt set register set the respective bits in the mask interrupt register. The zeroes have no effect. Writing a one to any bit enables the interrupt for the respective GPIO.

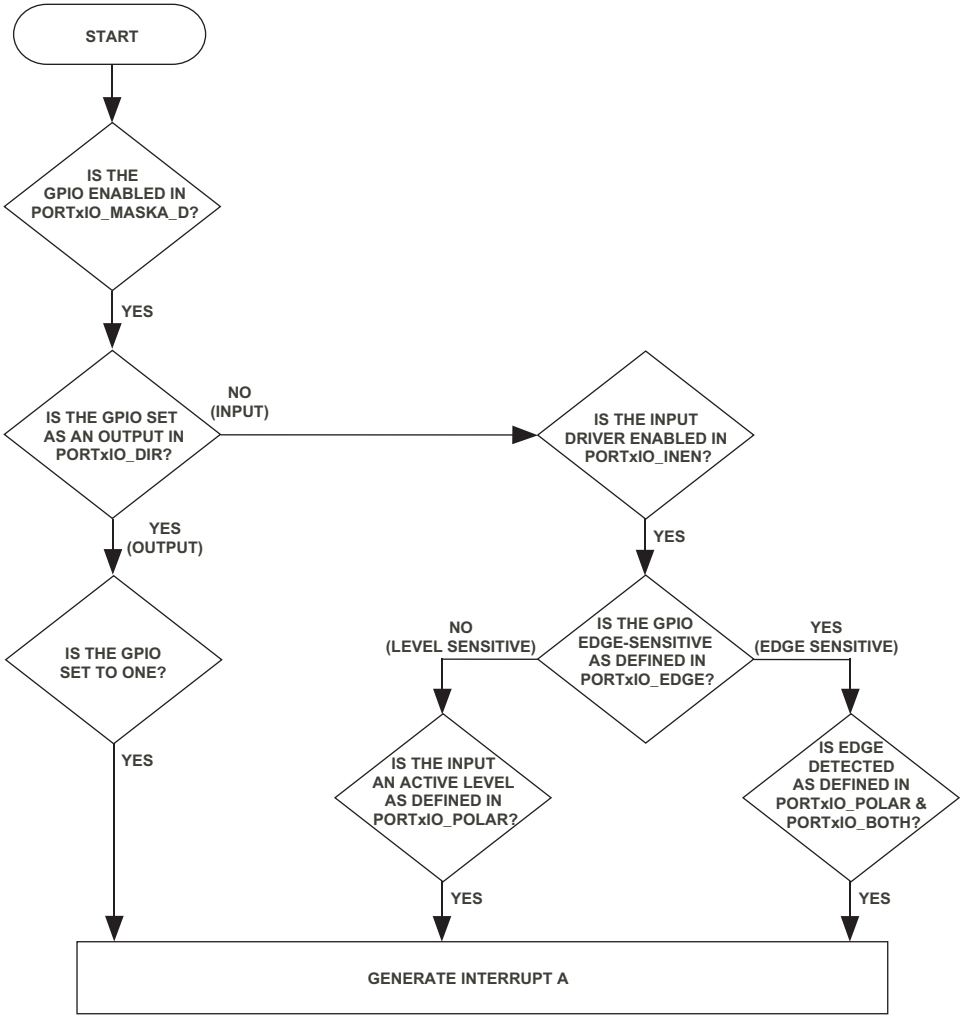


Figure 14-5. GPIO Interrupt Generation Flow for Interrupt Channel A

The GPIO mask interrupt clear registers (`PORTxIO_MASKA_CLEAR`, `PORTxIO_MASKB_CLEAR`) provide an alternative port to manipulate the GPIO mask interrupt registers. While a direct write to a mask interrupt

register alters all bits in the register, writes to the mask interrupt clear register can be used to clear a single bit or a few bits only. No read-modify-write operations are required.

The mask interrupt clear registers are write-1-to-clear registers. All ones contained in the value written to the mask interrupt clear register clear the respective bits in the mask interrupt register. The zeroes have no effect.Writing a one to any bit disables the interrupt for the respective GPIO.

The GPIO mask interrupt toggle registers (PORTxIO_MASKA_TOGGLE, PORTxIO_MASKB_TOGGLE) provide an alternative port to manipulate the GPIO mask interrupt registers. While a direct write to a mask interrupt register alters all bits in the register, writes to a mask interrupt toggle register can be used to toggle a single bit or a few bits only. No read-modify-write operations are required.

The mask interrupt toggle registers are write-1-to-clear registers. All ones contained in the value written to the mask interrupt toggle register toggle the respective bits in the mask interrupt register. The zeroes have no effect. Writing a one to any bit toggles the interrupt for the respective GPIO.

Figure 14-5 illustrated the interrupt flow of any GPIO module's interrupt A channel. The interrupt B channel behaves identically.

All GPIOs assigned to the same interrupt channel are ORed. If multiple GPIOs are assigned to the same interrupt channel, it is up to the interrupt service routine to evaluate the GPIO data registers to determine the signaling interrupt source.

Although each GPIO module provides two independent interrupt channels, the interrupt A channels of port F and port G are ORed as shown in Figure 14-6. The total number of GPIO interrupt channels is five, therefore.

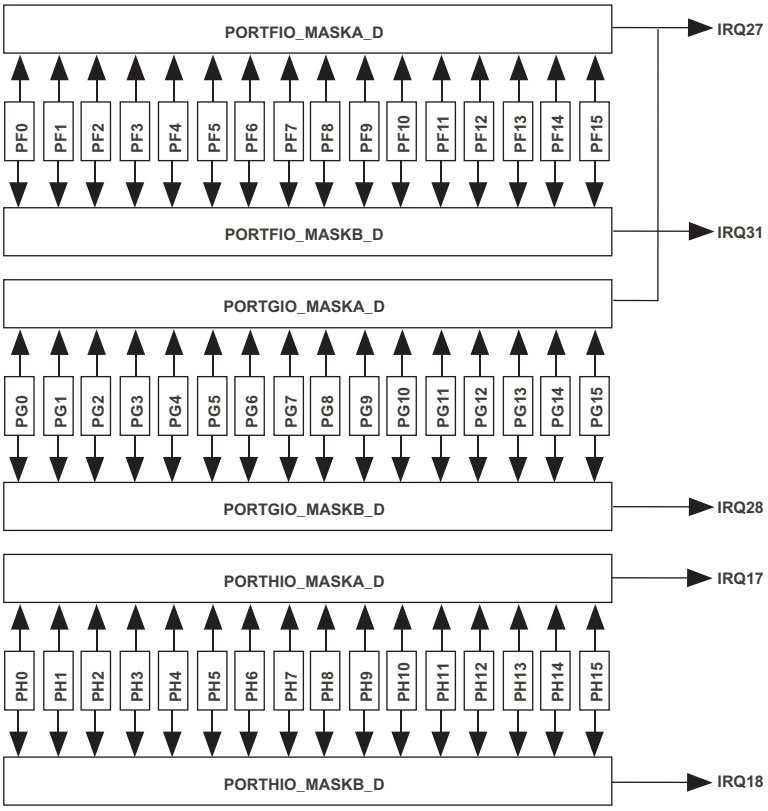


Figure 14-6. GPIO Interrupt Channels

Programming Model

Figure 14-7 and Figure 14-8 show the programming model for the general-purpose ports.

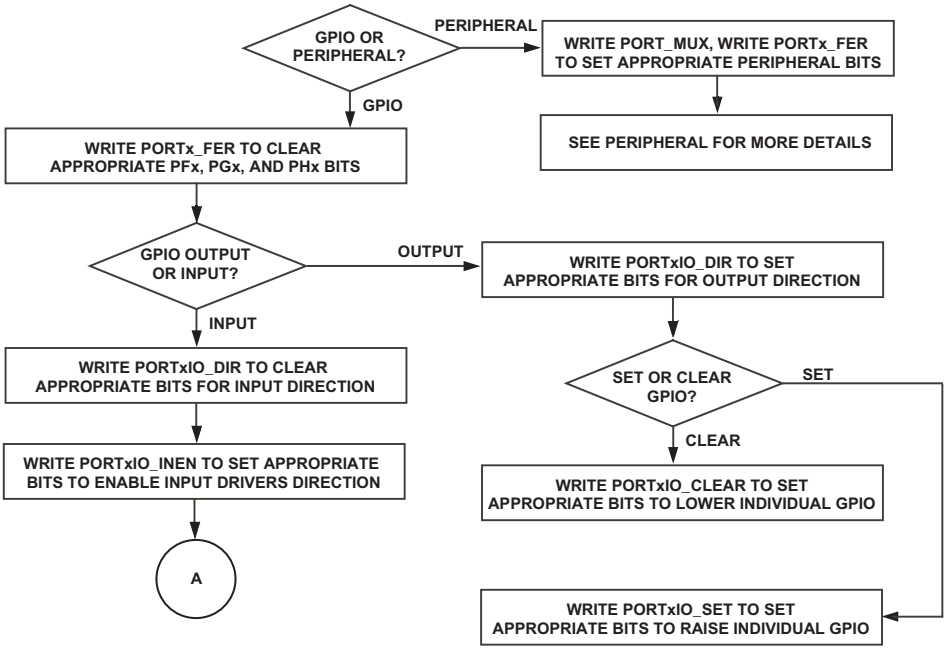


Figure 14-7. GPIO Flow Chart (Part 1 of 2)

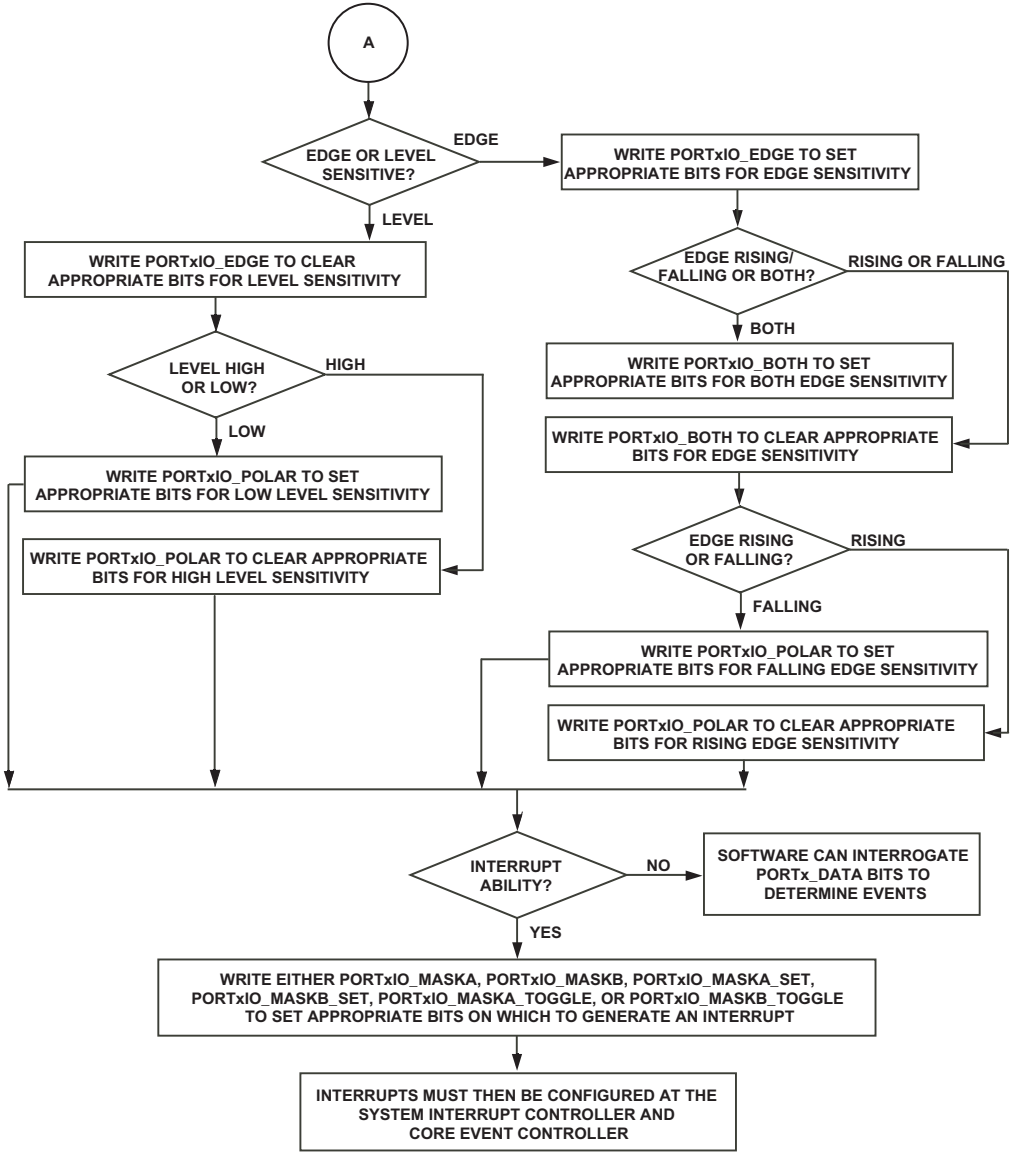


Figure 14-8. GPIO Flow Chart (Part 2 of 2)

Memory-Mapped GPIO Registers

The GPIO registers are part of the system memory-mapped registers (MMRs). Figure 14-9 through Figure 14-27 on page 14-34 illustrate the GPIO registers. The addresses of the programmable flag MMRs appear in Appendix B.

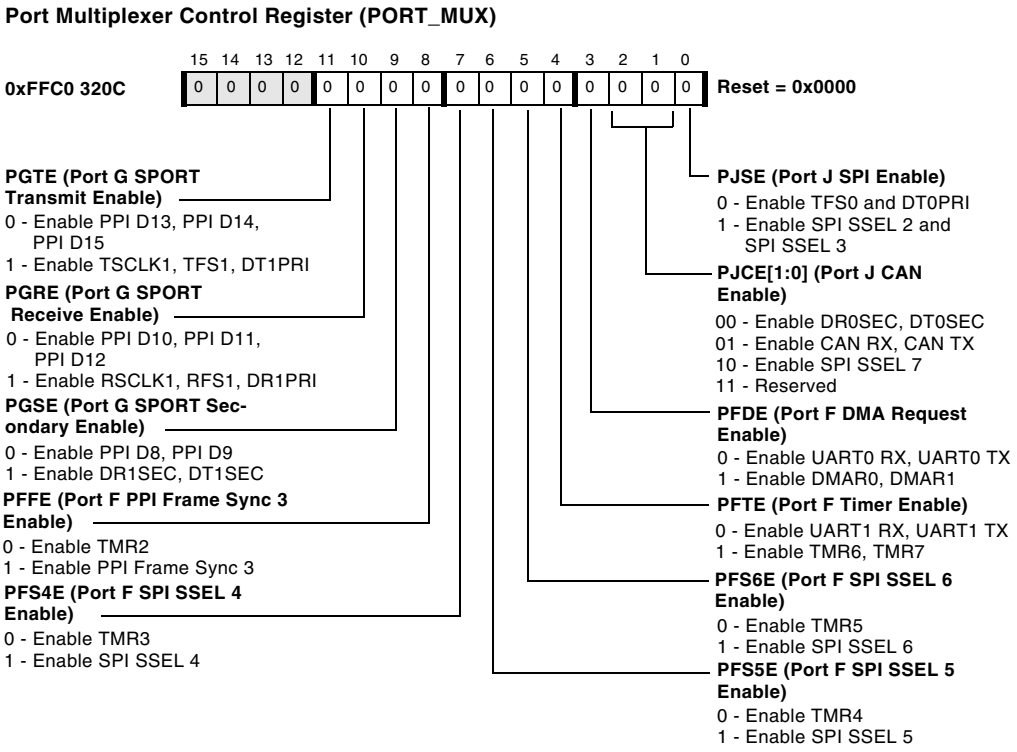


Figure 14-9. Port Multiplexer Control Register

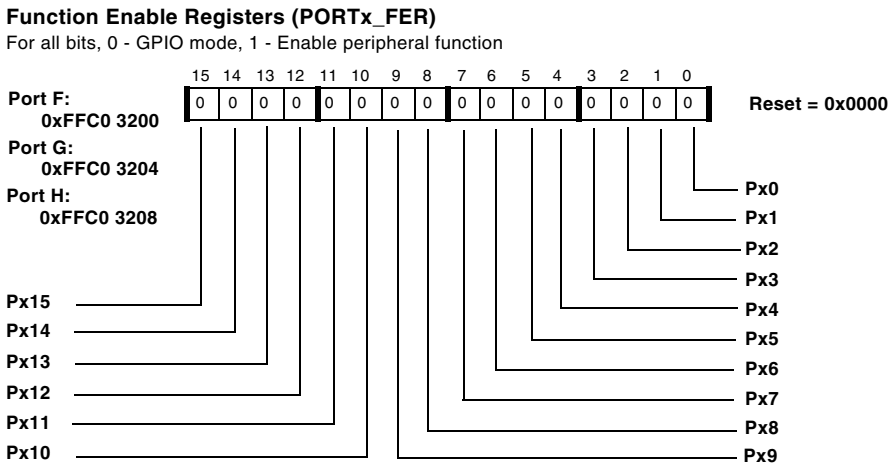


Figure 14-10. Function Enable Registers

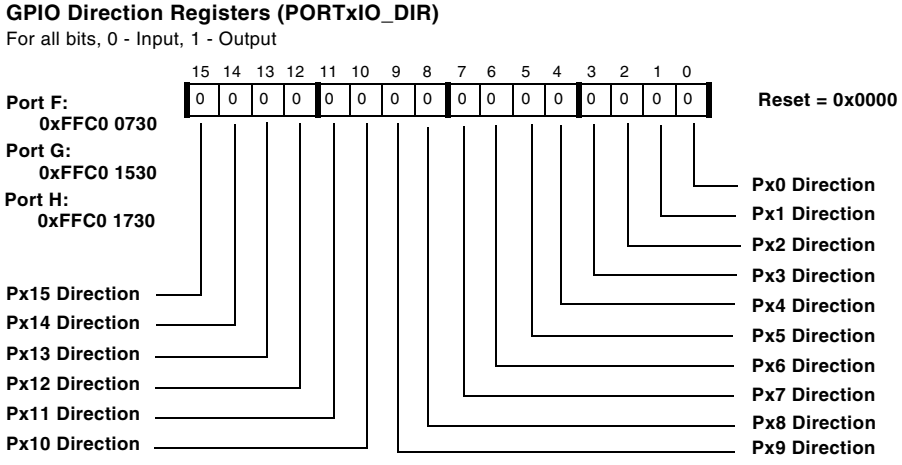


Figure 14-11. GPIO Direction Registers

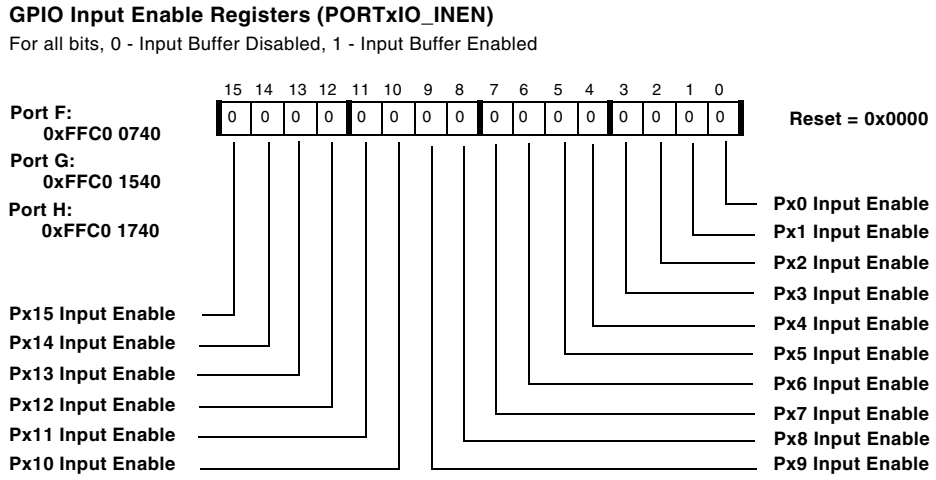


Figure 14-12. GPIO Input Enable Registers

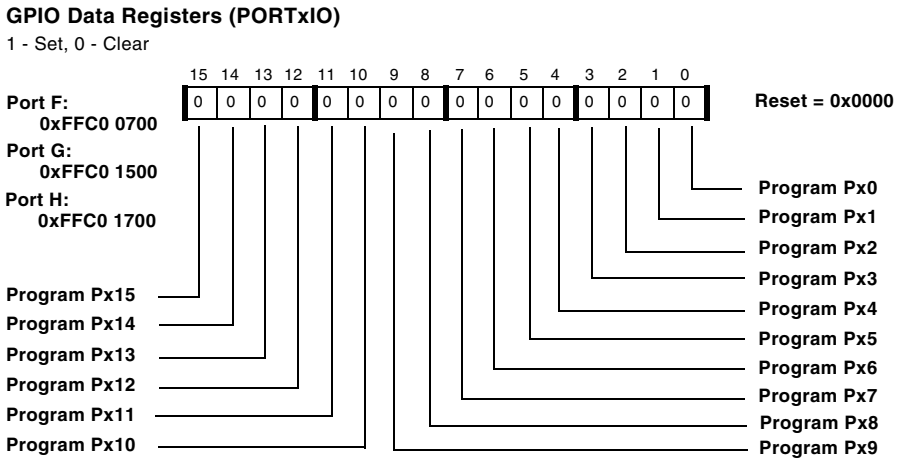


Figure 14-13. GPIO Data Registers

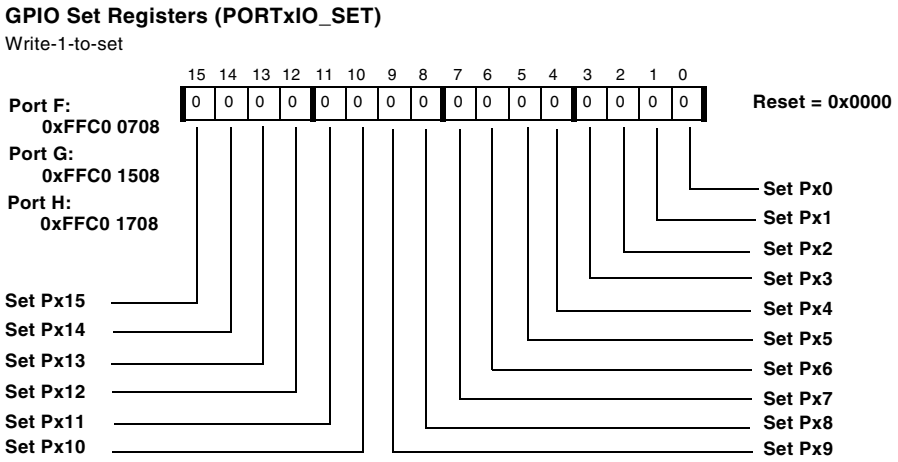


Figure 14-14. GPIO Set Registers

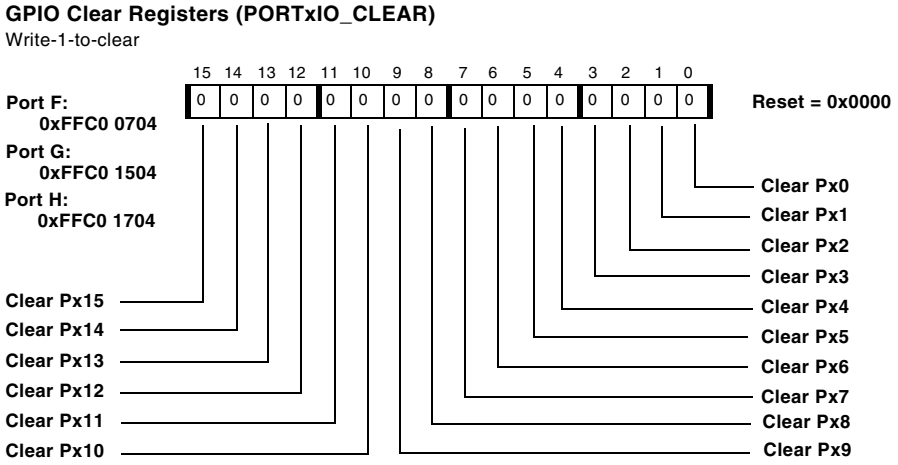


Figure 14-15. GPIO Clear Registers

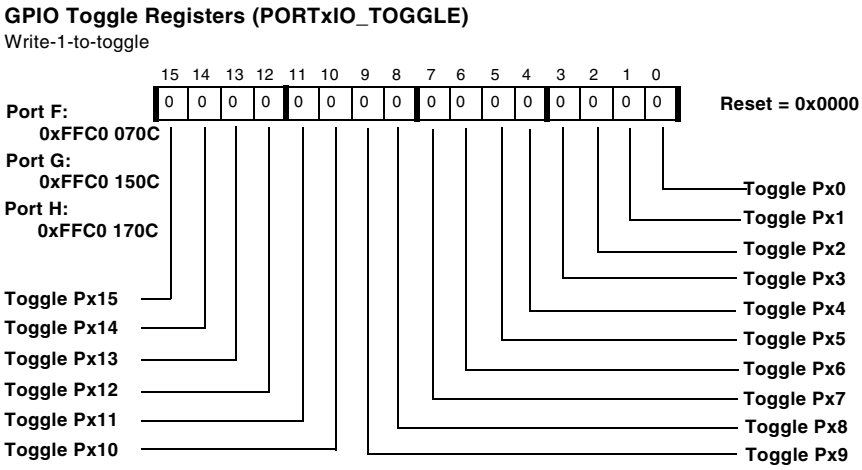


Figure 14-16. GPIO Toggle Registers

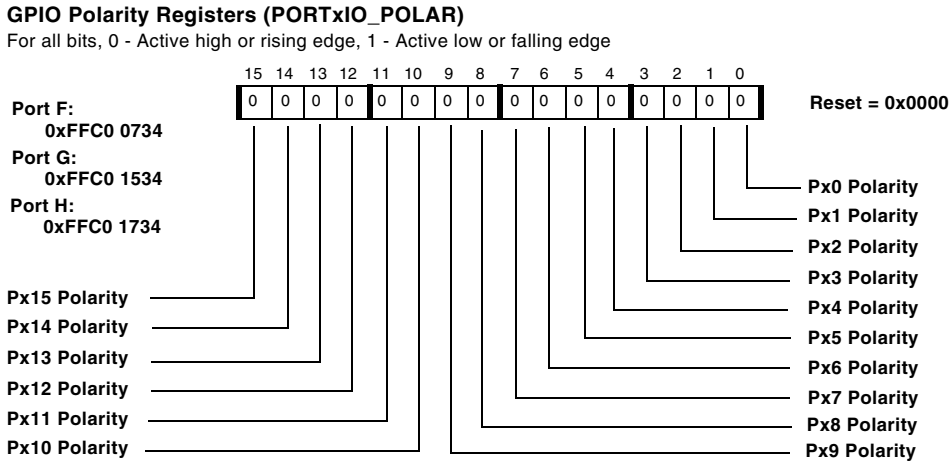


Figure 14-17. GPIO Polarity Registers

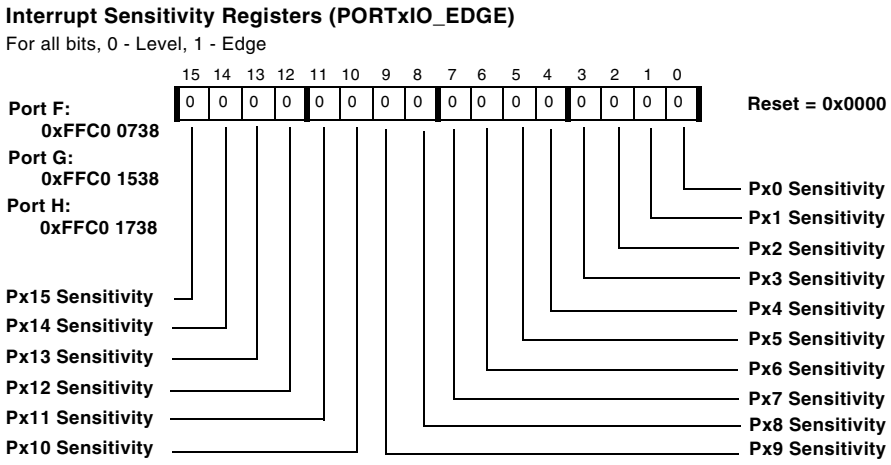


Figure 14-18. Interrupt Sensitivity Registers

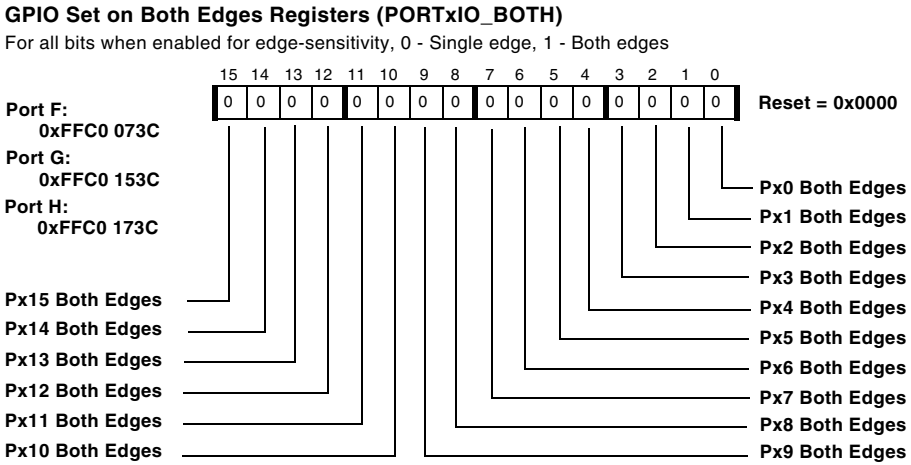


Figure 14-19. GPIO Set on Both Edges Registers

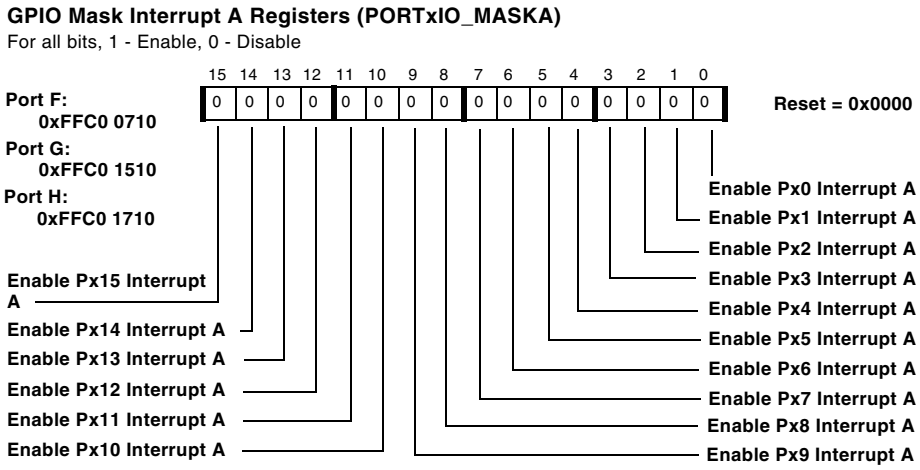


Figure 14-20. GPIO Mask Interrupt A Registers

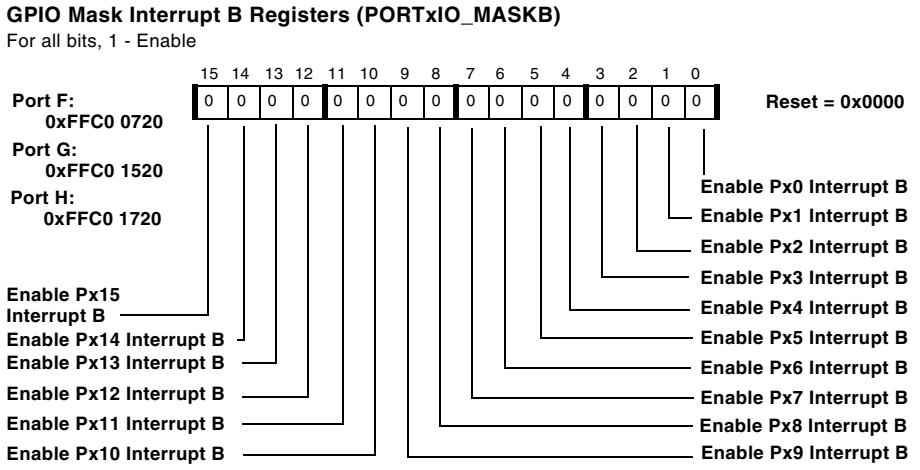


Figure 14-21. GPIO Mask Interrupt B Registers

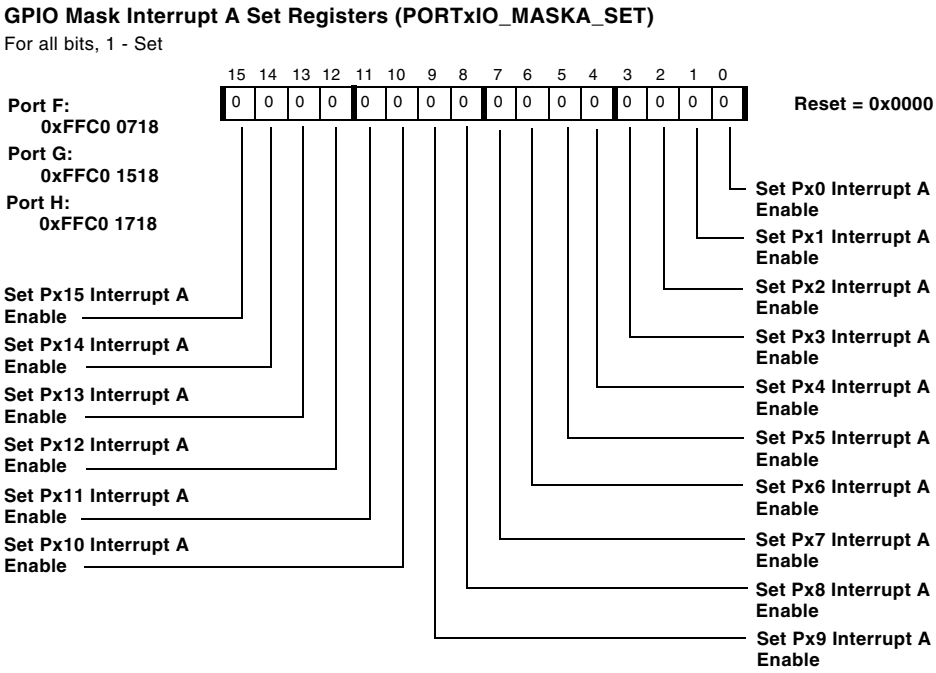


Figure 14-22. GPIO Mask Interrupt A Set Registers

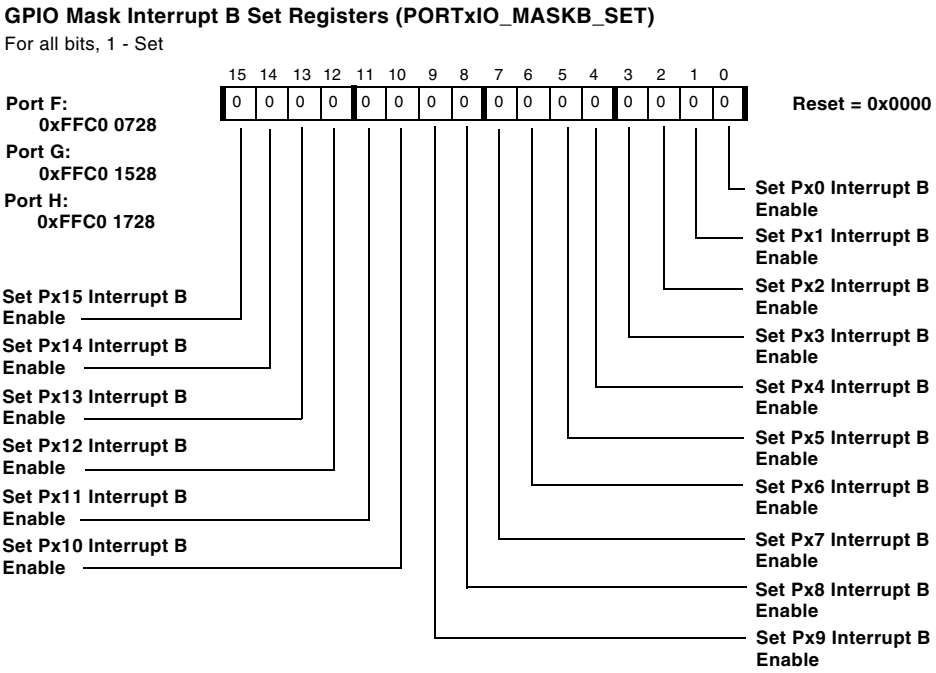


Figure 14-23. GPIO Mask Interrupt B Set Registers

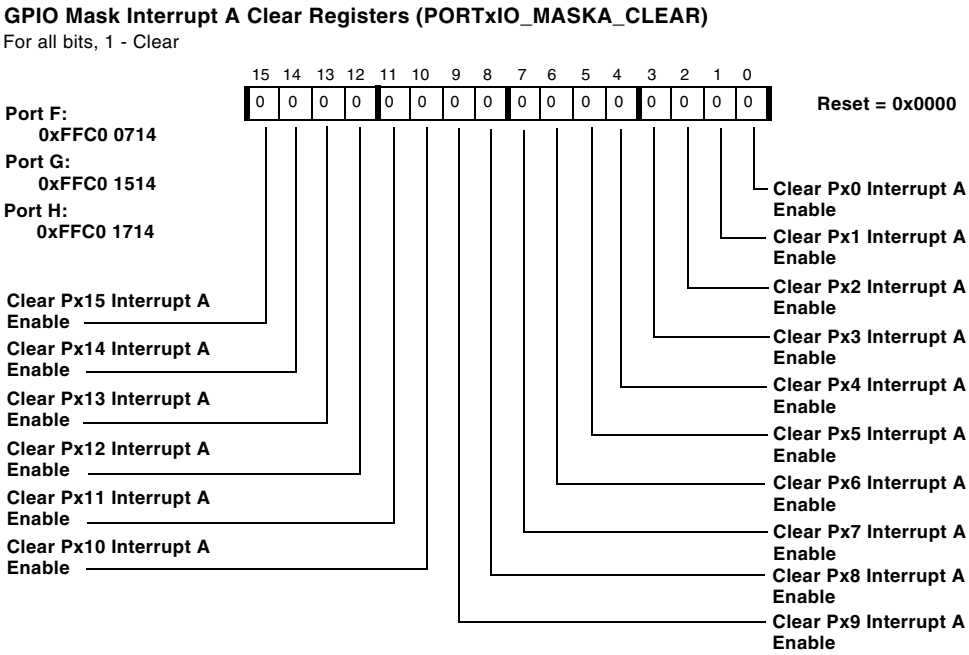


Figure 14-24. GPIO Mask Interrupt A Clear Registers

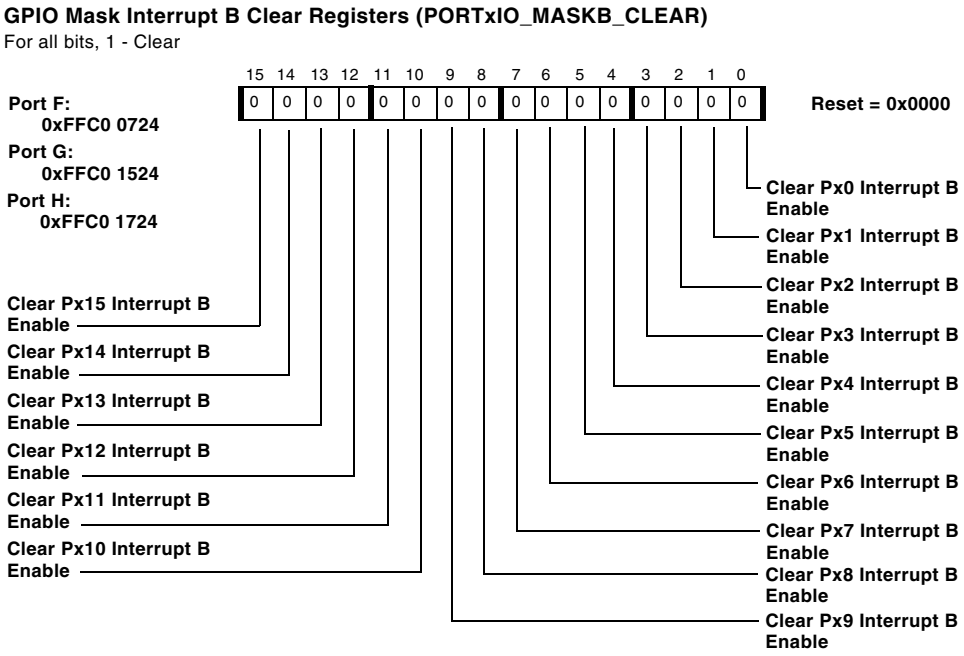


Figure 14-25. GPIO Mask Interrupt B Clear Registers

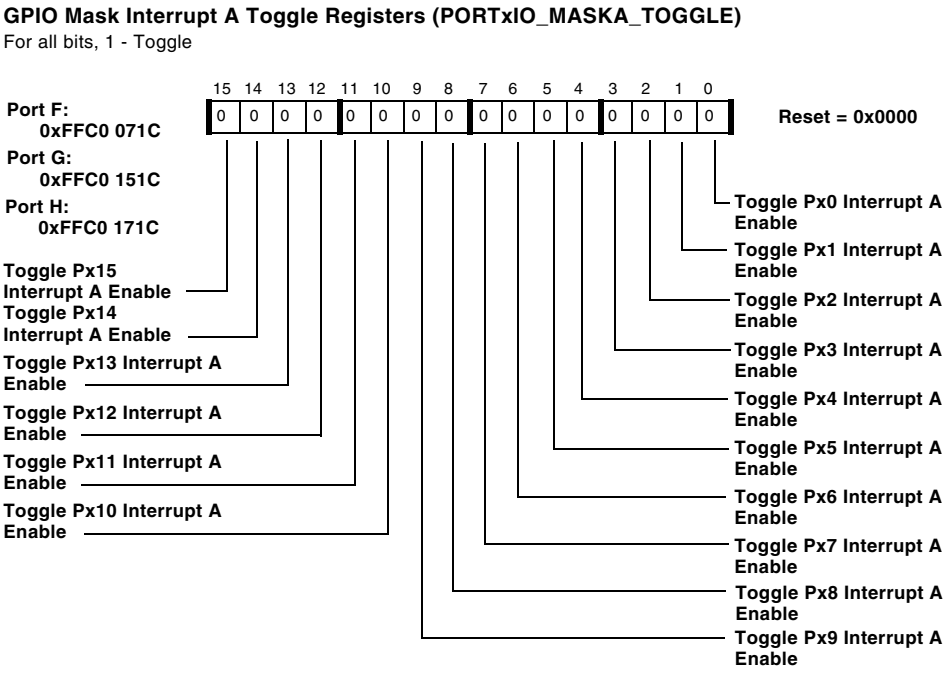


Figure 14-26. GPIO Mask Interrupt A Toggle Registers

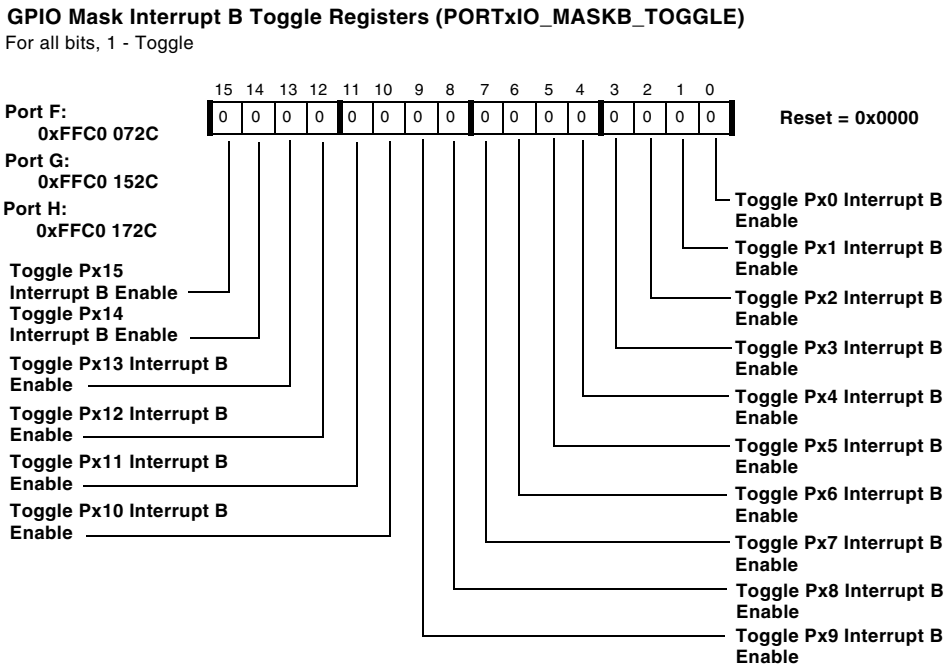


Figure 14-27. GPIO Mask Interrupt B Toggle Registers

Programming Examples

Listing 14-1 provides examples for using the general-purpose ports.

Listing 14-1. General-Purpose Ports

```
/* set port f function enable register to GPIO (not peripheral)
*/
p0.l = lo(PORTF_FER);
p0.h = hi(PORTF_FER);

R0.h = 0x0000;
```

```
r0.l = 0x0000;
w[p0] = r0;

/* set port f direction register to enable some GPIO as output,
remaining are input */
p0.l = lo(PORTFIO_DIR);
p0.h = hi(PORTFIO_DIR);
r0.h = 0x0000;
r0.l = 0x0FC0;
w[p0] = r0;
ssync;

/* set port f clear register */
p0.l = lo(PORTFIO_CLEAR);
p0.h = hi(PORTFIO_CLEAR);
    r0.l = 0xFC0;
    w[p0] = r0;
    ssync;

/* set port f input enable register to enable input drivers of
some GPIOs */
p0.l = lo(PORTFIO_INEN);
p0.h = hi(PORTFIO_INEN);
r0.h = 0x0000;
r0.l = 0x003C;
w[p0] = r0;
ssync;

/* set port f polarity register */
p0.l = lo(PORTFIO_POLAR);
p0.h = hi(PORTFIO_POLAR);
r0 = 0x00000;
w[p0] = r0;
ssync;
```

15 GENERAL-PURPOSE TIMERS

This chapter describes the general-purpose timer module. Following an overview and a list of key features is a description of operation and functional modes of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

Overview and Features

The processor features one general-purpose timer module that contains eight identical 32-bit 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.

Features

The general-purpose timers support the following operating modes:

- Single-shot 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 are:

- Synchronous operation of all timers
- Consistent management of period and pulse width values
- Interaction with PPI module for video frame sync operation
- Autobaud detection for CAN and both UART modules
- 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 15-1 shows the derivative-specific block diagram of the general-purpose timer module.

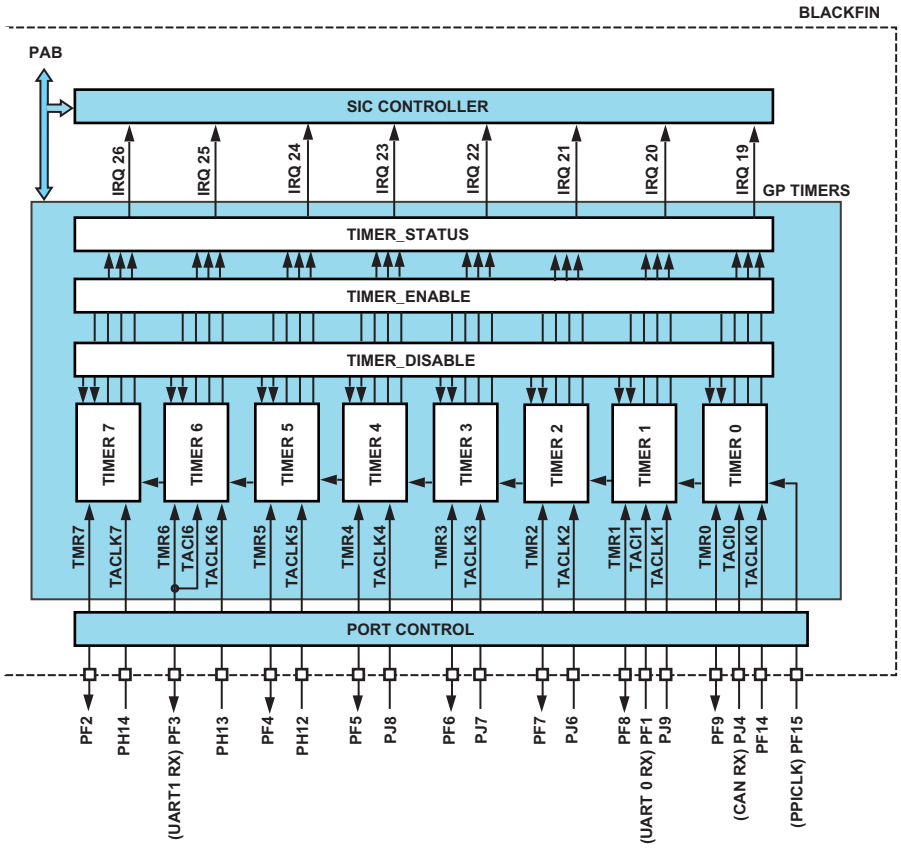


Figure 15-1. Timer Block Diagram

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 15-2, which shows the details of timer 0 representatively. The other timers have identical structure.

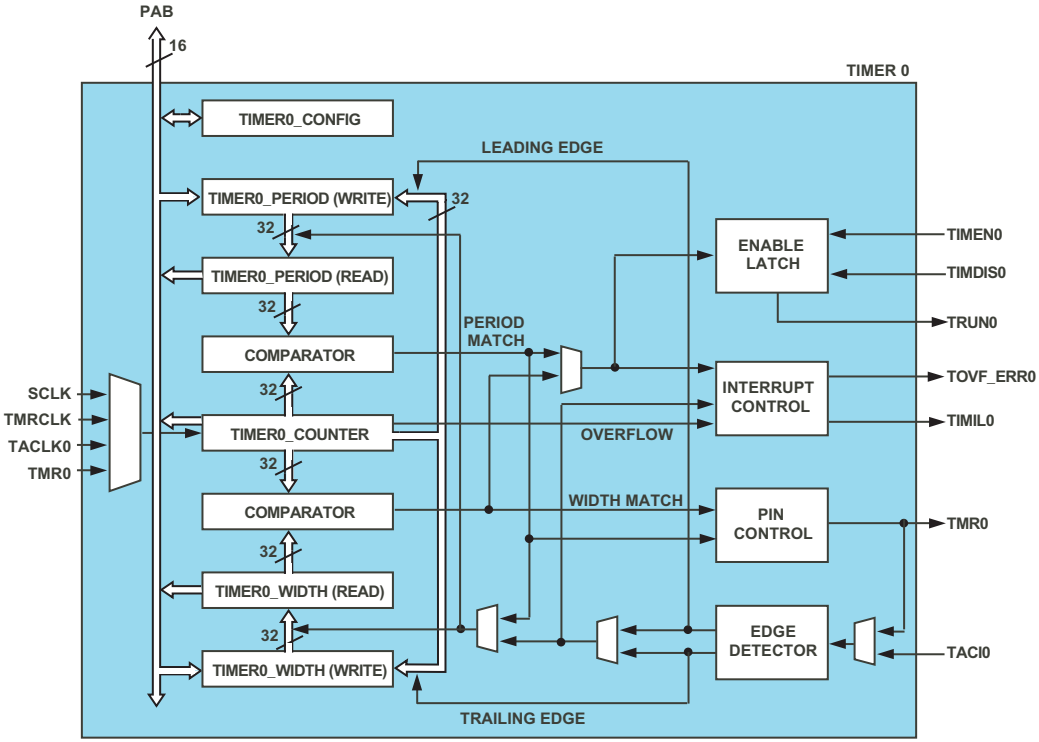


Figure 15-2. Internal Timer Structure

External Interface

Every timer has a dedicated TMRx pin that can be found on port F. If enabled, the TMRx 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.

The timer outputs TMR2 to TMR7 connect to pin drivers that can source and sink higher current than others. See the product data sheet for details.

Alternate clock (TACLKx) and capture (TACIx) inputs are found on port F, port H and port J. The TACLKx pins can alternatively clock the timers in PWM_OUT mode.

In WDTH_CAP mode, timer 0, timer 1, and timer 6 feature TACIx inputs that can be used for bit rate detection on CAN and UART inputs. The TACI0 pin connects to the CAN RX input, the TACI1 pin connects to the UART0 RX input, and the TACI6 pin connects to the UART1 RX input. The TACI2, TACI3, TACI4, TACI5, and TACI7 pins are not used.

The TMRCLK input is another clock input common to all eight timers. The PPI unit is clocked by the same pin; therefore any of the timers can be clocked by PPI_CLK. Since timer 0 and timer 1 are often used in conjunction with the PPI, they are internally looped back to the PPI module for frame sync generation.

In order to enable TMRCLK, PORTF_FER bit 15 must be set and input enable for GPIO bit 15 needs to be set in the PORTxIO_INEN register.

The timer signals TMR0, TMR1 and TMR2 are multiplexed with the PPI frame syncs when the frame syncs are applied externally. PPI modes requiring only two frame syncs free up TMR2 for any purpose. Similarly, PPI modes requiring only one frame sync free up TMR1. For details, see Chapter 7, “Parallel Peripheral Interface”.

If the PPI frame syncs are applied externally, timer 0, timer 1, and timer 2 are still fully functional and can be used for other purposes not involving the three TMRx pins. Timer 0 and timer 1 must not drive their TMR0 and TMR1 pins. If operating in PWM_OUT mode, the OUT_DIS bit in the TIMER0_CONFIG and TIMER1_CONFIG registers must be set.

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, therefore.

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. In total the module has eight interrupt outputs, therefore.

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 case, this is reported by the overflow/error bit TOVF_ERRx in the global timer status (TIMER_STATUS) register. 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_STATUS 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 W1C operations to clear the interrupt request. The global TIMER_STATUS is 32-bits wide. A single atomic 32-bit read can report the status of all eight timers consistently.

Before a timer can be enabled, its mode of operation is programmed in the individual timer-specific TIMERx_CONFIG registers. Then, the timers are started by writing a 1 to the representative bits in the global TIMER_ENABLE register.

The timer enable (TIMER_ENABLE) register can be used to enable all eight timers simultaneously. The register contains eight "write-1-to-set" control bits, one for each timer. Correspondingly, the timer disable (TIMER_DISABLE) register contains eight "write-1-to-clear" control bits to allow simultaneous or independent disabling of the eight timers. Either the timer enable or the timer disable register can be read back to check the enable status of the timers. A 1 indicates that the corresponding timer is enabled. The timer starts counting three 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 eight timers can generate a single interrupt. The eight resulting interrupt signals are routed to the system interrupt controller block for prioritization and masking. The timer status (TIMER_STATUS) register latches the timer interrupts to provide a means for software to determine the interrupt source.

To enable interrupt generation, set the IRQ_ENA bit and unmask the interrupt source in the IMASK and SIC_IMASK 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 by writing 0x000F 000F to the TIMER_STATUS register.

If interrupts are enabled, make sure that the interrupt service routine (ISR) clears the TIMILx bit in the TIMER_STATUS register before the RTI instruction executes. This ensures that the interrupt is not reissued. Remember that writes to system registers are delayed. If only a few instructions separate the TIMILx clear command from the RTI instruction, an extra SSYNC instruction may be inserted. In EXT_CLK mode, reset the

TIMILx bit in the TIMER_STATUS register at the very beginning of the interrupt service routine to avoid missing any timer events. Figure 15-3 shows the timers interrupt structure.

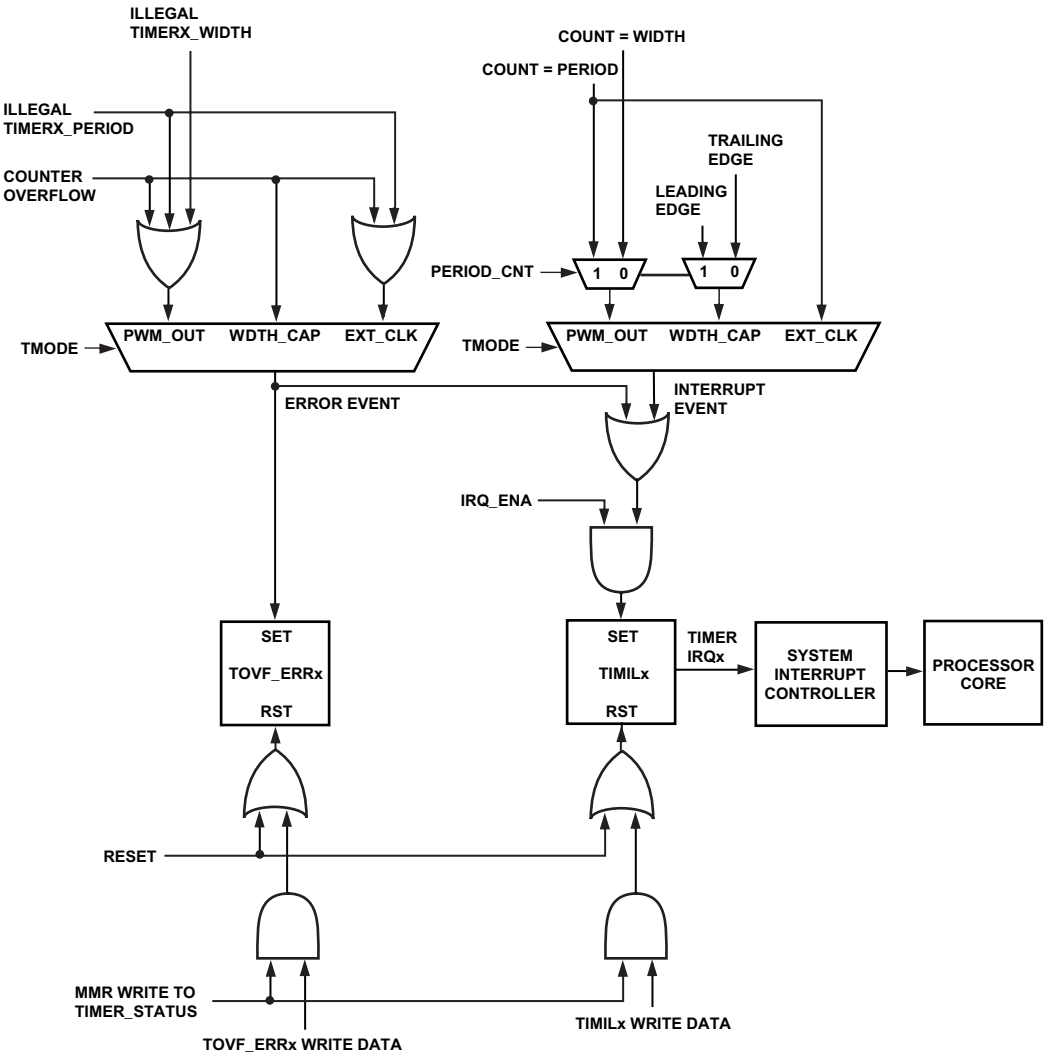


Figure 15-3. Timers Interrupt Structure

Illegal States

Every timer features an error detection circuit. It handles overflow situations but also performs pulse width vs. period plausibility checks. Errors are reported by the TOVF_ERRx bits in the TIMER_STATUS register and the ERR_TYP bit field in the individual TIMERx_CONFIG registers. Table 15-1 provides a summary of error conditions, by using these terms:

- **Startup**. The first clock period during which the timer counter is running after the timer is enabled by writing TIMER_ENABLE.
- **Rollover**. The time when the current count matches the value in TIMERx_PERIOD and the counter is reloaded with the value 1.
- **Overflow**. The timer counter was incremented instead of doing a rollover when it was holding the maximum possible count value of 0xFFFF FFFF. The counter does not have a large enough range to express the next greater value and so erroneously loads a new value of 0x0000 0000.
- **Unchanged**. No new error.
    - When ERR_TYP is unchanged, it displays the previously reported error code or 00 if there has been no error since this timer was enabled.
    - When TOVF_ERR is unchanged, it reads 0 if there has been no error since this timer was enabled, or if software has performed a W1C to clear any previous error. If a previous error has not been acknowledged by software, TOVF_ERR reads 1.

Software should read TOVF_ERR to check for an error. If TOVF_ERR is set, software can then read ERR_TYP for more information. Once detected, software should write 1 to clear TOVF_ERR to acknowledge the error.

The following table can be read as: "In mode __ at event __, if TIMERx_PERIOD is __ and TIMERx_WIDTH is __, then ERR_TYP is __ and TOVF_ERR is __."

Startup error conditions do not prevent the timer from starting. Similarly, overflow and rollover error conditions do not stop the timer. Illegal cases may cause unwanted behavior of the TMRx pin.

Table 15-1. Overview of Illegal States

Mode	Event	TIMERx_ PERIOD	TIMERx_ WIDTH	ERR_TYP	TOVF_ERR
PWM_OUT, PERIOD_ CNT = 1	Startup (No boundary condition tests performed on TIMERx_ WIDTH)	== 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	< TIMERx_ PERIOD	Unchanged	Unchanged
		>= 2	>= TIMERx_ PERIOD	b#11	Set
	Overflow, not possible unless there is also another error, such as TIMERx_ PERIOD == 0.	Anything	Anything	b#01	Set

Table 15-1. 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
		This case is not detected at startup, but results in an overflow error once the counter counts through its entire range.			
		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
WDTH_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 15-4 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 timer configuration register.

In PWM_OUT mode, the bits PULSE_HI, PERIOD_CNT, IRQ_ENA, OUT_DIS, CLK_SEL, EMU_RUN, and TOGGLE_HI enable orthogonal functionality. They may be set individually or in any combination, although some combinations are not useful (such as TOGGLE_HI = 1 with OUT_DIS = 1 or PERIOD_CNT = 0).

Once a timer has been enabled, the timer counter register is loaded with a starting value. If CLK_SEL = 0, the timer counter starts at 0x1. If CLK_SEL = 1, it is reset to 0x0 as in EXT_CLK mode. The timer counts upward to the value of the timer 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

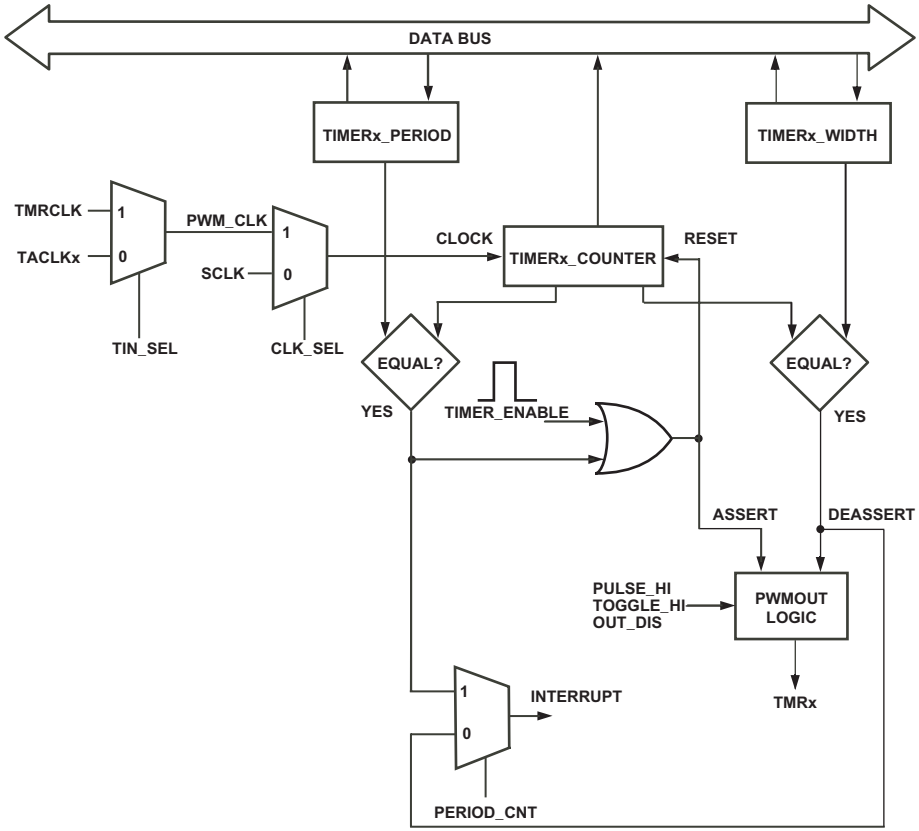


Figure 15-4. Timer Flow Diagram, PWM_OUT Mode

generates a repeating (and possibly modulated) waveform. It generates an interrupt (if enabled) at the end of each period and stops only after it is disabled. A setting of `PERIOD_CNT = 0` counts to the end of the width; a setting of `PERIOD_CNT = 1` counts to the end of the period.

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.

Output Pad Disable

The output pin can be disabled in PWM_OUT mode by setting the OUT_DIS bit in the timer configuration 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 function enable and the multiplexer control 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 pulse width register, and the period register is not used. See Figure 15-5.

At the end of the pulse, the timer interrupt latch bit TIMILx is set, and the timer is stopped automatically. No writes to the TIMERx_DISABLE register are required in this mode. If the PULSE_HI bit is set, an active high pulse is generated on the TMRx pin. If PULSE_HI is not set, the pulse is active low.

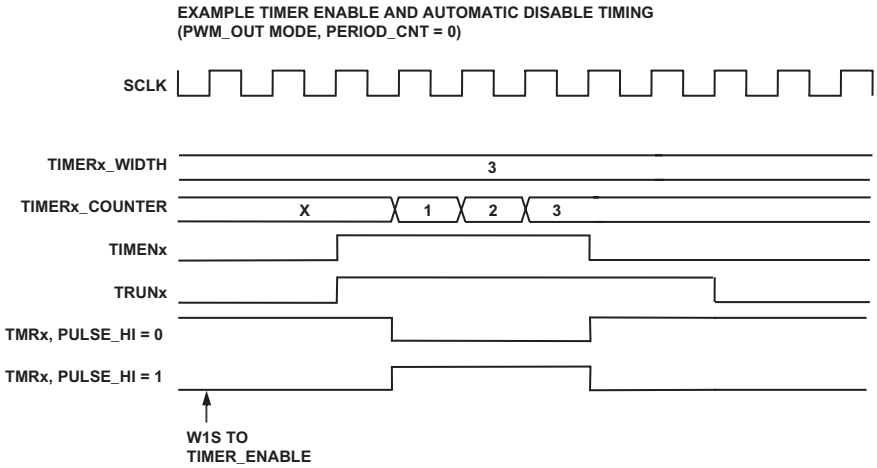


Figure 15-5. Timer Enable and Automatic Disable Timing

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 cycle (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 pulse width register, and the pin is asserted again when the period expires (or when the timer gets started).

To control the assertion sense of the 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 15-6 shows timing details.

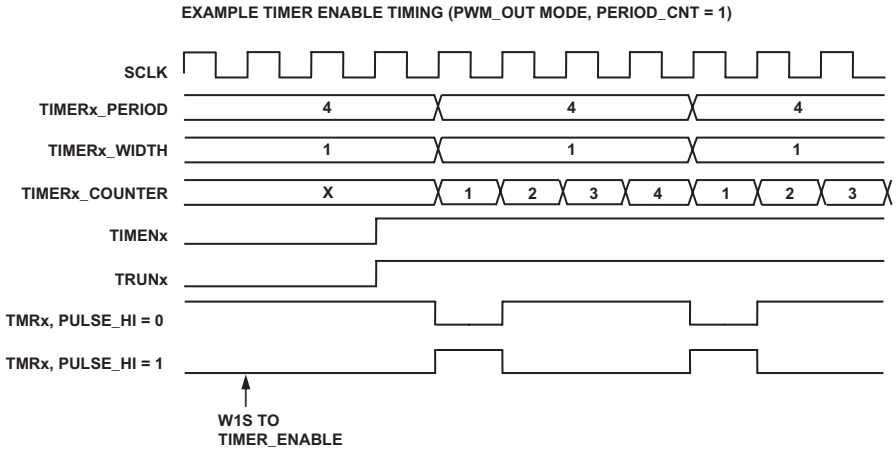


Figure 15-6. Timer Enable Timing

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 timer period and timer pulse width registers 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 the timer counter register rolls over. It is also set if the timer pulse width register is greater than or equal to the timer period register 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 (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 (via the TIMERx_WIDTH register). When two timers are running synchronously by the same period settings, the pulses are aligned to the asserting edge as shown in Figure 15-7.

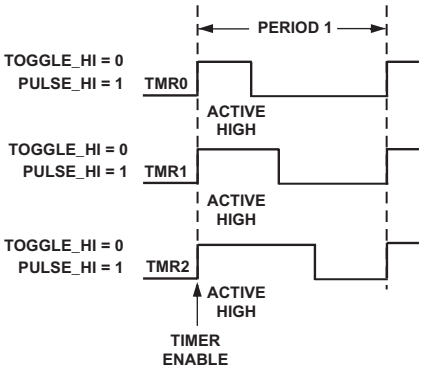


Figure 15-7. Timers With Pulses Aligned to Asserting Edge

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

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 15-8 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.

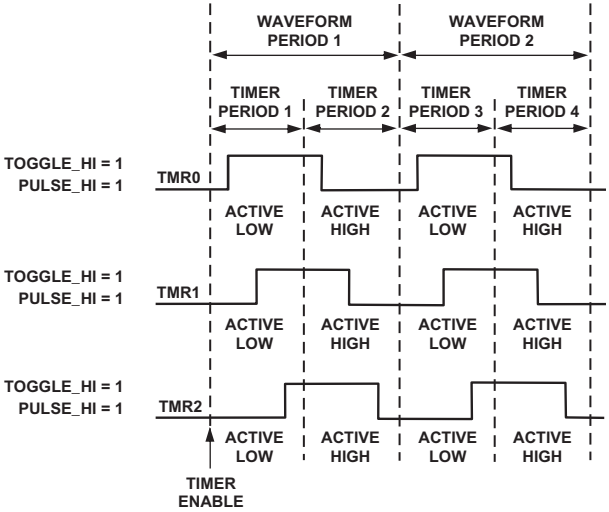


Figure 15-8. Three Timers With Same Period Settings

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 15-9).

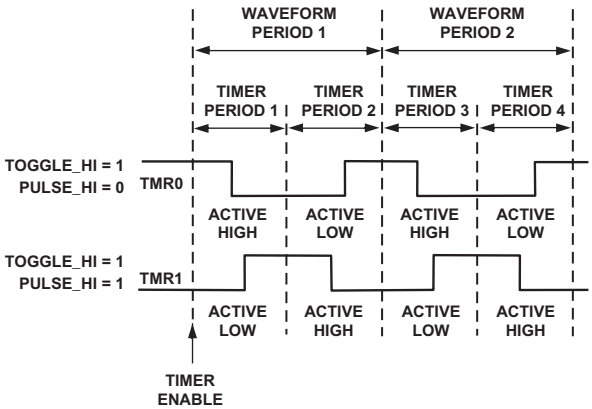


Figure 15-9. Two Timers With Non-overlapping Clocks

When TOGGLE_HI = 0, software updates the timer period and timer pulse width registers once per waveform period. When TOGGLE_HI = 1, software updates the timer period and timer pulse width registers twice per waveform. Period values are half as large. In odd-numbered periods, write (Period - Width) instead of Width to the timer pulse width register 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 (TRUNx bit in the TIMER_STATUS register) is updated only at the end of even-numbered periods in TOGGLE_HI mode. When TIMER_DISABLE 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 TIMERx_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 (TIMERx_CONFIG) register is set, the timer is clocked by PWM_CLK. The PWM_CLK is normally input from the TACLKx pin, but may be taken from the common TMRCLK pin regardless of whether the timers are configured to work with the PPI. Different timers may receive different signals on their PWM_CLK inputs, depending on configuration. As selected by the PERIOD_CNT bit, the PWM_OUT mode either generates pulse width modulation waveforms or generates a single pulse with pulse width defined by the TIMERx_WIDTH register.

When CLK_SEL is set, the counter resets to 0x0 at startup and increments on each rising edge of PWM_CLK. The TMRx pin transitions on rising edges of PWM_CLK. There is no way to select the falling edges of PWM_CLK. In this mode, the PULSE_HI bit controls only the polarity of the pulses produced. The timer interrupt may occur slightly before the corresponding edge on the TMRx pin (the interrupt occurs on an SCLK edge, the pin transitions on a later PWM_CLK edge). It is still safe to program new period and pulse width values as soon as the interrupt occurs. After a period expires, the counter rolls over to a value of 0x1.

The PWM_CLK clock waveform is not required to have a 50% duty cycle, but the minimum PWM_CLK clock low time is one SCLK period, and the minimum PWM_CLK clock high time is one SCLK period. This implies the maximum PWM_CLK clock frequency is SCLK/2.

The 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 multiplexer control and function enable registers.

Using PWM_OUT Mode With the PPI

Up to three timers are used to generate frame sync signals for certain PPI modes. For detailed instructions on how to configure the timers for use with the PPI, refer to "Frame Synchronization in GP Modes" on page 7-20 of the PPI chapter.

Stopping the Timer in PWM_OUT Mode

In all PWM_OUT mode variants, the timer treats a disable operation (W1C to TIMER_DISABLE) as a "stop is pending" condition. When disabled, it automatically completes the current waveform and then stops cleanly. This prevents truncation of the current pulse and unwanted PWM patterns at the TMRx pin. The processor can determine when the timer stops running by polling for the corresponding TRUNx bit in the TIMER_STATUS 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_DISABLE to stop the timer. At the end of the pulse, the timer stops automatically, the corresponding bit in TIMER_ENABLE (and TIMER_DISABLE) is cleared, and the corresponding TRUNx bit is cleared. See Figure 15-5 on page 15-15. To generate multiple pulses, write a 1 to TIMER_ENABLE, wait for the timer to stop, then write another 1 to TIMER_ENABLE.

In continuous PWM generation mode (PWM_OUT, PERIOD_CNT = 1) software can stop the timer by writing to the TIMERx_DISABLE register. To prevent the ongoing PWM pattern from being spoiled in unpredictable fashion, the timer does not stop immediately when the corresponding 1 has been written to the TIMERx_DISABLE 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 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 15-10 shows detailed timing.

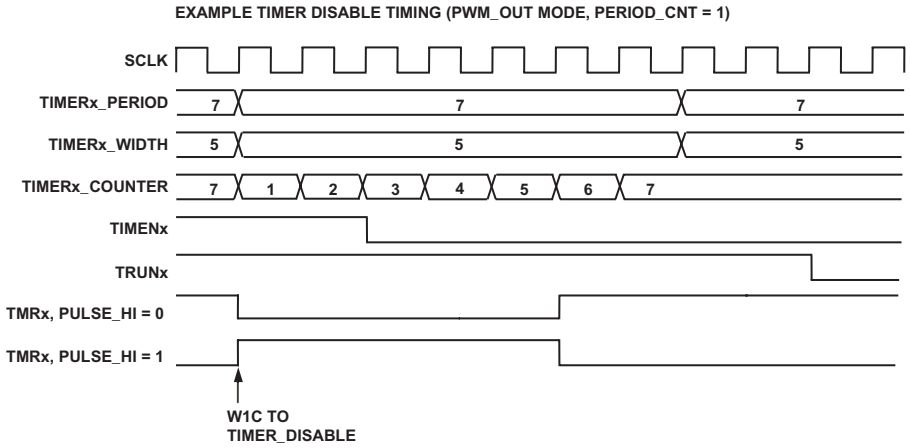


Figure 15-10. Timer Disable Timing

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_DISABLE, and then writing a 1 to the corresponding TRUNx bit in TIMER_STATUS. This stops the timer whether the pending stop was waiting for the end of the current period (PERIOD_CNT = 1) or the end of the current pulse width (PERIOD_CNT = 0). This feature may be used to regain immediate control of a timer during an error recovery sequence.

Use this feature carefully, because it may corrupt the PWM pattern generated at the TMRx pin.

When timers are disabled, the timer counter registers retain their state; when a timer is re-enabled, the timer counter is reinitialized based on the operating mode. The timer counter registers are read-only. Software cannot overwrite or preset the timer counter value directly.

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 15-11 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 counter register 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.

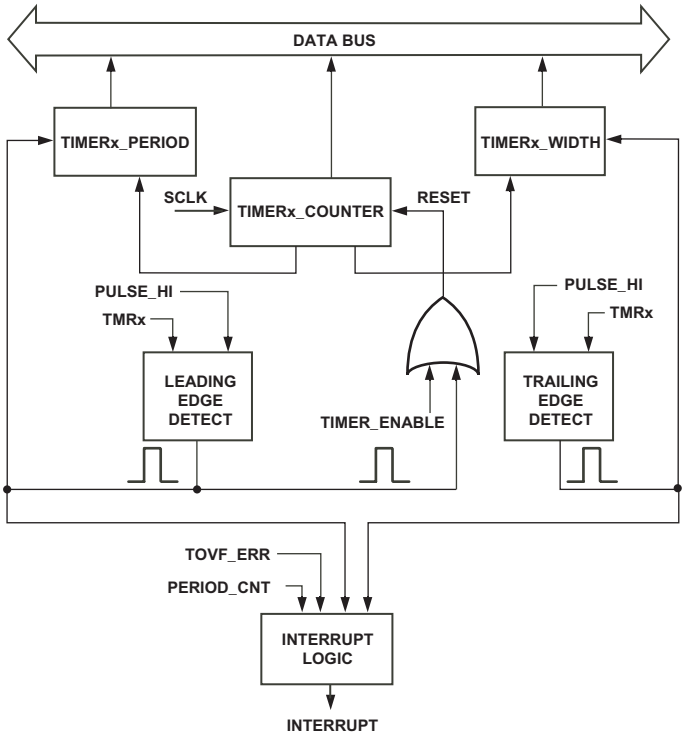


Figure 15-11. Timer Flow Diagram, WDTH_CAP Mode

In `WDTH_CAP` mode, these three events always occur at the same time as one unit:

1. The `TIMERx_PERIOD` register is updated from the period buffer register.
2. The `TIMERx_WIDTH` register is updated from the width buffer register.
3. The `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 at which 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 at most once per input signal period.

The current timer counter value is always copied to the width buffer and period buffer registers at the trailing and leading edges of the input signal, respectively, but these values are not visible to software. A measurement report event samples the captured values into visible registers and sets the timer interrupt to signal that TIMERx_PERIOD and TIMERx_WIDTH are ready to be read. When the PERIOD_CNT bit is set, the measurement report occurs just after the period buffer register captures its value (at a leading edge). When the PERIOD_CNT bit is cleared, the measurement report occurs just after the width buffer register captures its value (at a trailing edge).

If the PERIOD_CNT bit is set and a leading edge occurred (see Figure 15-12), 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 15-13), 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 15-13. 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

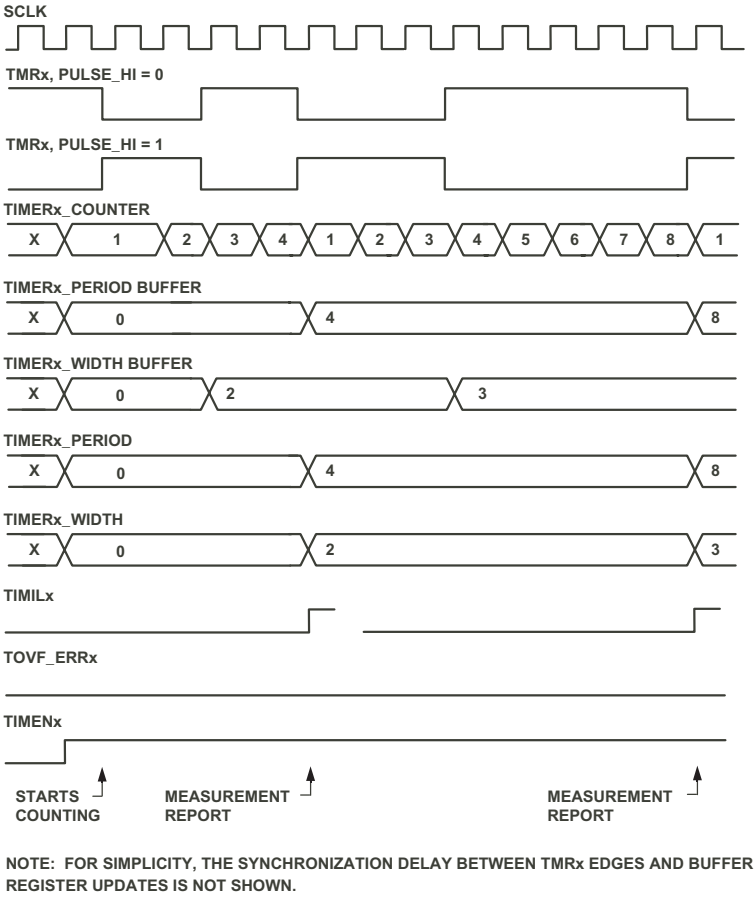


Figure 15-12. Example of Period Capture Measurement Report Timing (WDTH_CAP mode, PERIOD_CNT = 1)

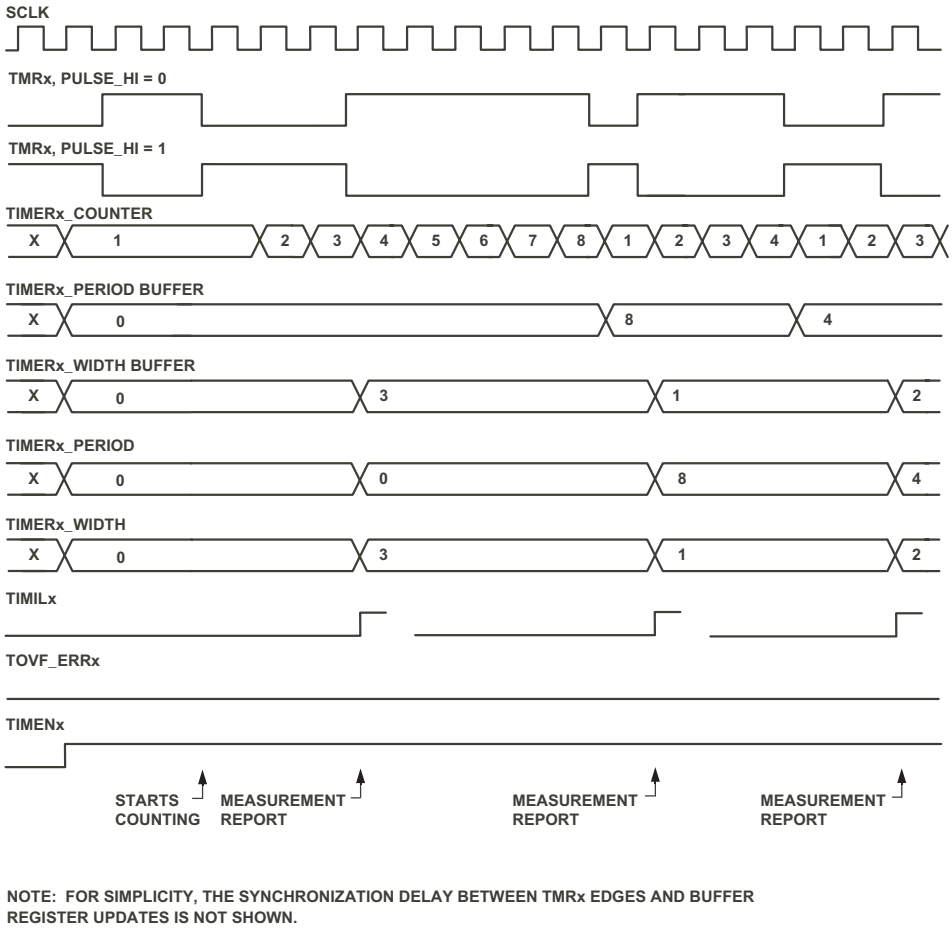


Figure 15-13. Example of Width Capture Measurement Report Timing (WDTH_CAP mode, PERIOD_CNT = 0)

(because no measurement report occurred to copy the value captured in the width buffer register to TIMERx_WIDTH). See the first interrupt in Figure 15-14.

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 the timer counter register wraps around from 0xFFFF FFFF to 0 in the absence of a leading edge. At that point, the TOVF_ERRx bit in the TIMER_STATUS register and the ERR_TYP bits in the TIMERx_CONFIG register are set, indicating a count overflow due to a period greater than the counter's range. This is called an error report. When a timer generates an interrupt in WDTH_CAP mode, either an error has occurred (an error report) or a new measurement is ready to be read (a measurement report), but never both at the same time. The TIMERx_PERIOD and TIMERx_WIDTH registers are never updated at the time an error is signaled. Refer to Figure 15-14 and Figure 15-15 for more information.

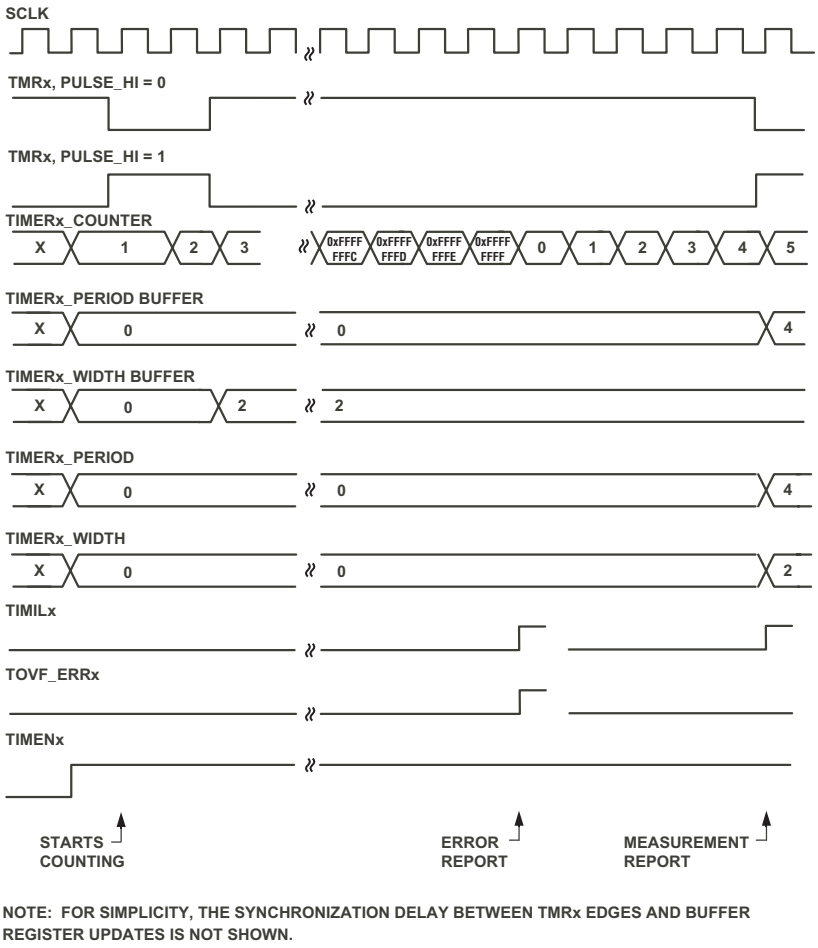
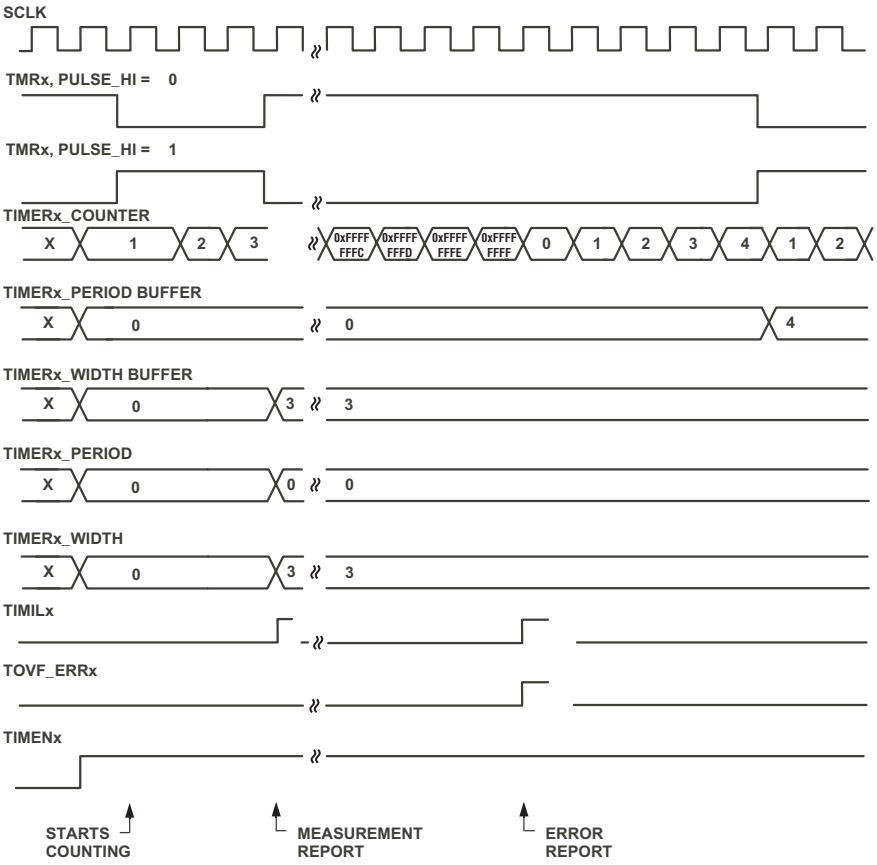


Figure 15-14. Example Timing for Period Overflow Followed by Period Capture (WDTH_CAP mode, PERIOD_CNT = 1)



NOTE: FOR SIMPLICITY, THE SYNCHRONIZATION DELAY BETWEEN TMRx EDGES AND BUFFER REGISTER UPDATES IS NOT SHOWN.

Figure 15-15. 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 15-14 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 operation. For details with the UART interface, see Chapter 13, “UART Port Controllers”. For details with the CAN interface, see Chapter 9, “CAN Module”.

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 15-16 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.

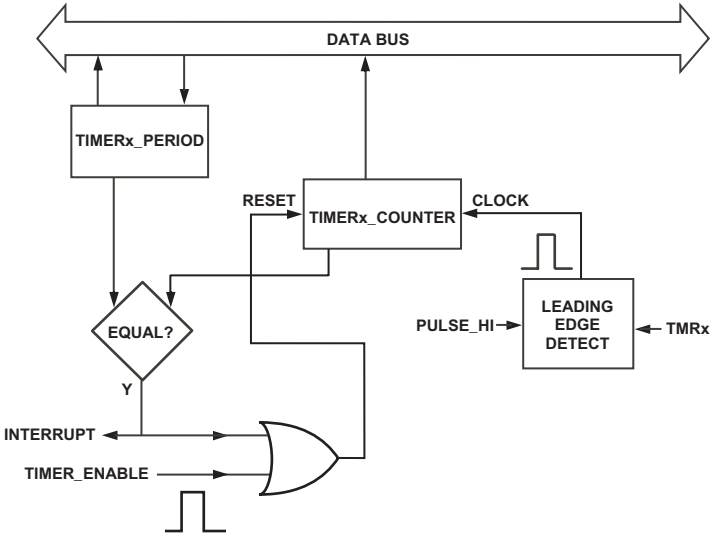


Figure 15-16. Timer Flow Diagram, EXT_CLK Mode

After the timer has been enabled, it resets the timer counter register to 0x0 and then waits for the first leading edge on the TMRx pin. This edge causes the timer counter register to 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 the timer counter register again with 0x1. The timer continues counting until it is disabled. The PULSE_HI bit determines whether the leading edge is rising (PULSE_HI set) or falling (PULSE_HI cleared).

The configuration bits, TIN_SEL and PERIOD_CNT, have no effect in this mode. The TOVF_ERRx and ERR_TYP bits are set if the timer counter register wraps around from 0xFFFF FFFF to 0 or if Period = 0 at startup or when the timer counter register rolls over (from Count = Period to Count = 0x1). The timer pulse width register is unused.

Programming Model

The architecture of the timer block enables any of the eight timers to work individually or synchronously along with others as a group of timers. 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. 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 WDTH_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 eight identical timer units.

Each timer provides four registers:

- TIMERx_CONFIG[15:0] – timer configuration register
- TIMERx_WIDTH[31:0] – timer pulse width register
- TIMERx_PERIOD[31:0] – timer period register
- TIMERx_COUNTER[31:0] – timer counter register

Additionally, three registers are shared between the eight timers:

- TIMER_ENABLE[15:0] – timer enable register
- TIMER_DISABLE[15:0] – timer disable register
- TIMER_STATUS[31:0] – timer status register

The size of accesses is enforced. A 32-bit access to a timer configuration register or a 16-bit access to a timer pulse width, timer period, or timer counter register results in a Memory-Mapped Register (MMR) error. Both 16- and 32-bit accesses are allowed for the timer enable, timer disable, and timer status registers. On a 32-bit read of one of the 16-bit registers, the upper word returns all 0s.

TIMER_ENABLE Register

The TIMER_ENABLE register, shown in Figure 15-17, allows all eight timers to be enabled simultaneously in order to make them run completely synchronously. For each timer there is a single W1S control bit. Writing a 1 enables the corresponding timer; writing a 0 has no effect. The eight bits can be set individually or in any combination. A read of the TIMER_ENABLE register shows the status of the enable for the corresponding timer. A 1 indicates that the timer is enabled. All unused bits return 0 when read.

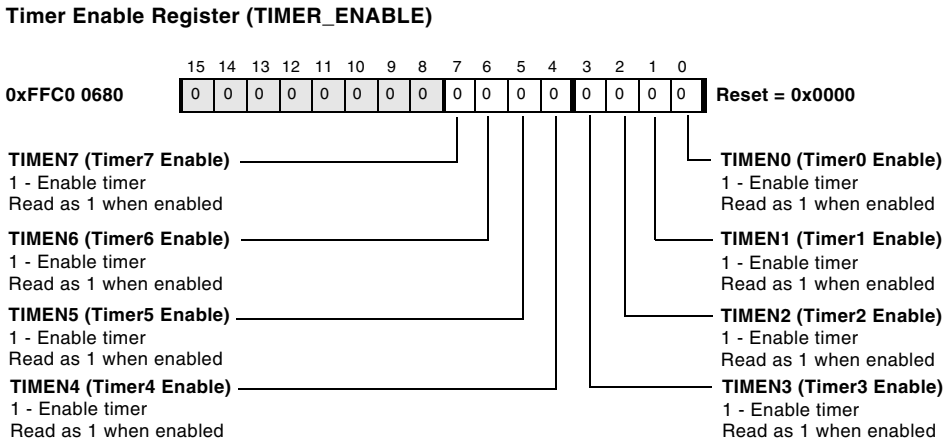


Figure 15-17. Timer Enable Register

TIMER_DISABLE Register

The `TIMER_DISABLE` register, shown in Figure 15-18, allows all eight timers 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 eight bits can be cleared individually or in any combination. A read of the `TIMER_DISABLE` register returns a value identical to a read of the `TIMER_ENABLE` register. A 1 indicates that the timer is enabled. All unused bits return 0 when read.

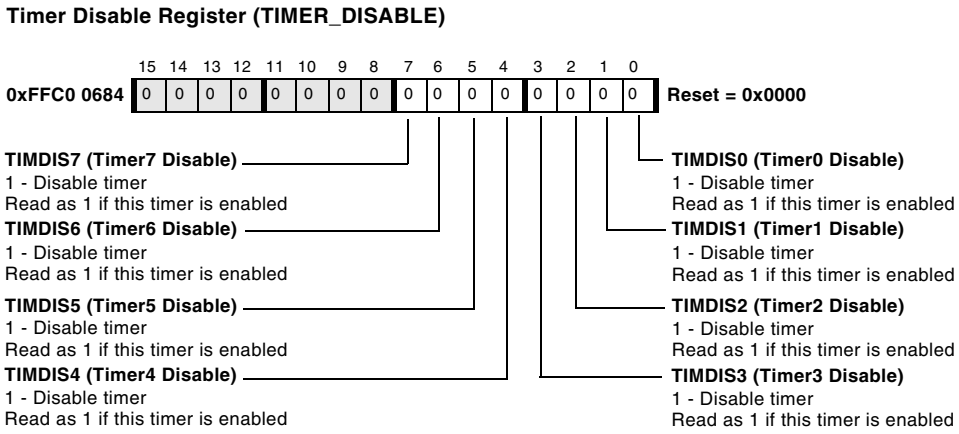


Figure 15-18. Timer Disable Register

In PWM_OUT mode, a write of a 1 to TIMER_DISABLE 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_DISABLE, and then writing a 1 to the corresponding TRUNx bit in TIMER_STATUS. See "Stopping the Timer in PWM_OUT Mode" on page 15-23.

In WDTH_CAP and EXT_CLK modes, a write of a 1 to TIMER_DISABLE stops the corresponding timer immediately.

TIMER_STATUS Register

The TIMER_STATUS register indicates the status of the timers and is used to check the status of eight timers with a single read. The TIMER_STATUS register, shown in Figure 15-19, reports the status of timer 0 through timer 7. 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_STATUS register read access, all reserved or unused bits return a 0.

For detailed behavior and usage of the TRUNx bit see "Stopping the Timer in PWM_OUT Mode" on page 15-23. 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 15-10.

Timer Status Register (TIMER_STATUS)

All bits are W1C

0xFFC0 0688

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Reset = 0x0000 0000

TRUN7 (Timer7 Slave Enable Status)
Read as 1 if timer running, W1C to abort in PWM_OUT mode

TRUN6 (Timer6 Slave Enable Status)
Read as 1 if timer running, W1C to abort in PWM_OUT mode

TRUN5 (Timer5 Slave Enable Status)
Read as 1 if timer running, W1C to abort in PWM_OUT mode

TRUN4 (Timer4 Slave Enable Status)
Read as 1 if timer running, W1C to abort in PWM_OUT mode

TOVF_ERR7 (Timer7 Counter Overflow)
Indicates that an error or an overflow occurred

TOVF_ERR6 (Timer6 Counter Overflow)
Indicates that an error or an overflow occurred

TIMIL4 (Timer4 Interrupt)
Indicates an interrupt request when IRQ_ENA is set

TIMIL5 (Timer5 Interrupt)
Indicates an interrupt request when IRQ_ENA is set

TIMIL6 (Timer6 Interrupt)
Indicates an interrupt request when IRQ_ENA is set

TIMIL7 (Timer7 Interrupt)
Indicates an interrupt request when IRQ_ENA is set

TOVF_ERR4 (Timer4 Counter Overflow)
Indicates that an error or an overflow occurred

TOVF_ERR5 (Timer5 Counter Overflow)
Indicates that an error or an overflow occurred

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

TRUN3 (Timer3 Slave Enable Status)
Read as 1 if timer running, W1C to abort in PWM_OUT mode

TRUN2 (Timer2 Slave Enable Status)
Read as 1 if timer running, W1C to abort in PWM_OUT mode

TRUN1 (Timer1 Slave Enable Status)
Read as 1 if timer running, W1C to abort in PWM_OUT mode

TRUN0 (Timer0 Slave Enable Status)
Read as 1 if timer running, W1C to abort in PWM_OUT mode

TOVF_ERR3 (Timer3 Counter Overflow)
Indicates that an error or an overflow occurred

TOVF_ERR2 (Timer2 Counter Overflow)
Indicates that an error or an overflow occurred

TIMIL0 (Timer0 Interrupt)
Indicates an interrupt request when IRQ_ENA is set

TIMIL1 (Timer1 Interrupt)
Indicates an interrupt request when IRQ_ENA is set

TIMIL2 (Timer2 Interrupt)
Indicates an interrupt request when IRQ_ENA is set

TIMIL3 (Timer3 Interrupt)
Indicates an interrupt request when IRQ_ENA is set

TOVF_ERR0 (Timer0 Counter Overflow)
Indicates that an error or an overflow occurred

TOVF_ERR1 (Timer1 Counter Overflow)
Indicates that an error or an overflow occurred

Figure 15-19. Timer Status Register

TIMERx_CONFIG Registers

The operating mode for each timer is specified by its TIMERx_CONFIG register. The TIMERx_CONFIG register, shown in Figure 15-20, may be written only when the timer is not running. After disabling the timer in PWM_OUT mode, make sure the timer has stopped running by checking its TRUNx bit in TIMER_STATUS before attempting to reprogram TIMERx_CONFIG. The TIMERx_CONFIG registers may be read at any time. The ERR_TYP field is read-only. It is cleared at reset and when the timer is enabled. Each time TOVF_ERRx is set, ERR_TYP[1:0] is loaded with a code that identifies the type of error that was detected. This value is held until the next error or timer enable occurs. For an overview of error conditions, see Table 15-1 on page 15-11. 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.

When operating the PPI in GP output modes with internal frame syncs, the CLK_SEL and the TIN_SEL bits must be set to 1.

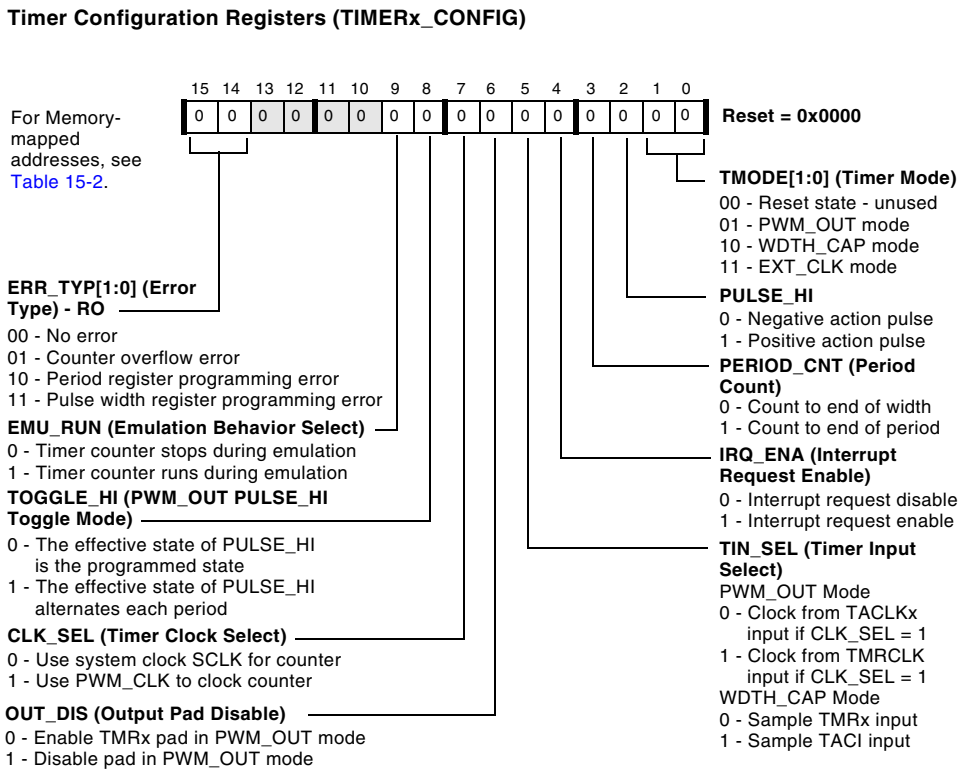


Figure 15-20. Timer Configuration Registers

Table 15-2. Timer Configuration Register Memory-mapped Addresses

Register Name	Memory-mapped Address
TIMER0_CONFIG	0xFFC0 0600
TIMER1_CONFIG	0xFFC0 0610
TIMER2_CONFIG	0xFFC0 0620
TIMER3_CONFIG	0xFFC0 0630

Table 15-2. Timer Configuration Register Memory-mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
TIMER4_CONFIG	0xFFC0 0640
TIMER5_CONFIG	0xFFC0 0650
TIMER6_CONFIG	0xFFC0 0660
TIMER7_CONFIG	0xFFC0 0670

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 15-21, 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 TMRCLK pin, which is most likely used as the PPI clock (PPI_CLK).

While the processor core is being accessed by an external emulator debugger, all code execution stops. By default, the TIMERx_COUNTER also halts its counting during an emulation access in order to remain synchronized with the software. While stopped, the count does not advance—in PWM_OUT mode, the TMRx pin waveform is "stretched"; in WDTH_CAP mode, measured values are incorrect; in EXT_CLK mode, input events on TMRx may be missed. All other timer functions such as register reads and writes, interrupts previously asserted (unless cleared), and the loading of TIMERx_PERIOD and TIMERx_WIDTH in WDTH_CAP mode remain active during an emulation stop.

Some applications may require the timer to continue counting asynchronously to the emulation-halted processor core. Set the EMU_RUN bit in TIMERx_CONFIG to enable this behavior.

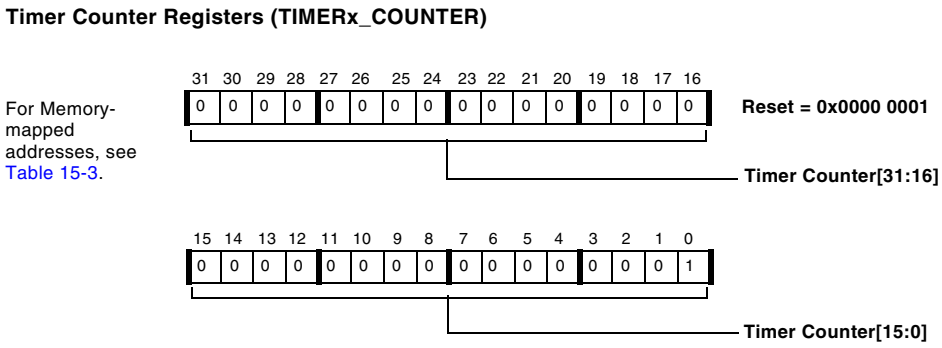


Figure 15-21. Timer Counter Registers

Table 15-3. Timer Counter Register Memory-mapped Addresses

Register Name	Memory-mapped Address
TIMER0_COUNTER	0xFFC0 0604
TIMER1_COUNTER	0xFFC0 0614
TIMER2_COUNTER	0xFFC0 0624
TIMER3_COUNTER	0xFFC0 0634
TIMER4_COUNTER	0xFFC0 0644
TIMER5_COUNTER	0xFFC0 0654
TIMER6_COUNTER	0xFFC0 0664
TIMER7_COUNTER	0xFFC0 0674

TIMERx_PERIOD and TIMERx_WIDTH Registers

When a timer is enabled and running, and the software writes new values to the timer period register and the timer pulse width register, the writes are buffered and do not update the registers until the end of the current period (when the timer counter register equals the timer period register).

Usage of the TIMERx_PERIOD register, shown in Figure 15-22, and the TIMERx_WIDTH register, shown in Figure 15-23, varies depending on the mode of the timer:

- In pulse width modulation mode (PWM_OUT), both the timer period and timer pulse width register values can be updated "on-the-fly" since the timer period and timer pulse width (duty cycle) register values change simultaneously.
- In pulse width and period capture mode (WDTH_CAP), the timer period and timer pulse width buffer values are captured at the appropriate time. The timer period and timer pulse 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 timer period register is writable and can be updated "on-the-fly." The timer pulse width register is not used.

If new values are not written to the timer period register or the timer pulse width register, the value from the previous period is reused. Writes to the 32-bit timer period register and timer pulse width register are atomic; it is not possible for the high word to be written without the low word also being written.

Values written to the timer period registers or timer pulse width registers are always stored in the buffer registers. Reads from the timer period or timer pulse width registers always return the current, active value of period or pulse width. Written values are not read back until they become active. When the timer is enabled, they do not become active until after the timer period and timer pulse width registers are updated from their respective buffers at the end of the current period. See Figure 15-2 on page 15-4.

When the timer is disabled, writes to the buffer registers are immediately copied through to the timer period or timer pulse width register so that they will be ready for use in the first timer period. For example, to change

the values for the timer period and/or timer pulse width registers in order to use a different setting for each of the first three timer periods after the timer is enabled, the procedure to follow is:

1. Program the first set of register values.
2. Enable the timer.
3. Immediately program the second set of register values.
4. Wait for the first timer interrupt.
5. Program the third set of register values.

Each new setting is then programmed when a timer interrupt is received.

In `PWM_OUT` mode with very small periods (less than 10 counts), there may not be enough time between updates from the buffer registers to write both the timer period register and the timer pulse width register. The next period may use one old value and one new value. In order to prevent "pulse width >= period" errors, write the timer pulse width register before the timer period register when decreasing the values, and write the timer period register before the timer pulse width register when increasing the value.

Timer Period Registers (TIMERx_PERIOD)

For Memory-mapped addresses, see Table 15-4.

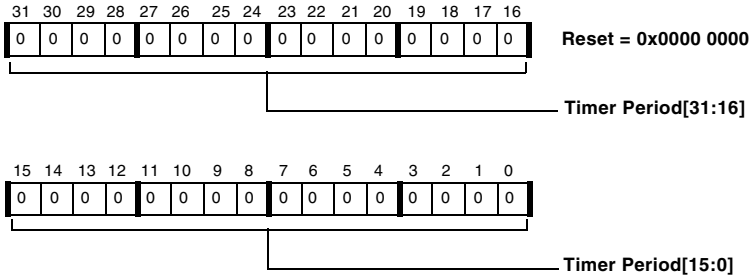


Figure 15-22. Timer Period Registers

Table 15-4. Timer Period Register Memory-mapped Addresses

Register Name	Memory-mapped Address
TIMER0_PERIOD	0xFFC0 0608
TIMER1_PERIOD	0xFFC0 0618
TIMER2_PERIOD	0xFFC0 0628
TIMER3_PERIOD	0xFFC0 0638
TIMER4_PERIOD	0xFFC0 0648
TIMER5_PERIOD	0xFFC0 0658
TIMER6_PERIOD	0xFFC0 0668
TIMER7_PERIOD	0xFFC0 0678

Timer Width Registers (TIMERx_WIDTH)

For Memory-mapped addresses, see Table 15-5.

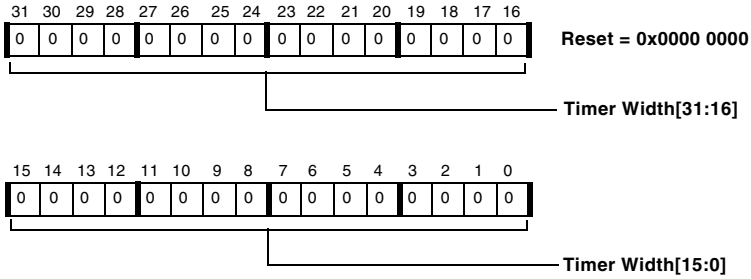


Figure 15-23. Timer Width Registers

Table 15-5. Timer Width Register Memory-mapped Addresses

Register Name	Memory-mapped Address
TIMER0_WIDTH	0xFFC0 060C
TIMER1_WIDTH	0xFFC0 061C
TIMER2_WIDTH	0xFFC0 062C
TIMER3_WIDTH	0xFFC0 063C

Table 15-5. Timer Width Register Memory-mapped Addresses (Cont'd)

Register Name	Memory-mapped Address
TIMER4_WIDTH	0xFFC0 064C
TIMER5_WIDTH	0xFFC0 065C
TIMER6_WIDTH	0xFFC0 066C
TIMER7_WIDTH	0xFFC0 067C

Summary

Table 15-6 summarizes control bit and register usage in each timer mode.

Table 15-6. Control Bit and Register Usage Chart

Bit / Register	PWM_OUT Mode	WDTH_CAP Mode	EXT_CLK Mode
TIMER_ENABLE	1 - Enable timer 0 - No effect	1 - Enable timer 0 - No effect	1 - Enable timer 0 - No effect
TIMER_DISABLE	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 15-6. 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 appropri-ate
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

Table 15-6. 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 15-1 configures the port control registers in a way that all eight `TMRx` pins are connected to port F.

Listing 15-1. Port Setup

```
timer_port_setup:
    [--sp] = (r7:7, p5:5);
```

```
    p5.h = hi(PORTF_FER);
    p5.l = lo(PORTF_FER);
    r7.l = PF2|PF3|PF4|PF5|PF6|PF7|PF8|PF9;
    w[p5] = r7;
    p5.l = lo(PORT_MUX);
    r7.l = PFTE;
    w[p5] = r7;
    (r7:7, p5:5) = [sp++];
    rts;
timer_port_setup.end:
```

Listing 15-2 generates signals on the TMR4 (PF5) and TMR5 (PF4) 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 15-2. Signal Generation

```
// #define SINGLE_PULSE
timer45_signal_generation:
    [--sp] = (r7:7, p5:5);
    p5.h = hi(TIMER_ENABLE);
    p5.l = lo(TIMER_ENABLE);
#ifdef SINGLE_PULSE
    r7.l = PULSE_HI | PWM_OUT;
#else
    r7.l = PERIOD_CNT | PULSE_HI | PWM_OUT;
```

```
#endif
    w[p5 + TIMER5_CONFIG - TIMER_ENABLE] = r7;
    w[p5 + TIMER4_CONFIG - TIMER_ENABLE] = r7;
    r7 = 0x10 (z);
    [p5 + TIMER5_WIDTH - TIMER_ENABLE] = r7;
    r7 = 0x20 (z);
    [p5 + TIMER4_WIDTH - TIMER_ENABLE] = r7;
#ifndef SINGLE_PULSE
    r7 = 0x40 (z);
    [p5 + TIMER5_PERIOD - TIMER_ENABLE] = r7;
    [p5 + TIMER4_PERIOD - TIMER_ENABLE] = r7;
#endif
    r7.l = TIMEN5 | TIMEN4;
    w[p5] = r7;
    (r7:7, p5:5) = [sp++];
    rts;
timer45_signal_generation.end:
```

All subsequent examples use interrupts. Thus, Listing 15-3 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 15-3. 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_IMASK);
    p5.l = lo(SIC_IMASK);
/* assign timer 5 IRQ = IRQ24 to IVG7 */
    r7 = -1 (x);
    [p5 + SIC_IAR0 - SIC_IMASK] = r7;
    [p5 + SIC_IAR1 - SIC_IMASK] = r7;
    [p5 + SIC_IAR2 - SIC_IMASK] = r7;
    r7.h = hi(P24_IVG(7));
    r7.l = lo(P24_IVG(7));
    [p5 + SIC_IAR3 - SIC_IMASK] = r7;
/* enable timer 5 IRQ */
    r7 = [p5];
    bitset(r7, 24);
    [p5] = r7;
/* enable interrupt nesting */
    (r7:7, p5:5) = [sp++];
    [--sp] = reti;
    rts;
timer5_interrupt_setup.end:
```

The example shown in Listing 15-4 does not drive the TMRx pin. It generates periodic interrupt requests every 0x1000 SCLK cycles. If the preprocessor constant SINGLE_PULSE was 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 just the clearing of the interrupt request and counting interrupt occurrences.

Listing 15-4. Periodic Interrupt Requests

```
// #define SINGLE_PULSE
timer5_interrupt_generation:
    [--sp] = (r7:7, p5:5);
    p5.h = hi(TIMER_ENABLE);
```

```
    p5.l = lo(TIMER_ENABLE);
#ifdef SINGLE_PULSE
    r7.l = EMU_RUN | IRQ_ENA | OUT_DIS | PWM_OUT;
#else
    r7.l = EMU_RUN | IRQ_ENA | PERIOD_CNT | OUT_DIS | PWM_OUT;
#endif
    w[p5 + TIMER5_CONFIG - TIMER_ENABLE] = r7;
    r7 = 0x1000 (z);
#ifndef SINGLE_PULSE
    [p5 + TIMER5_PERIOD - TIMER_ENABLE] = r7;
    r7 = 0x1 (z);
#endif
    [p5 + TIMER5_WIDTH - TIMER_ENABLE] = 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_STATUS);
    p5.l = lo(TIMER_STATUS);
    r7.h = hi(TIMIL5);
    r7.l = lo(TIMIL5);
    [p5] = r7;
    r0+= 1;
    ssync;
    (r7:7, p5:5) = [sp++];
    astat = [sp++];
    rti;
isr_timer5.end:
```

Listing 15-5 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.

Figure 15-24 explains how the signal waveform represented by the period P and the pulse width W translates to timer period and width values. Table 15-7 summarizes the register writes.

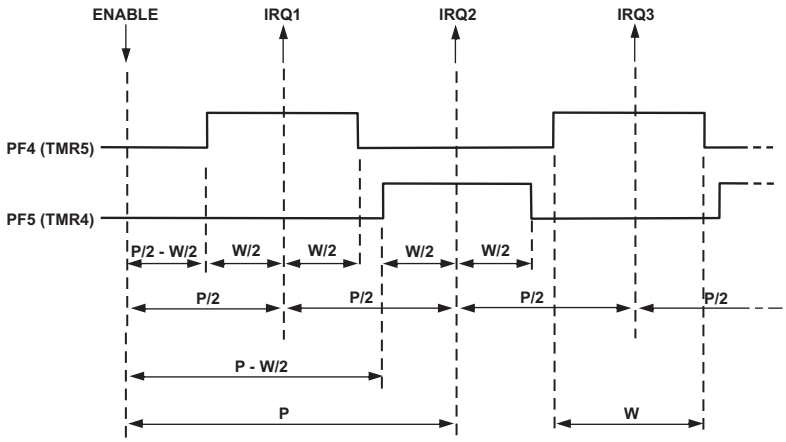


Figure 15-24. Non-Overlapping Clock Pulses

Table 15-7. 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

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 15-5. 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_ENABLE);
    p5.l = lo(TIMER_ENABLE);
/* config timers */
    r7.l = IRQ_ENA | PERIOD_CNT | TOGGLE_HI | PULSE_HI | PWM_OUT;
    w[p5 + TIMER5_CONFIG - TIMER_ENABLE] = r7;
    r7.l = PERIOD_CNT | TOGGLE_HI | PULSE_HI | PWM_OUT;
    w[p5 + TIMER4_CONFIG - TIMER_ENABLE] = 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_ENABLE] = r0;
    [p5 + TIMER4_WIDTH - TIMER_ENABLE] = r5;
    [p5 + TIMER5_PERIOD - TIMER_ENABLE] = r3;
    [p5 + TIMER5_WIDTH - TIMER_ENABLE] = r4;
```

```
/* start timers */
    r7.l = TIMEN5 | TIMEN4 ;
    w[p5 + TIMER_ENABLE - TIMER_ENABLE] = r7;
/* write values for second period */
    [p5 + TIMER4_PERIOD - TIMER_ENABLE] = r3;
    [p5 + TIMER5_WIDTH - TIMER_ENABLE] = 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_ENABLE);
    p5.l = lo(TIMER_ENABLE);
/* clear interrupt request */
    r7.h = hi(TIMIL5);
    r7.l = lo(TIMIL5);
    [p5 + TIMER_STATUS - TIMER_ENABLE] = r7;
/* toggle width values (width = period - width) */
    r7 = [p5 + TIMER5_PERIOD - TIMER_ENABLE];
    r6 = [p5 + TIMER5_WIDTH - TIMER_ENABLE];
    r5 = r7 - r6;
    [p5 + TIMER5_WIDTH - TIMER_ENABLE] = r5;
    r5 = [p5 + TIMER4_WIDTH - TIMER_ENABLE];
    r7 = r7 - r5;
    CC = r7 < 0;
    if CC r7 = r6;
    [p5 + TIMER4_WIDTH - TIMER_ENABLE] = 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_DISABLE - TIMER_ENABLE] = r7;
    (r7:5, p5:5) = [sp++];
    astat = [sp++];
    rti;
isr_timer5.end:
```

Listing 15-5 generates N pulses on both timer output pins. Disabling the timers does not corrupt the generated pulse pattern anyhow.

Listing 15-6 configures timer 5 in WDTH_CAP mode. If looped back externally, this code might be used to receive N PWM patterns generated by one of the other timers. Ensure that the PWM generator and consumer both use the same PERIOD_CNT and PULSE_HI settings.

Listing 15-6. 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_ENABLE);
    p5.l = lo(TIMER_ENABLE);
```

```
    r7.l = EMU_RUN|IRQ_ENA|PERIOD_CNT|PULSE_HI|WDTH_CAP;
    w[p5 + TIMER5_CONFIG - TIMER_ENABLE] = r7;
    r7.l =  TIMEN5;
    w[p5 + TIMER_ENABLE - TIMER_ ENABLE] = 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_STATUS);
    p5.l = lo(TIMER_STATUS);
    r7.h = hi(TIMIL5);
    r7.l = lo(TIMIL5);
    [p5] = r7;
    r7 = [p5 + TIMER0_PERIOD - TIMER_STATUS];
    [i2++] = r7;
    r7 = [p5 + TIMER0_WIDTH - TIMER_STATUS];
    [i2++] = r7;
    ssync;
    (r7:7, p5:5) = [sp++];
    astat = [sp++];
    rti;
isr_timer5.end:
```

16 CORE TIMER

This brief chapter describes the core timer. Following an overview, functional description, and consolidated register definitions, the chapter concludes with a programming example.

Overview and Features

The core timer is a programmable 32-bit interval timer which can generate periodic interrupts. Unlike other peripherals, the core timer resides inside the Blackfin core and runs at the core clock (CCLK) rate. Core timer features include:

- 32-bit timer with 8-bit prescaler
- Operates at core clock (CCLK) rate
- Dedicated high-priority interrupt channel
- Single-shot or continuous operation

Timer Overview

Figure 16-1 provides a block diagram of the core timer.

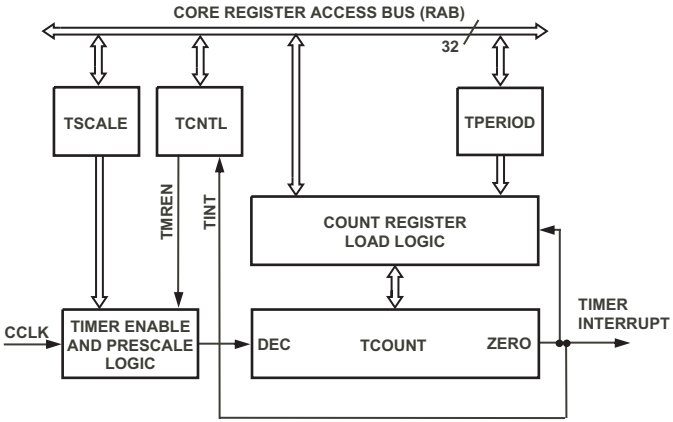


Figure 16-1. Core Timer Block Diagram

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.

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 `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 is set, the core timer may then be enabled by setting the TMREN bit in the TCNTL register.

Hardware behavior is undefined if TMREN 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 peripherals, the core interrupt request is edge sensitive and cleared by hardware automatically as soon as the interrupt serviced.

The TINT bit in the TCNTL register indicates that an interrupt has been generated. Note that this is *not* a W1C bit. Write a 0 to clear it. However, the write is optional. It is not required to clear interrupt requests. The core time module doesn't 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 four core Memory-Mapped Registers (MMRs), the timer control register (TCNTL), the timer count register (TCOUNT), the timer period register (TPERIOD), and the timer scale register (TSCALE). As with all core MMRs, these registers are always accessed by 32-bit read and write operations.

TCNTL Register

The timer control register, shown in Figure 16-2, functions as control and status register.

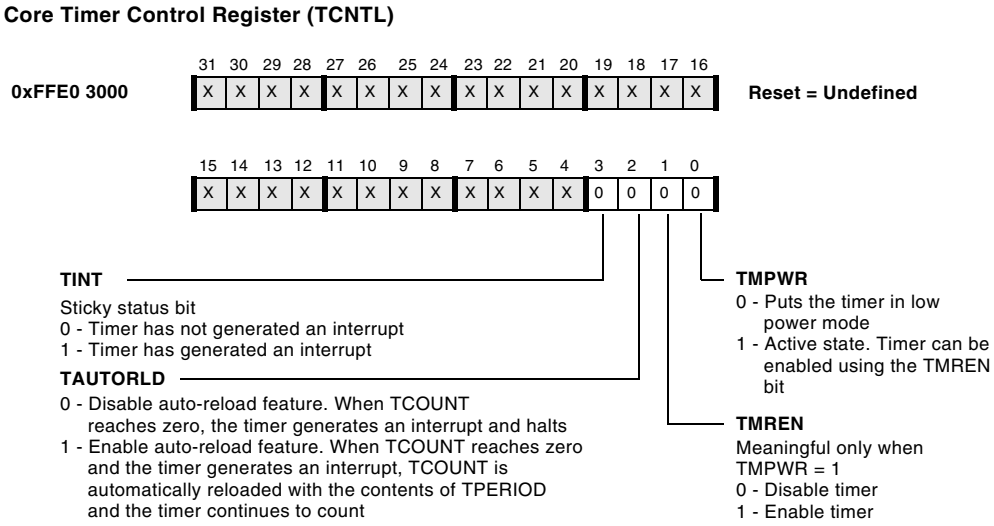


Figure 16-2. Core Timer Control Register

TCOUNT Register

The core timer count register (TCOUNT, shown in Figure 16-3) 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 afterward.

Writes to TCOUNT are ignored once the timer is running.

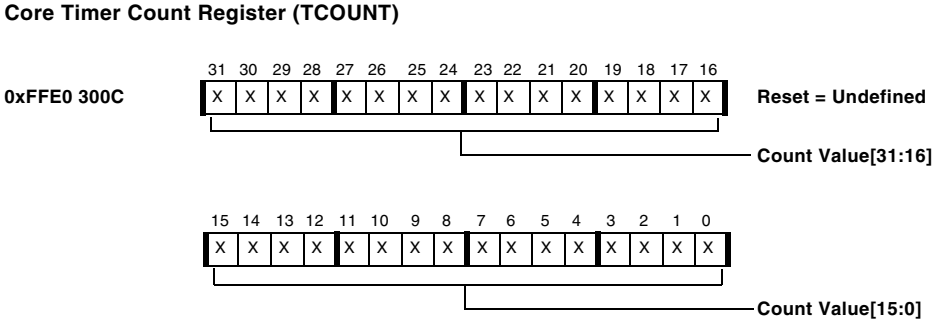


Figure 16-3. Core Timer Count Register

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 16-4), whenever TCOUNT reaches 0. Writes to TPERIOD are ignored when the timer is running.

Core Timer Period Register (TPERIOD)

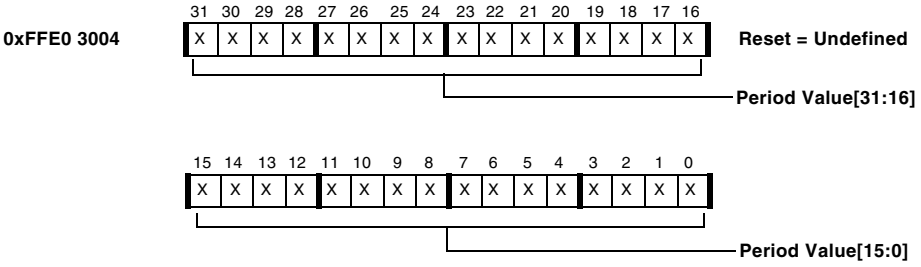


Figure 16-4. Core Timer Period Register

TSCALE Register

The core timer scale register (TSCALE, shown in Figure 16-5,) stores the scaling value that is one less than the number of cycles between decrements of TCOUNT. For example, if the value in the TSCALE register is 0, the counter register decrements once every CCLK clock cycle. If TSCALE is 1, the counter decrements once every two cycles.

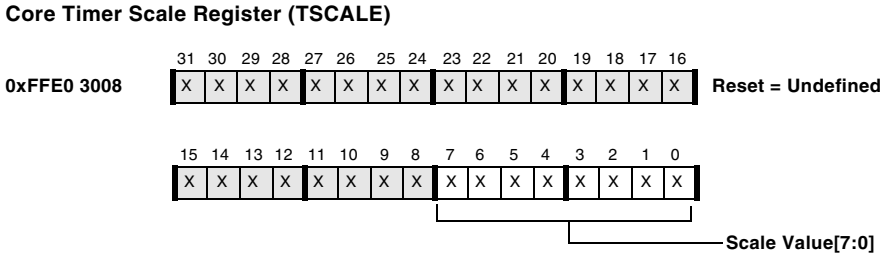


Figure 16-5. Core Timer Scale Register

Programming Examples

Listing 16-1 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 16-1. Core Timer Configuration

```
#include <defBF534.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:
```

17 WATCHDOG TIMER

This brief chapter describes the watchdog timer. Following an overview, functional description, and consolidated register definitions, the chapter concludes with programming examples.

Overview and Features

The processor includes a 32-bit timer that can be used to implement a software watchdog function. A software watchdog can improve system reliability by generating an event to the processor core if the 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

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 in 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 be rebooted. 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 17-1 provides a block diagram of the watchdog timer.

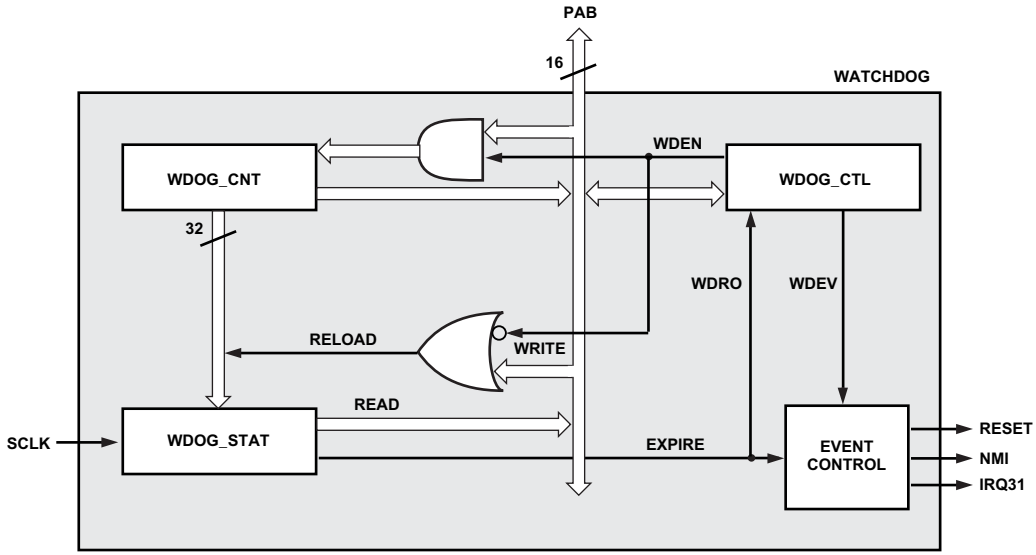


Figure 17-1. Watchdog Timer Block Diagram

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.

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 enabled yet, the write to the WDOG_CNT registers automatically pre-loads the WDOG_STAT register as well.
2. In the watchdog control register (WDOG_CTL), select the event to be generated 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 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 19, “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 be reloaded 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 WDG_CTL register.

Register Definitions

The watchdog timer is controlled by three registers.

WDOG_CNT Register

The watchdog count register (WDOG_CNT, shown in Figure 17-2) 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.

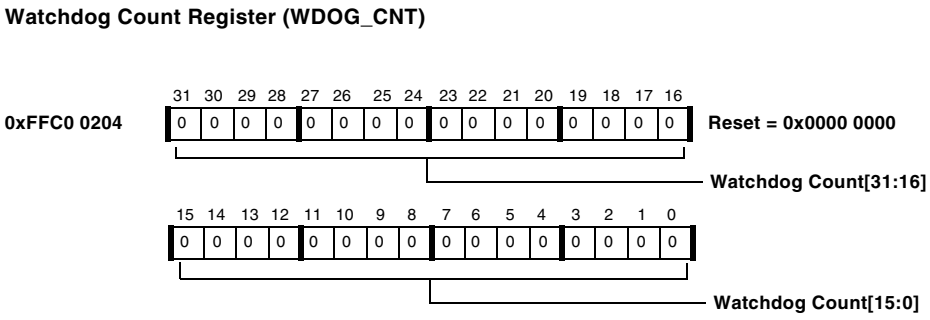


Figure 17-2. Watchdog Count Register

WDOG_STAT Register

The 32-bit watchdog status register (WDOG_STAT, shown in Figure 17-3) contains the current count value of the watchdog timer. Reads to WDOG_STAT return the current count value. Values cannot be stored directly in WDOG_STAT, but are instead copied from WDOG_CNT. This can happen in two ways.

- While the watchdog timer is disabled, writing the WDOG_CNT register pre-loads the WDOG_STAT register.
- While the watchdog timer is enabled, but not rolled over yet, writes to the WDOG_STAT register load it with the value in WDOG_CNT.

Enabling the watchdog timer does not automatically reload WDOG_STAT from WDOG_CNT.

The WDOG_STAT register is a 32-bit unsigned system memory-mapped register that must be accessed with 32-bit reads and writes.

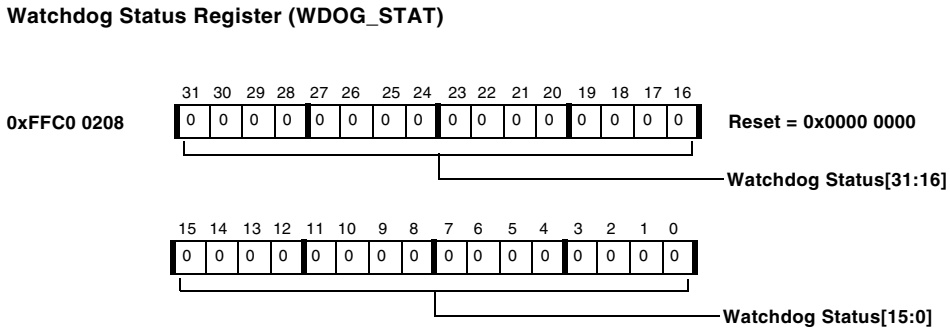


Figure 17-3. Watchdog Status Register

WDOG_CTL Register

The watchdog control register (WDOG_CTL, shown in Figure 17-4) 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)

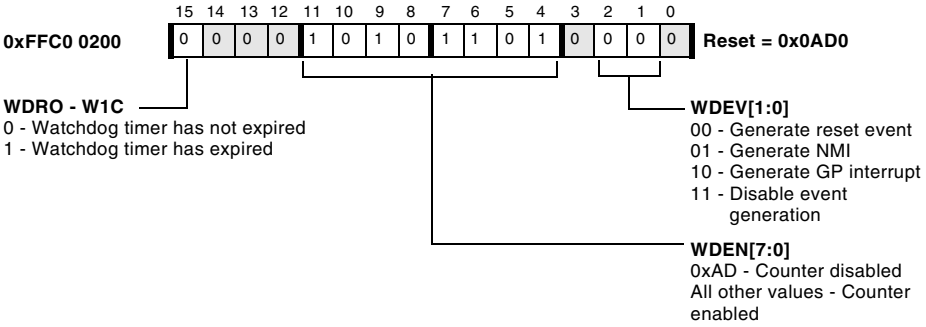


Figure 17-4. Watchdog Control Register

Programming Examples

Listing 17-1 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 has been caused by the watchdog. Additionally, the example sets the NOBOOT bit to prevent the memory from being rebooted.

Listing 17-1. Watchdog Timer Configuration

```
#include <defBF534.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 17-2 can be called by software to service the watchdog. Note that the value written to the `WDOG_STAT` register does not matter.

Listing 17-2. Service Watchdog

```
service_watchdog:
    [--sp] = p5;
    p5.h = hi(WDOG_STAT);
    p5.l = lo(WDOG_STAT);
    [p5] = r0;
    p5 = [sp++];
    rts;
service_watchdog.end:
```

Listing 17-3 is an interrupt service routine that restarts the watchdog. Note that the watchdog must be disabled first.

Listing 17-3. 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(WDRO));
    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:
```

18 REAL-TIME CLOCK

This chapter describes the Real-Time Clock (RTC). Following an overview and list of key features is a description of operation. The chapter concludes with a programming model, consolidated register definitions, and programming examples.

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.

The RTC input clock is divided down to a 1 Hz signal by a prescaler, which can be bypassed. When bypassed, the RTC is clocked at the 32.768 kHz crystal rate. In normal operation, the prescaler is enabled.

The primary function of the RTC is to maintain an accurate day count and time of day. The RTC accomplishes this by means of four counters:

- 60-second counter
- 60-minute counter

- 24-hour counter
- 32768-day counter

The RTC increments the 60-second counter once per second and increments the other three counters when appropriate. The 32768-day counter is incremented each day at midnight (0 hours, 0 minutes, 0 seconds). Interrupts can be issued periodically, either every second, every minute, every hour, or every day. Each of these interrupts can be independently controlled.

The RTC provides two alarm features, programmed with the RTC Alarm register (`RTC_ALARM`). The first is a time of day alarm (hour, minute, and second). When the alarm interrupt is enabled, the RTC generates an interrupt each day at the time specified. The second alarm feature allows the application to specify a day as well as a time. When the day alarm interrupt is enabled, the RTC generates an interrupt on the day and time specified. The alarm interrupt and day alarm interrupt can be enabled or disabled independently.

The RTC provides a stopwatch function that acts as a countdown timer. The application can program a second count into the RTC stopwatch count register (`RTC_SWCNT`). When the stopwatch interrupt is enabled and the specified number of seconds have elapsed, the RTC generates an interrupt.

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).

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 18-1 provides a block diagram of the RTC.

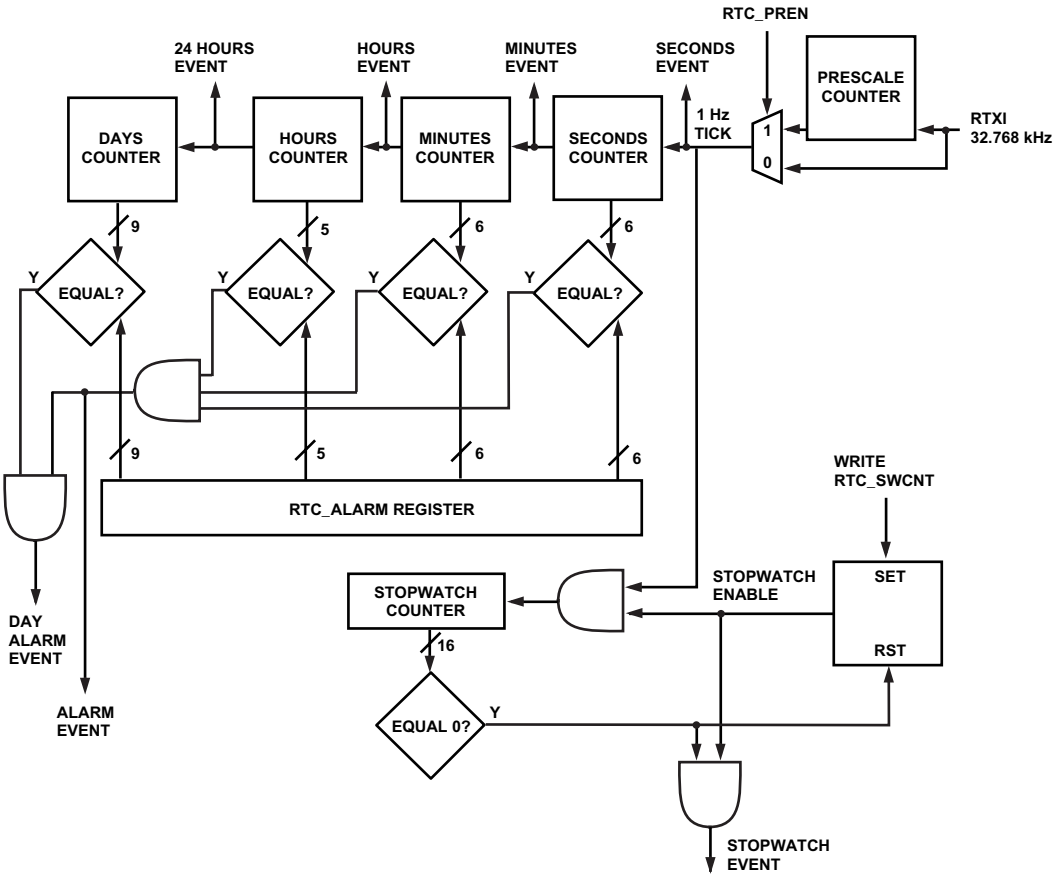


Figure 18-1. RTC Block Diagram

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 RTXO 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 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 via 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 µs.

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 via the system MMR status registers at any time.

The primary RTC functionality, shown in Figure 18-1 on page 18-3, 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 18-2 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).

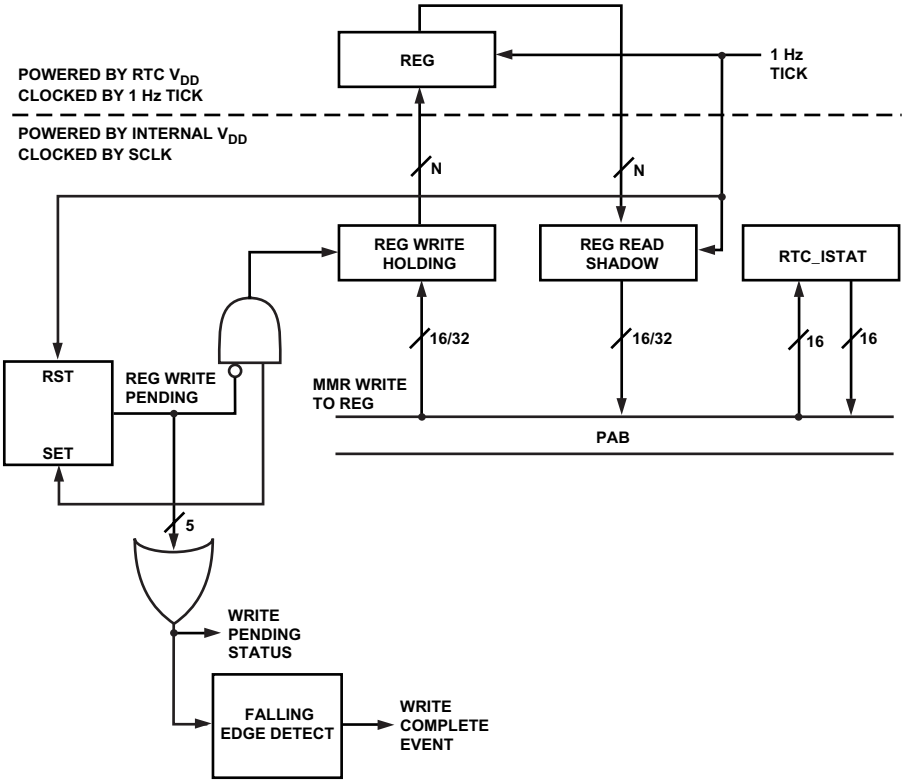


Figure 18-2. RTC Register Architecture

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_ICTL to cause an

interrupt. Software does not have to wait for writes to one 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 has been written before the write complete flag is set will return the old value. Always check the write pending status bit before attempting a read or write.

Write Latency

Writes to the RTC MMRs are synchronized to the 1 Hz RTC clock. When setting the time of day, do not factor in the delay when writing to the RTC MMRs. The most accurate method of setting the 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 ms after a 1 Hz tick completes on the next 1 Hz tick, but the simplest, most predictable and recommended

technique is to only post writes to RTC_STAT, RTC_ALARM, RTC_SWCNT, RTC_ICTL, or RTC_PREN immediately after a seconds interrupt or event. All five registers may be written in the same second.

W1C bits in the RTC_ISTAT register take effect immediately.

Register Reads

There is no latency when reading RTC MMRs, as the values come from the read shadow registers. The shadows are updated and ready for reading by the time any RTC interrupts or event flags for that second are asserted. Once the internal 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. 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 µs) after SCLK restarts. This loads all shadow registers with up-to-date values and sets the seconds event flag. Other event flags may also be set. When the system wakes up from deep sleep, whether by an RTC event or a hardware reset, all of the RTC events that occurred during that second (and only that second) are reported in RTC_ISTAT.

When the system wakes up from deep sleep state, software does not need to W1C the bits in RTC_ISTAT. All 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

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 1s to clear the RTC_ISTAT flags for alarm, day alarm, stop-watch, 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.

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 register (RTC_SWCNT) 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

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 18-3, the RTC generates an interrupt request (IRQ) to the processor core for event handling and wakeup from a sleep state. The RTC generates a separate signal for wakeup from a deep sleep or from an internal V_{dd} power-off state. The deep sleep wakeup signal is asserted at the 1 Hz tick when any RTC interval event enabled in RTC_ICTL occurs. The assertion of the deep sleep wakeup signal causes the processor core clock (CCLK) and the system clock (SCLK) to restart. Any enabled event that asserts the RTC deep sleep wakeup signal also causes the RTC IRQ to assert once SCLK restarts.

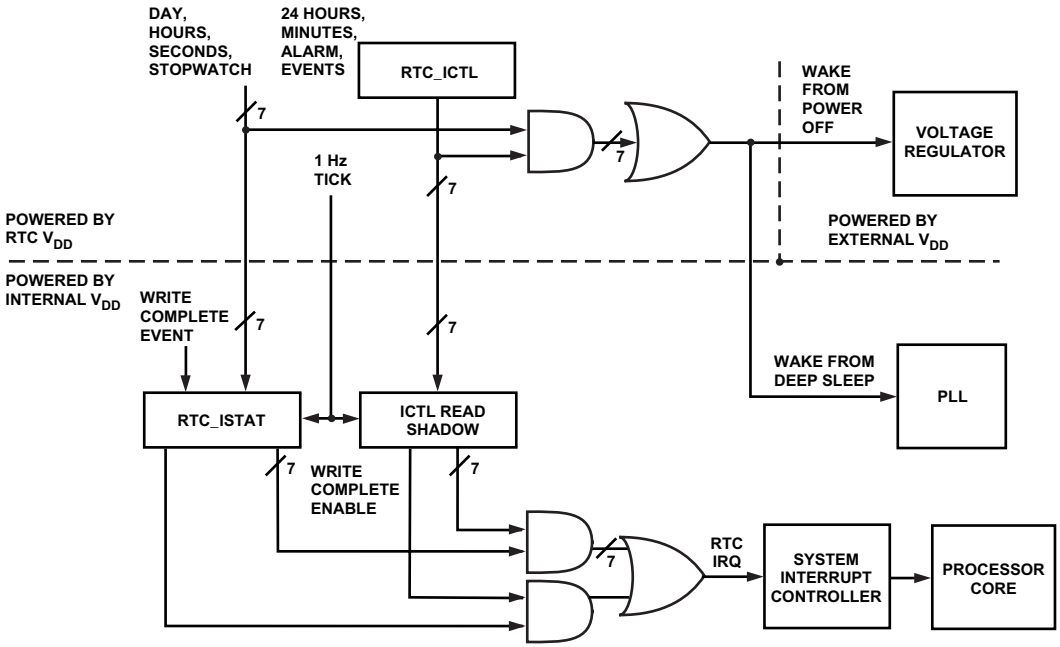


Figure 18-3. RTC Interrupt Structure

State Transitions Summary

Table 18-1 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 20, "Dynamic Power Management". "No power" means none of the processor power supply pins are connected to a source of energy. "Off" means the processor core, peripherals, and memory are not powered (Internal V_{dd} is off), while the RTC is still powered and running. External V_{dd} may still be powered. Registers described as "as written" are holding the last value software wrote to the register. If the register has not been written since RTC V_{dd} power was applied, then the state is unknown (for all bits of `RTC_STAT`, `RTC_ALARM`, and `RTC_SWCNT`, and for some bits of `RTC_ISTAT`, `RTC_PREN`, and `RTC_ICTL`).

Table 18-1. 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 18-2 summarizes software's responsibilities with respect to the RTC at various system state transition events.

Table 18-2. 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 18-2. 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 will be 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).

Register Definitions

The following sections contain the register definitions. Figure 18-4 through Figure 18-9 on page 18-22 illustrate the registers.

Table 18-3 shows the functions of the RTC registers.

Table 18-3. RTC Register Mapping

Register Name	Function	Notes
RTC_STAT	RTC status register	Holds time of day
RTC_ICTL	RTC interrupt control register	Bits 14:7 are reserved
RTC_ISTAT	RTC interrupt status register	Bits 13:7 are reserved
RTC_SWCNT	RTC stopwatch count register	Undefined at reset
RTC_ALARM	RTC alarm register	Undefined at reset
RTC_PREN	Prescaler enable register	Always set PREN = 1 for 1 Hz ticks

RTC_STAT Register

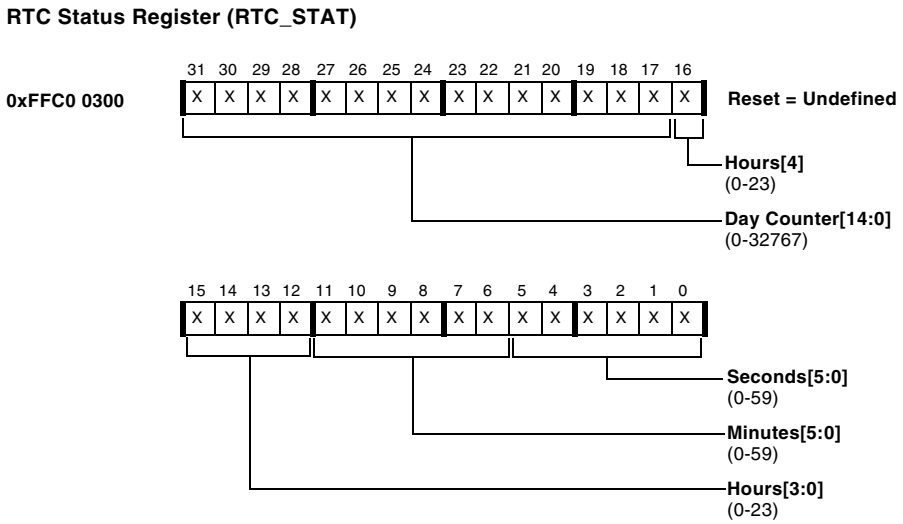


Figure 18-4. RTC Status Register

RTC_ICTL Register

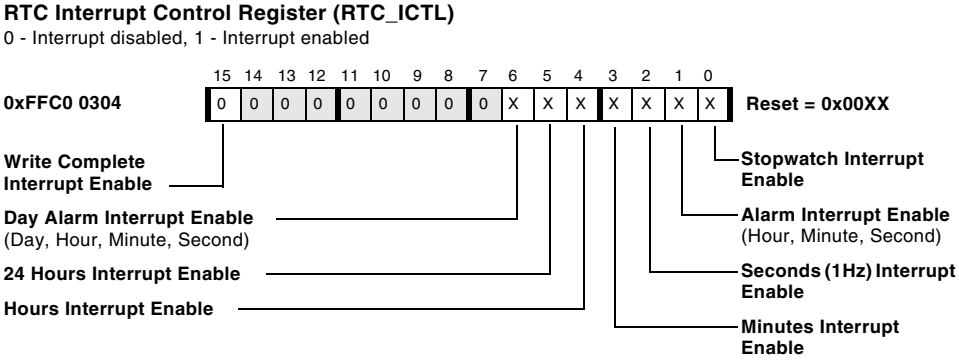


Figure 18-5. RTC Interrupt Control Register

RTC_ISTAT Register

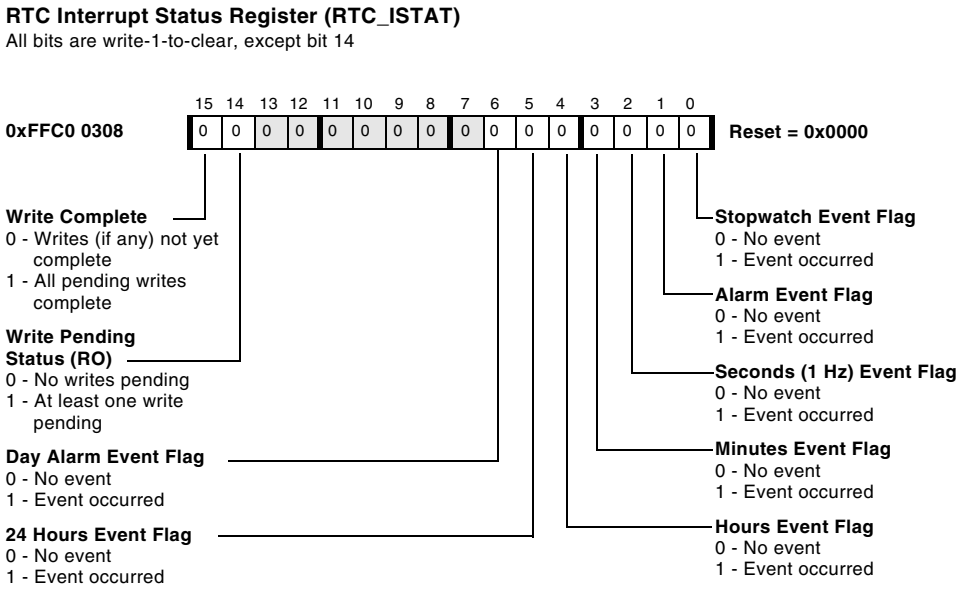


Figure 18-6. RTC Interrupt Status Register

RTC_SWCNT Register

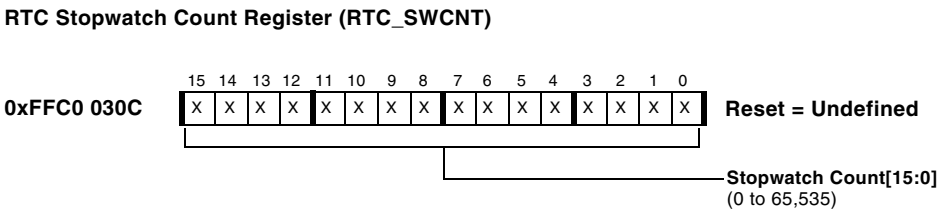


Figure 18-7. RTC Stopwatch Count Register

RTC_ALARM Register

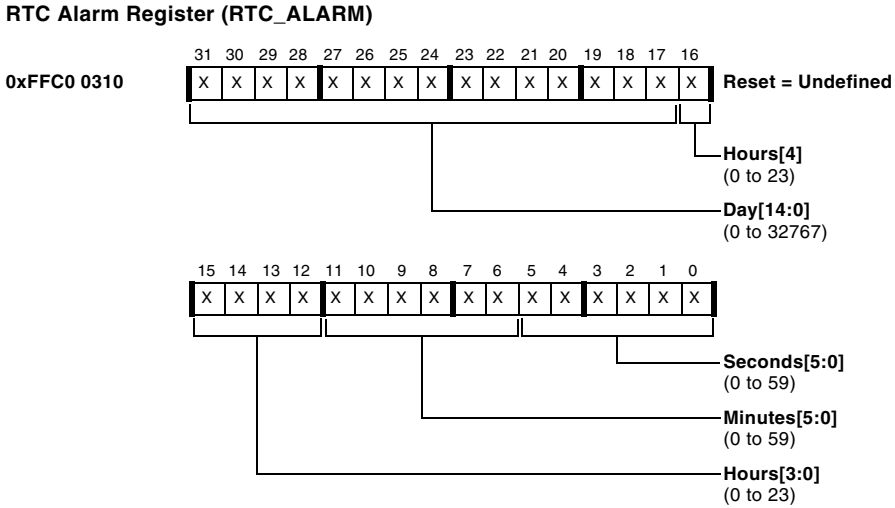


Figure 18-8. RTC Alarm Register

RTC_PREN Register

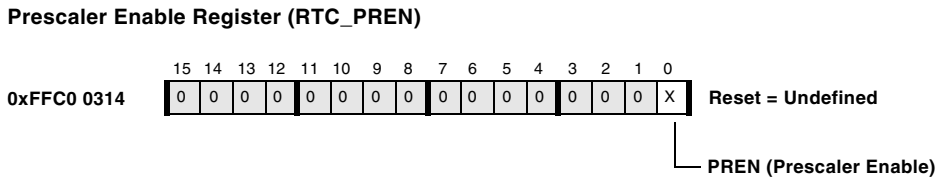


Figure 18-9. Prescaler Enable Register

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 <defBF537.h>` for ADSP-BF537 projects).

Enable RTC Prescaler

Listing 18-1 properly enables the prescaler and clears any pending interrupts.

Listing 18-1. 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 18-2 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 18-2. RTC Stopwatch Interrupt to Exit Deep Sleep

```
/* RTC Wake-Up Interrupt To Be Used With Deep Sleep Code */
_RTC_Interrupt:
  P0.H = HI(PLL_CTL);
  P0.L = LO(PLL_CTL);
  R0 = W[P0](Z);
  BITCLR (R0, BITPOS(BYPASS));
  W[P0] = R0;  /* BYPASS Set By Default, Must Clear It */

  IDLE;  /* Must go to IDLE for PLL changes to be effected */

  R0 = 0x807F(Z);
  P0.H = HI(RTC_ISTAT);
  P0.L = LO(RTC_ISTAT);
  W[P0] = R7;   /* clear pending RTC IRQs */

  R0 = WRITE_COMPLETE(Z);  /* mask for WRITE-COMPLETE bit */
  Poll_WC_IRQ:   R1 = W[P0](Z);
                 R1 = R1 & R0;  /* wait for Write Complete */
                 CC = AZ;
                 IF CC JUMP Poll_WC_IRQ;
  RTI;

Deep_Sleep_Code:
  P0.H = HI(RTC_SWCNT);
```

```
  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 18-3 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 18-3. Setting RTC Alarm to Exit Hibernate State

```
Hibernate_Code:
  P0.H = HI(RTC_ALARM);
  P0.L = LO(RTC_ALARM);
  R0 = 0x0010(Z);  /* set alarm to 16 seconds from now */
  W[P0] = R0.L;

  P0.L = LO(RTC_STAT);
  R0 = 0;  /* Clear RTC Status to Start Counting at 0 */
  W[P0] = R0.L;

  P0.L = LO(RTC_ICTL);
  R0 = ALARM(Z);
  W[P0] = R0.L;  /* enable Alarm interrupt */

  P0.L = LO(RTC_ISTAT);
  R0 = 0x807F(Z);
  W[P0] = R0.L;  /* clear any pending RTC interrupts */

  R0 = WRITE_COMPLETE(Z);
  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 RTC status */
GoToHibernate:
```

```
P0.H = HI(VR_CTL);
P0.L = LO(VR_CTL);
R0 = W[P0](Z);
BITCLR(R0, 0);  /* Clear FREQ (bits 0 and 1) to */
BITCLR(R0, 1);  /* go to Hibernate State        */
BITSET(R0, BITPOS(WAKE)); /* Enable RTC Wakeup  */
W[P0] = R0.L;

CLI R0;  /* Use PLL programming sequence to */
IDLE;    /* make VR_CTL changes take effect */
RTS;     /* Should Never Execute This */
```

19 SYSTEM RESET AND BOOTING

When the $\overline{\text{RESET}}$ input signal releases, the processor starts fetching and executing instructions from either off-chip asynchronous memory or from the on-chip boot ROM.

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 as well as special commands that instruct the boot kernel on how to initialize on-chip L1 SRAM 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 to jump to the processor's reset vector at 0xFFA0 0000 in on-chip L1 memory. This process is called "booting."

The processor features three dedicated input pins BMODE[2:0] that instruct the processor on how to behave after reset. If all three pins are low when $\overline{\text{RESET}}$ releases, the processor bypasses the boot ROM and starts code execution directly at address 0x2000 0000 in off-chip memory bank 0. Then a 16-bit memory device must be connected to the $\overline{\text{AMS0}}$ strobe. Otherwise program execution starts at 0xEF00 0000, which is populated by the boot ROM. The boot kernel further evaluates the BMODE pins and performs booting from respective sources. Table 19-1 shows the truth table of the BMODE pins.

Table 19-1. Reset Vector Addresses

Boot Source	BMODE[2:0]	Execution Start Address
Bypass boot ROM; execute from 16-bit external memory connected to ASYNC Bank 0	000	0x2000 0000
Use boot ROM to boot from 8-bit or 16-bit memory (PROM/flash)	001	0xEF00 0000
Reserved	010	0xEF00 0000
Boots from 8-, 16-, or 24-bit addressable SPI memory in SPI master mode with support for Atmel AT45DB041B, AT45DB081B, and AT45DB161B DataFlash® devices	011	0xEF00 0000
Boot from SPI host (slave mode)	100	0xEF00 0000
Boot from serial TWI memory (EEPROM/flash)	101	0xEF00 0000
Boot from TWI host (slave mode)	110	0xEF00 0000
Boot from UART host (slave mode)	111	0xEF00 0000

Reset and Powerup

Table 19-2 describes the five types of resets. Note all resets, except system software, reset the core.

Hardware Reset

The processor chip reset is an asynchronous reset event. The $\overline{\text{RESET}}$ input pin must be deasserted to perform a hardware reset. For more information, see the product data sheet.

Table 19-2. Resets

Reset	Source	Result
Hardware reset	The $\overline{\text{RESET}}$ pin causes a hardware reset.	Resets both the core and the peripherals, including the Dynamic Power Management Controller (DPMC). Resets the no boot on software reset bit in SYSCR. For more information, see "System Reset Configuration Register (SYSCR)" on page 19-4.
System software reset	Writing b#111 to bits [2:0] in the system MMR SWRST at address 0xFFC0 0100 causes a system software reset.	Resets only the peripherals, excluding the RTC (Real-Time Clock) block and most of the DPMC. The system software reset clears bit 4 (no boot on software reset) in the SYSCR register. Does not reset the core. Does not initiate a boot sequence.
Watchdog timer reset	Programming the watchdog timer appropriately causes a watchdog timer reset.	Resets both the core and the peripherals, excluding the RTC block and most of the DPMC. The software reset register (SWRST) can be read to determine whether the reset source was the watchdog timer.
Core double-fault reset	If the core enters a double-fault state, a reset can be caused by unmasking the core double fault reset mask bit in the system interrupt controller interrupt mask register (SIC_IMASK).	Resets both the core and the peripherals, excluding the RTC block and most of the DPMC. The SWRST register can be read to determine whether the reset source was core double fault.
Core-Only software reset	This reset is caused by executing a RAISE1 instruction or by setting the software reset (SYSRST) bit in the core debug control register (DBGCTL) via emulation software through the JTAG port. The DBGCTL register is not visible to the memory map.	Resets only the core. The peripherals do not recognize this reset.

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 BMODE state.

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_{DD} or V_{SS}. The pins and the corresponding bits in SYSCR configure the Boot mode that is employed after hardware reset or System Software reset. See the *ADSP-BF53x/BF56x Blackfin Processor Programming Reference* for further information.

System Reset Configuration Register (SYSCR)

The values sensed from the BMODE[2:0] pins are latched into the system reset configuration register (SYSCR) when the $\overline{\text{RESET}}$ pin is deasserted. The values are made available for software access and modification after the hardware reset sequence. Software can modify only the no boot on software reset bit in this register. Setting this bit has effect only in the case of a core-only software reset. All other types of reset clear this bit when issued. See Table 19-2.

The various configuration parameters are distributed to the appropriate destinations from SYSCR (see Figure 19-1).

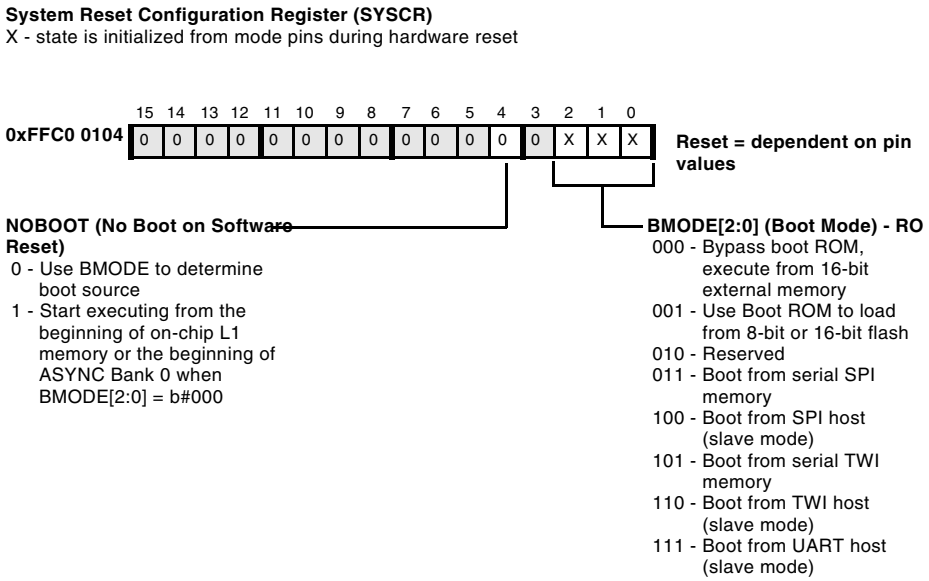


Figure 19-1. System Reset Configuration Register

Software Resets and Watchdog Timer

A software reset may be initiated in three ways:

- By the watchdog timer, if appropriately configured
- By setting the system software reset field in the software reset register (see Figure 19-2)
- By the `RAISE 1` instruction

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

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

When L1 instruction memory is configured as cache, make sure the system software reset sequence has been read into the cache.

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

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

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

Software Reset Register (SWRST)

A software reset can be initiated by setting bits [2:0] in the system software reset field in the software reset register (SWRST). Bit 3 can be read to determine whether the reset source was core double fault. A core double fault reset resets both the core and the peripherals, excluding 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 and bit 13, respectively, indicate whether the software watchdog timer or a core double fault has generated a software reset. Bits [15:13] are read-only and cleared when the register is read. Bits [3:0] are read/write.

When the BMODE pins are not set to b#000 and the no boot on software reset bit in SYSCR is set, the processor starts executing from the start of on-chip L1 memory. In this configuration, the core begins fetching instructions from the beginning of on-chip L1 memory.

When the BMODE pins are set to b#000 the core begins fetching instructions from address 0x2000 0000 (the beginning of async bank 0).

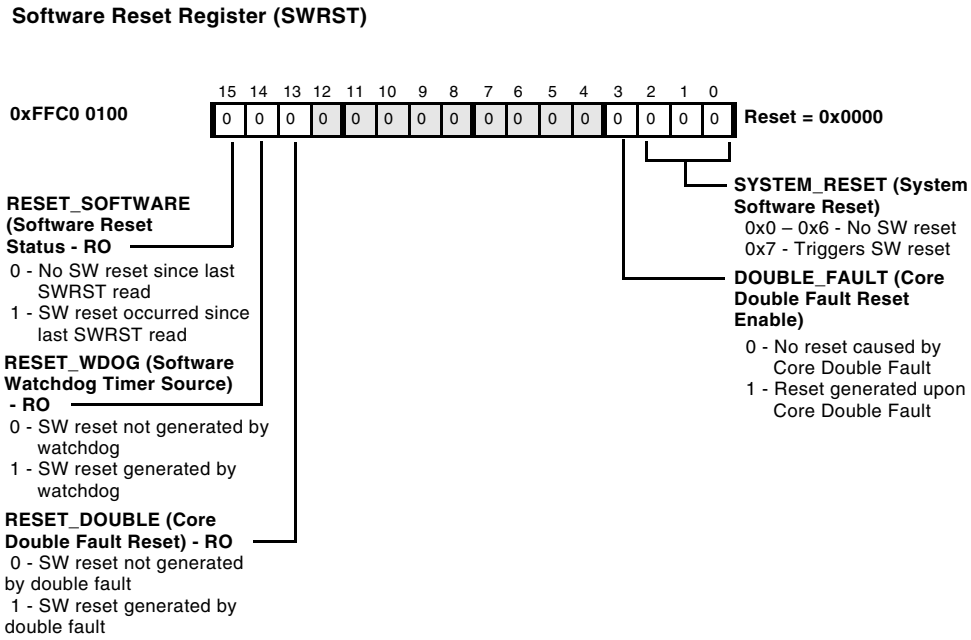


Figure 19-2. Software Reset Register

Core-Only Software Reset

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

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

Core and System Reset

To perform a system and core reset, use the code sequence shown in Listing 19-1.

Listing 19-1. Core and System Reset

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

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

/* Core reset - forces reboot */

RAISE 1 ;
```

Reset Vector

When reset releases in no-boot mode (BMODE=b#000), the processor starts fetching and executing instructions from off-chip memory at address 0x2000 0000. In all other boot modes the processor starts program execution at address 0xEF00 0000 which is populated by the on-chip boot ROM.

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. If bit 4 of the SYSCR register

is set, 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.

Neither hardware nor the kernel initializes the EVT1 register in no-boot mode. Programs should initialize EVT1 before issuing software resets in no-boot environments.

Servicing Reset Interrupts

The processor services a reset event just like other interrupts. The reset interrupt has top-priority. Only emulation events have higher priority. When it comes out of reset, the processor is in supervisor mode and has full access to all system resources. To enter the user mode, the reset service routine must initialize the RETI register and terminate by an RTI instruction. The code examples in Listing 19-2 and Listing 19-3 show the least instructions required to handle the reset event. See the *ADSP-BF53x/BF56x 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. The following listing shows how this is accomplished.

Listing 19-2. Exiting Reset to User Mode

```
_reset:
   P1.L = LO(_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 */
...
```

Listing 19-3. Exiting Reset by Staying in Supervisor Mode

```
_reset:
   P0.L = LO(EVT15) ;  /* Point to IVG15 in Event Vector Table */
   P0.H = HI(EVT15) ;
   P1.L = LO(_isr_IVG15) ; /* Point to start of IVG15 code */
   P1.H = HI(_isr_IVG15) ;
   [P0] = P1 ;         /* Initialize interrupt vector EVT15 */

   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 */
 ...
```

The reset handler most likely performs additional tasks not shown in the examples above. Stack pointers and EVTx registers are initialized here.

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.

Booting Process

After reset, 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 19-3 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.

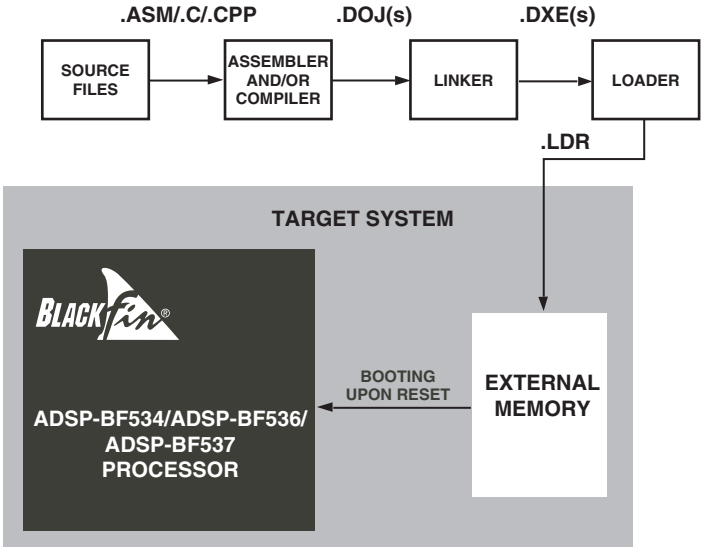


Figure 19-3. Project Flow for a Standalone System

Figure 19-4 shows the boot stream contained in a flash memory device, which could be of parallel or serial type. In host boot scenarios the non-volatile memory more likely connects to the host processor rather than directly to the Blackfin. After reset, the headers are read and parsed by the on-chip boot ROM, and processed block by block. Payload data is copied to destination addresses, either in on-chip L1 memory or off-chip SRAM/SDRAM.

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) is not supported. These memory locations are used by the boot kernel for intermediate storage of block header information and cannot be initialized at boot time. After booting, this memory range can be used by the application during runtime.

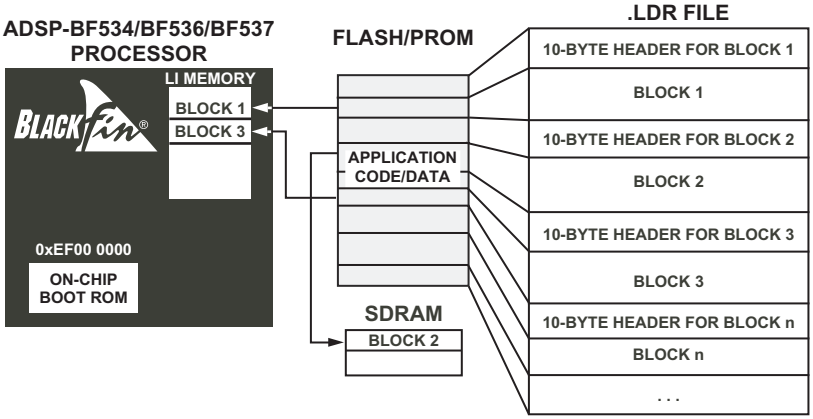


Figure 19-4. Booting Process

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 is done to the content of the boot ROM when silicon is updated.

Header Information

As shown in Figure 19-5, each 10-byte header within the loader file consists of a 4-byte `ADDRESS` field, a 4-byte `COUNT` field, and a 2-byte `FLAG` field.

This 10-byte header, which precedes each block in the loader file, contains the following information used by the on-chip boot ROM during the boot process:

- **ADDRESS** (4 bytes)—the target address, to which the block boots to within memory
- **COUNT** (4 bytes)—the number of data bytes in the block. The COUNT field can be any 32-bit value including zero.
- **FLAG** (2 bytes)—block type and control commands

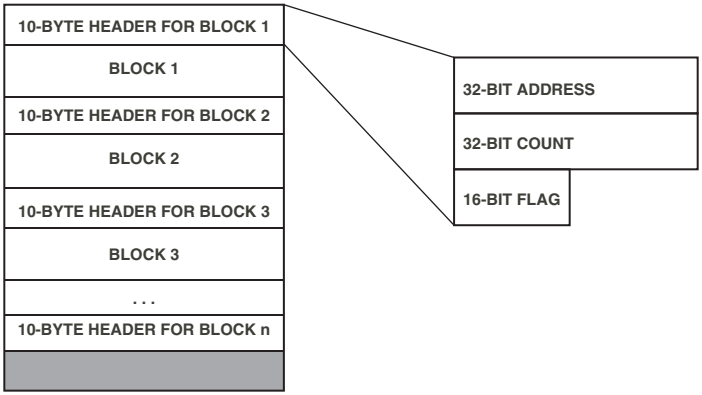


Figure 19-5. 10-Byte Header Contents

Figure 19-6 shows the individual flag word bits.

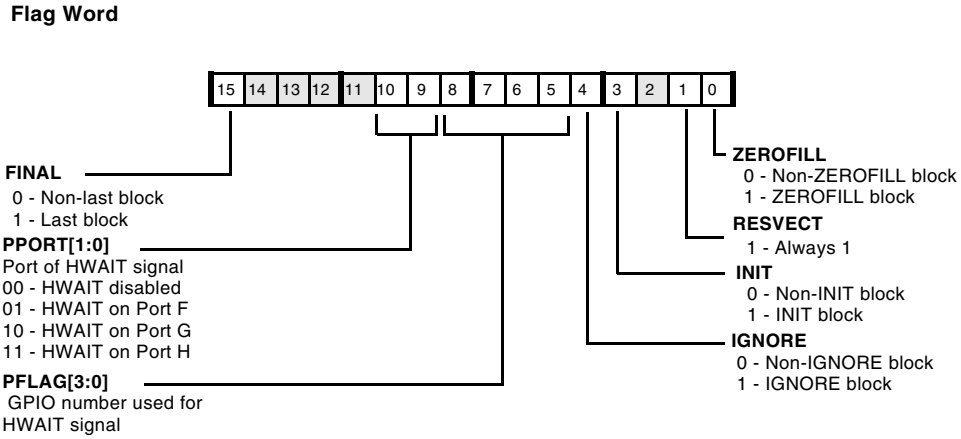


Figure 19-6. Individual Control Bits of the Flag Word Register

Additional information for the FLAG word bits includes:

- **ZEROFILL**

  Indicates that the block is a buffer with zeros. Blocks that are ZEROFILL have no payload data. They simply instruct the on-chip boot ROM to zero COUNT bytes starting from ADDRESS in memory. This yields a condensed loader file for applications with large zero buffers. It is also very helpful for ANSI-C compliant projects which often require large buffers to be zeroed during boot time.

- **RESVECT**

  This bit has no function on ADSP-BF534, ADSP-BF536, and ADSP-BF537 processors. For compatibility with Blackfin derivatives, it is always set to 1, indicating a reset vector address of 0xFFA0 0000.

  After a hardware reset, the reset vector (stored in the EVT1 register) is set to 0xFFA0 0000. If bit 4 (no boot on software reset) of the SYSCR register is set and a software reset is issued, the processor vectors to the address set in the EVT1 register. This reset vector can be reconfigure to another address during runtime and hence, an application can vector to an address other than 0xFFA0 0000 after a software reset. If the reset vector is modified during runtime, ensure that the reset vector address within the EVT1 register is a valid instruction address. This address can be internal instruction memory, SDRAM memory, or asynchronous memory. The EVT1 register does not have a default value. The value within this register is retained after a reset is issued. When BMODE = 000, the on-chip boot ROM is bypassed and the user's application must initialize the EVT1 register before issuing a software reset.

- **INIT**

  An initialization block (INIT block) is a block of code that executes before the actual application code boots over it. When the on-chip boot ROM detects an INIT block, it boots the block into internal memory and makes a `CALL` to it (initialization code must have an `RTS` at the end). After the initialization code is executed, it is typically overwritten with application code. See Figure 19-8. In the special case where the COUNT value of an INIT block is 0, the 10-byte header behaves like a command that instructs the boot kernel to simply issue the CALL command to the ADDRESS location.

- **IGNORE**

  Indicates a block that is not booted into memory. It instructs the boot ROM to skip `COUNT` bytes of the boot stream. In master boot modes, the boot ROM can just modify its source address pointer. In slave boot modes, the boot ROM actively boots in the payload data to a single location in memory. This essentially trashes the block. The current VisualDSP++ tools support `IGNORE` blocks for global headers only (currently the 4-byte DXE count, see "Multi-Application (Multi-DXE) Management" on page 19-26).

- **PFLAG**

  This field is used in all boot modes. It tells the boot ROM which GPIO pin is used for the `HWAIT` feedback strobe. If `PPORT` is zero, the `PFLAG` field is ignored. For more information, see "Host Wait Feedback Strobe (HWAIT)" on page 19-19.

- **PPORT**

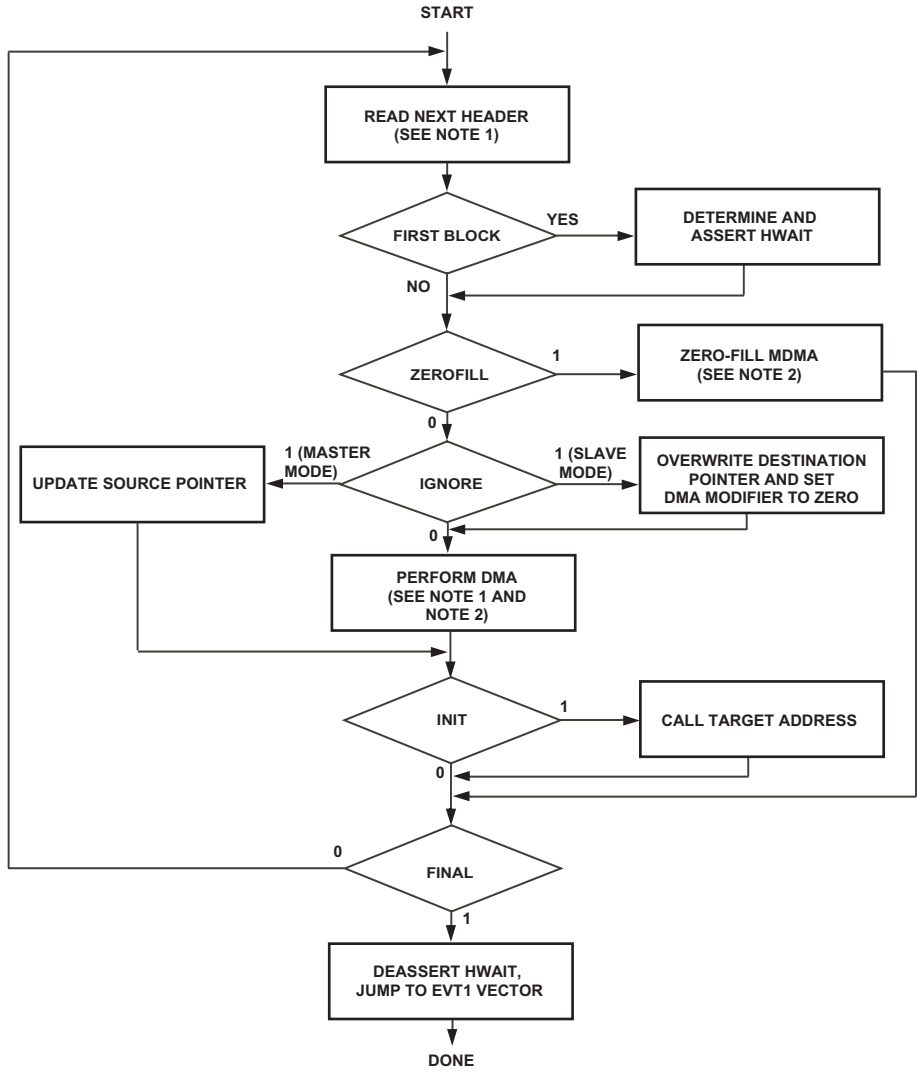
  This field tells the boot ROM whether the `HWAIT` signal should be activated and on which port it should appear. If enabled, the `PFLAG` field reports the respective GPIO number. For more information, see "Host Wait Feedback Strobe (HWAIT)" on page 19-19.

- **FINAL**

  Indicates boot process is complete after this block. After processing a FINAL block, the on-chip boot ROM jumps to the reset vector address stored in the EVT1 register. The EVT1 register is set to the start of L1 instruction memory upon a hardware reset. The processor is still in supervisor mode and in the lowest priority interrupt (IVG15) when it jumps to L1 memory for code execution.

For every boot block, the individual flags are processed by the algorithm shown in Figure 19-7. The IGNORE flag requires different processing for slave versus master boot modes. Where master modes can simply increment their source address pointer, slave modes actively consume and "trash" the payload data.

The PFLAG and PPORT bit fields are processed only once while processing the header of the first boot block. If the setting changes for later blocks, that change is ignored.



NOTE 1: TEMPORARILY, DEASSERT HWAIT TO REQUEST MORE DATA FROM HOST DEVICE

NOTE 2: DMA AND MDMA BLOCKS SUPPORT BYTE COUNTS FROM 0 TO 2^{32}-1 BY PERFORMING MULTIPLE DMA SEQUENCES AND GUARDING AGAINST ZERO

Figure 19-7. Boot Block Flag Processing

Host Wait Feedback Strobe (HWAIT)

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. This is especially critical when processing initialization code or zero-filling memory where the boot kernel may not be ready for additional data. It is used primarily for slave boot modes, such as with the UART and SPI slave modes, but is available to all boot modes.

The HWAIT strobe does not stick to a certain pin. Rather, any of the 48 GPIOs can be used for this purpose. The flag word of the first block in the boot stream tells the boot kernel which GPIO to use. The 2-bit field PPORT enables the HWAIT functionality and determines whether the HWAIT strobe operates at port F, G, or H. See Figure 19-6 on page 19-14 for the bit settings.

The 4-bit field PFLAG (also shown in Figure 19-6 on page 19-14) then determines which GPIO pin of the chosen port is used to handshake with the host. For example, if PPORT=b#10 and PFLAG=b#0110 then HWAIT strobe is activated on PG6.

For individual boot modes, certain GPIOs on port F cannot be used for HWAIT functionality as they are used for other purposes. Table 19-3 shows how the port function enable and muxing registers are programmed at boot time.

To specify which GPIO is to be used as the HWAIT signal use the -PFLAG switch in the VisualDSP++ loader options property page. For example, to use PG13 as HWAIT, in the additional options tab of the loader property page write:

```
-pflag PG13
```

The switch -pflag takes as an argument PF#, PG#, or PH#, where # is the number of the GPIO in the corresponding port, for example, PG10.

Table 19-3. Settings for Port Function Enable and Muxing Registers

BMODE	Boot Mode	PORTF_FER at boot time	PORT_MUX at boot time
000	Bypass ROM, No-Boot	–	–
001	Parallel Flash on /AMS0	–	–
010	Reserved	PF1	–
011	SPI Memory	PF11, PF12, PF13, PF14	–
100	SPI Host	PF11, PF12, PF13, PF14	–
101	TWI Memory	–	–
110	TWI Host	–	–
111	UART Host	PF0, PF1	–

The signal polarity of the HWAIT strobe is programmable by an external pull-up resistor in the 10 kΩ range. A pull-down resistor instructs the HWAIT signal to be active high. In this case the host is permitted to send data when HWAIT is low, but should pause while HWAIT is high. This is the mode used in SPI slave booting on other Blackfin derivatives. Similarly, active-low behavior is programmed by a pull-up resistor.

After reset, the boot kernel waits to receive the first 10-byte block header. At this time the HWAIT pin is not yet actively driven. Instead, the resistor pulls the signal to the inactive state, encouraging the host to send data. After receiving the ten bytes, the boot kernel knows whether to activate the HWAIT signal and which GPIO to use. If PPORT is not zero, the boot kernel first senses the polarity on the respective GPIO pin. Then, it enables the output driver and inverts the signal polarity to immediately hold off the host. The signal is not released again until the boot kernel is ready for further data, or when a receive DMA has been started. As soon as the DMA completes, HWAIT becomes active again.

This conservative behavior lets the host send data with high data rates without knowing the structure of the boot stream. The most critical speed path during the entire boot procedure is the time during when the kernel received the 10th bytes and must still evaluate the block header before it can drive HWAIT. For the highest data rate, the host is encouraged to perform these steps:

1. Send the first 10 bytes.
2. Wait until HWAIT goes active.
3. For every further byte, wait until HWAIT is inactive and then send the next byte.

Final Initialization

After the successful download of the application into the bootable memory, and before jumping to the EVT1 vector address, the boot kernel does some housekeeping work. Most of the used registers are changed back to their default state, but some register values may differ for the individual boot modes. These registers are reset to 0x0:

- SPI_CTL
- UART0_GCTL
- UART0_IER
- TWI_CONTROL
- MDMA_S0_CONFIG
- MDMA_D0_CONFIG
- MDMA_S1_CONFIG
- MDMA_D1_CONFIG
- DMA7_CONFIG

- DMA8_CONFIG
- PORTF_FER
- PORT_MUX

Initialization Code

Initialization code (INIT code) allows the execution of a piece of code before the actual application is booted into the processor. This code can serve a number of purposes including initializing the SDRAM controller, the SPI baud rate, or EBIU wait states for faster boot time. The INIT code is added to the beginning of the loader file stream via the elfloader `-Init Init_Code.DXE` command-line switch, where `Init_Code.DXE` refers to the user-provided custom initialization code executable, that is, by a separate VisualDSP++ project. Figure 19-8 shows a boot stream example that performs these steps:

1. Boot INIT code into L1 memory.
2. Execute INIT code.
3. The INIT initializes the SDRAM controller and returns.
4. Overwrite INIT code with final application code.
5. Boot data/code into SDRAM.
6. Continue program execution with block n.

When the on-chip boot ROM detects a block with the `INIT` bit set, it first boots it into Blackfin memory and then executes it, by issuing a `CALL` to its target address. For this reason, every INIT code 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.

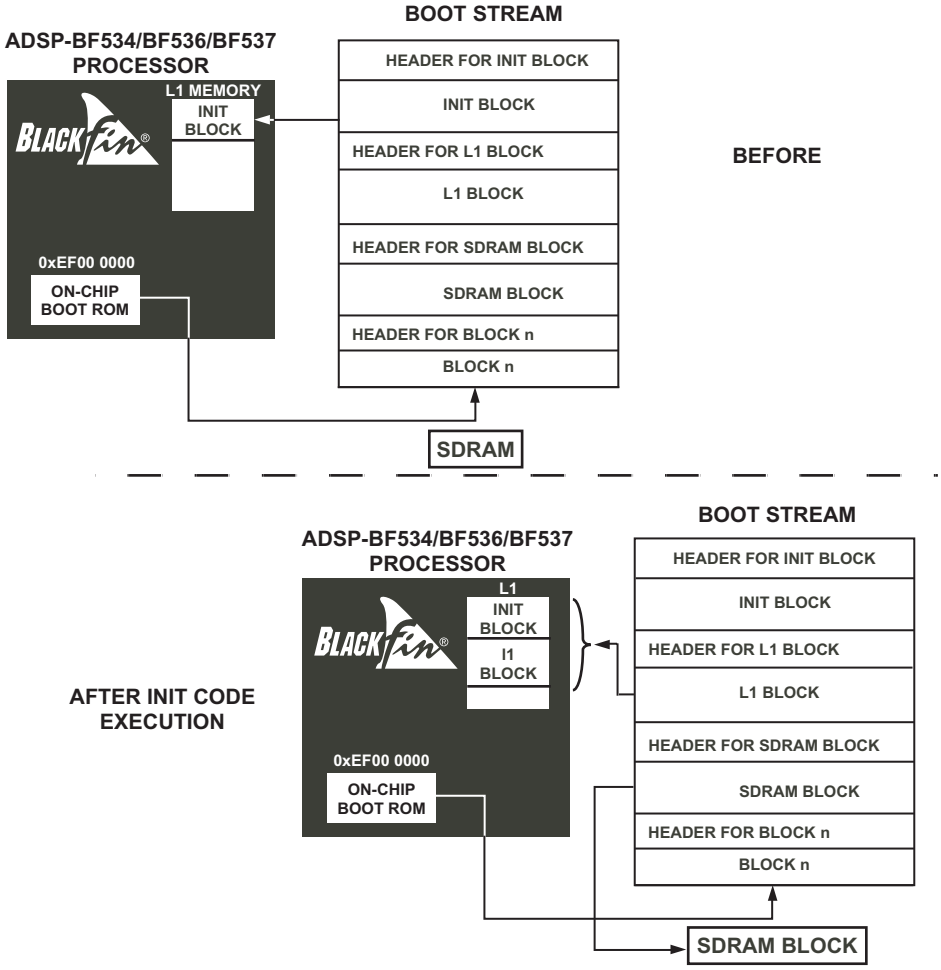


Figure 19-8. Initialization Code Execution/Boot

Programs must be sure to save all processor registers modified by INIT code and to restore them before the INIT code returns. At a minimum, it is recommended that every INIT code saves the ASTAT, RETS, and the Rx and Px registers used in the INIT code. The INIT code can perform push and pop operations through the stack pointer SP. The boot kernel provides sufficient stack space in scratchpad memory (0xFFB0 0000 – 0xFFB0 0FFF).

Listing 19-4 shows an example INIT code file that demonstrates the setup of the SDRAM controller.

Listing 19-4. Example INIT Code (SDRAM Controller Setup)

```
#include <defBF537.h>
.section program;
/*****************************************/
  [--SP] = ASTAT;  // Save registers onto Stack
  [--SP] = RETS;
  [--SP] = (R7:0);
  [--SP] = (P5:0);
/*****************************************/
/*******INIT Code Section******************/
/*******SDRAM Setup************/
Setup_SDRAM:
  P0.L = LO(EBIU_SDRRC);
  P0.H = HI(EBIU_SDRRC);  // SDRAM Refresh Rate Control Register
  R0 = 0x074A(Z);
  W[P0] = R0;
  SSYNC;

  P0.L = LO(EBIU_SDBCTL);
  P0.H = HI(EBIU_SDBCTL);// SDRAM Memory Bank Control Register
  R0 = EBCAW_8|EBSZ_16|EBE(Z);//This is just an example!
  W[P0] = R0;
  SSYNC;
```

```
  P0.L = LO(EBIU_SDGCTL);
  P0.H = HI(EBIU_SDGCTL);  //SDRAM Memory Global Control Register
  R0.H = HI(PSS|TWR_2|TRCD_3|TRP_3|TRAS_6|CL_3|SCTLE);
  R0.L = LO(PSS|TWR_2|TRCD_3|TRP_3|TRAS_6|CL_3|SCTLE);
  [P0] = R0;
  SSYNC;
/****************************************/
  (P5:0) = [SP++];   // Restore registers from Stack
  (R7:0) = [SP++];
  RETS = [SP++];
  ASTAT = [SP++];
/****************************************/
  RTS;
```

Typically, INIT code consists of a single section and is represented by a single block within the boot stream. This block has, of course, the `INIT` bit set. Nevertheless, an INIT block can also consist of multiple sections. Then, multiple blocks represent the INIT code within the boot stream. Only the last block has the `INIT` bit set. The elfloader utility ensures that the last of these blocks vectors to the INIT code's entry address. It instructs the on-chip boot ROM to execute a `CALL` instruction to the given `ADDRESS`.

Although INIT code `.DXE` files are built through their own VisualDSP++ projects, they differ from standard projects. INIT codes provide a callable sub-function only, and thus, they look more like a library than an application. An INIT code is always a heading for the regular application code. Consequently, regardless whether the INIT code consists of one or multiple blocks, it is not terminated by a `FINAL` bit indicator, which would cause the boot ROM to terminate the boot process.

Multi-Application (Multi-DXE) Management

In addition to pre-boot initialization, the INIT code feature can also be used for boot management. A loader file (.LDR) can store multiple applications if multiple executable (.DXE) files are listed on the elfloader command line. The elfloader utility creates multiple boot streams with the individual executable files appended one after the other with the INIT code DXE (if any) located at the beginning (see Figure 19-9).

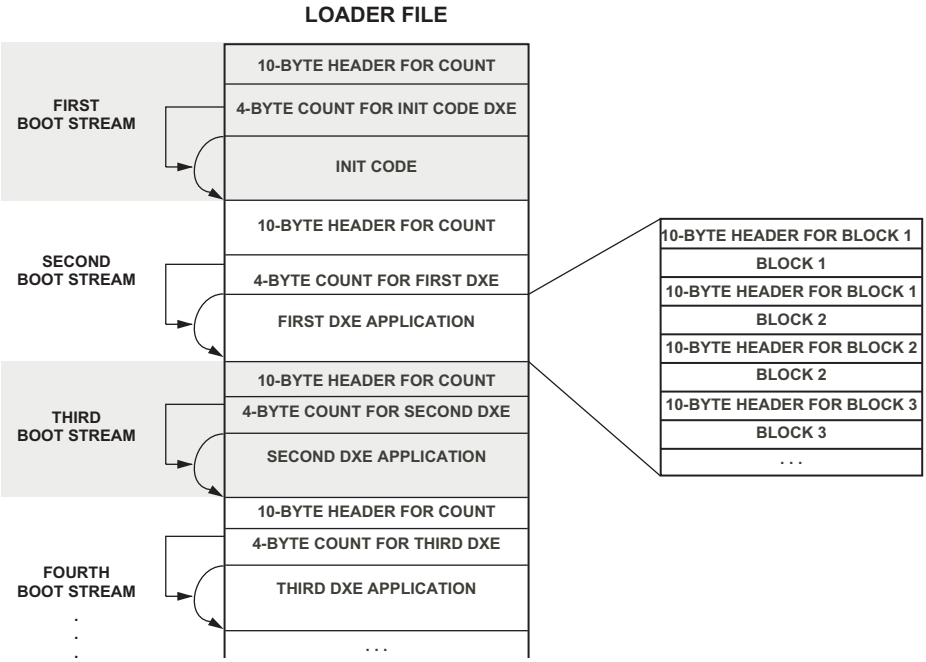


Figure 19-9. Multi-DXE Loader File Contents

The Blackfin processor loader file (.LDR) structure can be used to determine the boundary between the individual boot streams stored in the external memory, and hence, provides the ability to boot in a specific DXE application.

Each DXE application is represented by a complete boot stream in the .LDR file. By definition, each boot stream is preceded by a special IGNORE block. Currently, this IGNORE block contains a 4-byte pointer value, which is the number of bytes contained within the DXE application including headers. In other words, it is the offset to the next DXE application.

This relative Next DXE Pointer (NDP) guides to the start address of the next boot stream. In the most likely case, when one boot stream appends to the other contiguously, the NDP also represents the byte count of the boot stream it is heading, with the exception of the first IGNORE block. This is why the NDP is often called "DXE byte count" as in Figure 19-9. Note that each IGNORE block is headed by a 10-byte header.

Currently, the initial IGNORE block contains only 4 bytes of data to store the NDP. In future VisualDSP++ versions the length of initial IGNORE blocks may increase without further notice.

With this NDP, the boot streams are structured like a chained list. The user can essentially "jump" through whole DXE applications within the .LDR file until the DXE application chosen to be booted in is reached.

User-callable Boot ROM Functions

The boot ROM contains a set of functions that can be called at runtime from the user code. The major purpose of these functions is to support multi-DXE and second-stage loader scenarios.

Booting a Different Application

As discussed in "Multi-Application (Multi-DXE) Management" on page 19-26, Blackfin hardware must often execute different applications. In slave boot modes the host device may be required to pull the processor's $\overline{\text{RESET}}$ pin and to provide new application data through the interface

chosen by the BMODE pins. In the three master modes that boot from flash, SPI or TWI memory, this is not possible. Therefore, the on-chip boot ROM provides special support for these three boot modes.

When a different application (.DXE) needs to be booted into a running Blackfin, it should be booted into the processor's internal memory. Programs do not require simulating the built-in boot kernel—instead, the original kernel can be reused for this purpose. The boot ROM provides the proper entry addresses for the three master boot modes. All three functions expect register R7 to hold the start address where the boot stream to be booted resides. For flash boot the following sequence may boot in a stream that is stored in asynchronous memory bank 1.

```
#include <defBF537.h>   /* provides function entry addresses */

    P0.H = HI(_BOOTROM_Boot_DXE_Flash) ;
    P0.L = LO(_BOOTROM_Boot_DXE_Flash) ;
    R7.H = HI(0x20100000) ;   /* start of async bank 1 */
    R7.L = LO(0x20100000) ;
    JUMP (P0) ;               /* jump to Boot ROM */
```

This example assumes that the $\overline{\text{AMS1}}$ strobe has already been activated. The function determines whether an 8-bit or a 16-bit memory device is connected. Note that the boot stream may also reside in SRAM or SDRAM memory.

The SPI master boot from flash version takes two more parameters: R6, which holds the GPIO on port F that the SPI memory's $\overline{\text{CS}}$ input connects to, and R5 which holds the value that is written to the SPI_BAUD register internally. Since the routine writes this value directly into the PORTFIO_DIR register, other GPIOs on port F might be impacted. The following example loads a boot stream from address 0 of an SPI memory connected to the PF4 pin:

```
#include <defBF537.h>  /* provides function entry addresses */
    P0.H = HI(_BOOTROM_Boot_DXE_SPI) ;
```

```
P0.L = LO(_BOOTROM_Boot_DXE_SPI) ;
R7 = 0 (Z) ;                /* SPI address is zero */
R6 = PF4 (z) ;              /* SPI's /CS connects to PF4 pin */
R5 = 0x0085(Z) ;            /* SPI clock divider */
JUMP (P0) ;                 /* jump to Boot ROM */
```

Note the following additional points on master booting from flash.

- For standard booting after reset, connect the SPI memory's chip select to the PF10 pin. The example above assumes a second SPI device is connected to the PF4 pin.
- The optimal SPI_BAUD value written to R5 depends on the SCLK clock rate and the timing parameters of the particular SPI memory device.

TWI master boot from flash takes three input parameters where R7 is again the start address and R6 holds the 7-bit TWI chip select address. If three address inputs of a TWI memory are named A2, A1 and A0, then R6 should provide binary b#1010.A2.A1.A0.x in its lower byte. In this case, R5 holds the clock divider value that is written to the TWI_CLKDIV register as shown in the example below.

```
#include <defBF537.h>    /* provides function entry addresses */

    P0.H = HI(_BOOTROM_Boot_DXE_TWI) ;
    P0.L = LO(_BOOTROM_Boot_DXE_TWI) ;
    R7 = 0 (Z) ;          /* TWI address is zero */
    R6 = 0xA2 (Z) ;       /* A2=0, A1=0, A0=1 */
    R5 = 0x0811 (Z) ;     /* default clock divider */
    JUMP (P0) ;           /* jump to Boot ROM */
```

Determining Boot Stream Start Addresses

In some cases the individual boot streams reside at hard coded start addresses, for example, at page boundaries of flash devices. If multiple boot streams are generated by a single pass of the VisualDSP++ elfloader

utility, they might be appended one after another and the start addresses vary on build variables. When the individual boot streams are chained by the Next DXE Pointer (NDP) scheme as described in "Multi-Application (Multi-DXE) Management" on page 19-26, a set of Boot ROM functions help to determine the start addresses of the individual boot streams. Again, different functions are available for the three master boot modes.

The user is responsible for initializing the stack and saving and restoring all needed registers prior to calling these functions.

The following example boots the third boot stream contained in the flash connected to AMS0. Register R7 simply requires the number of the boot stream to be loaded. The parameter passed in R6 tells the function where to find the chain of boot streams.

```
#include <defBF537.h>  /* provides function entry addresses */

    P0.H = HI(_BOOTROM_Get_DXE_Address_Flash) ;
    P0.L = LO(_BOOTROM_Get_DXE_Address_Flash) ;
    R7 = 2 (Z) ;   /* DXE #3 - start counting from zero */
    R6.H = HI(0x20000000) ;   /* start of async bank 0 */
    R6.L = LO(0x20000000) ;
    CALL (P0) ;   /* call to Boot ROM */
                  /* start address of DXE #3 is returned in R7 */
                  /* R7 is passed to the next function */
    P0.L = LO(_BOOTROM_Boot_DXE_Flash) ;
    JUMP (P0) ;   /* boot DXE #3 */
```

While parsing the chain the function tests for every individual boot stream whether it is homed in an 8-bit or a 16-bit device. Therefore the chain may cross different memory types heterogeneously. However, the data width must not change within the same boot stream.

The Get_DXE_Address functions require a valid stack. These functions may use the address range between 0xFF80 7FF0 and 0xFF80 7FFF for local data storage.

Care must be taken when the Get_DXE_Address functions are used in a manner other than as shown. Because of space restrictions in the boot ROM, these functions do not save and restore the registers modified locally. All core registers, as well as DMA, GPIO, and port control registers, may be subject to change by the routines. Programs must initialize the stack and save all required registers to the stack, as the stack pointer SP is returned correctly.

The _BOOTROM_Get_DXE_Address_SPI function requires the same three parameters as its _BOOTROM_Boot_DXE_SPI counterpart. Register R7 holds the number of the DXE whose address is to be returned by the function.

Enable the SPI signals in the PORTF_FER register before calling this function.

```
R1 = PF13 | PF12 | PF11 (Z);
P1.L = LO(PORTF_FER);
P1.H = HI(PORTF_FER);
[P1 ] = R1; /* function enable on SPI signals */

R5 = 0x0085 (Z); /* SPI baud rate */
R6 = 0xA; /* R6 holds the addressed memory device chip select */
R7 = 0x3; /* holds the DXE # to be booted in. */

[--SP] = R6;
[--SP] = R5;

p5.l = lo(_BOOTROM_GET_DXE_ADDRESS_SPI);
p5.h = hi(_BOOTROM_GET_DXE_ADDRESS_SPI);

call (p5); /* GET DXE ADDRESS */
```

```
/**************************************************
          BOOT DXE
          R7 holds the DXE address
**************************************************/
R5 = [SP++];
R6 = [SP++];

p5.l = lo(_BOOTROM_BOOT_DXE_SPI);
p5.h = hi(_BOOTROM_BOOT_DXE_SPI);
jump (p5);
```

Also the `_BOOTROM_Get_DXE_Address_TWI` function expects the same parameters as `_BOOTROM_Boot_DXE_TWI`. Additionally, pointer `P2` should point to a 32-bit scratch location.

```
_main:
/**************************************************
user must setup the stack pointer as well as save
and restore needed resources
**************************************************/
sp.h = 0xFFB0;
sp.l = 0x1000;
/**************************************************
           GET DXE ADDRESS
           R7 holds the DXE #
**************************************************/
R5 = 0x0811 (Z);  /* TWI_CLKDIV value to produce 400 KHz SCL in a
~30% duty cycle */
R6 = 0xA0 (Z);  /* R6 holds the addressed memory device */
R7 = 0x3;  /* holds the DXE # to be booted in */

p2.h = 0xff90;  /* P2 holds a 4 byte scratch location */
p2.l = 0x0000;

[--SP] = R6;
```

```
[--SP] = R5;

p5.l = lo(_BOOTROM_GET_DXE_ADDRESS_TWI);
p5.h = hi(_BOOTROM_GET_DXE_ADDRESS_TWI);
call (p5); /* GET DXE ADDRESS */

/**************************************************
           BOOT DXE
           R7 holds the DXE address
**************************************************/
R5 = [SP++];
R6 = [SP++];

p5.l = lo(_BOOTROM_BOOT_DXE_TWI);
p5.h = hi(_BOOTROM_BOOT_DXE_TWI);
jump (p5);
_main.END:
```

When the `Get_DXE_Address` function is used in an INIT code application, care must be taken to not restore these registers before returning to the boot ROM code:

- `R0` for flash booting
- `R3` for SPI master booting
- `R4` for TWI master booting

For example, when booting from SPI, do not restore `R3`, when booting from TWI do not restore `R4`, and so on.

When the processor returns to the on-chip boot ROM after the `RTS` instruction, the on-chip boot ROM continues booting from the location stored in the register for the corresponding boot mode (`R0`, `R3`, `R4`).

Specific Blackfin Boot Modes

The Blackfin processors feature seven different boot modes (and one no-boot mode).

- "Bypass (No-Boot) Mode (BMODE = 000)" on page 19-34
- "8-Bit Flash/PROM Boot (BMODE = 001)" on page 19-36
- "16-Bit Flash/PROM Boot (BMODE = 001)" on page 19-40
- "SPI Slave Mode Boot From SPI Host (BMODE = 100)" on page 19-48
- "SPI Master Mode Boot from SPI Memory (BMODE = 011)" on page 19-42
- "TWI Master Boot Mode (BMODE = 101)" on page 19-53
- "TWI Slave Boot Mode (BMODE = 110)" on page 19-55
- "UART Slave Mode Boot via Master Host (BMODE = 111)" on page 19-56

This section discusses the hardware connections required for each boot mode and explains topics specific to the individual modes.

Bypass (No-Boot) Mode (BMODE = 000)

In bypass mode (`BMODE = 000`), the processor starts code execution from address 0x2000 0000. This is the first address in the asynchronous memory bank 0. The memory device must therefore be connected to the $\overline{\text{AMS0}}$ strobe. Code execution from 8-bit memory is not supported. Bypass mode always requires 16-bit wide memory, which may be built from a single 16-bit device or two parallel 8-bit devices. Because the jump to the

0x2000 0000 address is the first action performed after powerup, the device connected is most likely a non-volatile memory such as a flash or EPROM device.

Figure 19-10 illustrates the pin-to-pin connections between the Blackfin processor and a 16-bit flash/PROM.

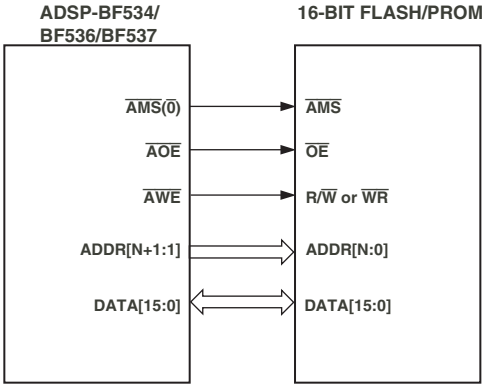


Figure 19-10. Connections Between a Blackfin Processor and a 16-Bit Flash/PROM

In addition to the signal connection shown in Figure 19-10, a pull-up resistor on the AMS0 chip select line may prevent any data corruption when the Blackfin processor is in any undefined state, for example, during powerup.

In bypass mode the content of the flash device is not formatted in any way. Instead it holds plain application code. The VisualDSP++ loader utility needs to be invoked in its "splitter mode" to create a proper file. Refer to the *VisualDSP++ Loader Manual* for more information. Execution from external 16-bit memory (BMODE = 000) is discussed in the application note *Running Programs from Flash on ADSP-BF533 Blackfin Processors (EE-239)*.

8-Bit Flash/PROM Boot (BMODE = 001)

Booting from 8-bit flash requires the same BMODE settings as with 16-bit flash. The two modes are differentiated by the first byte read in. While booting from 8-bit flash/PROM sounds cheaper and may result in smaller board space, there are also some disadvantages compared to the 16-bit mode: 8-bit mode supports only up to 512k byte devices directly, whereas 16-bit mode supports up to 1M byte. Because code execution from 8-bit external memory is not supported, in 8-bit mode the usage of the flash/PROM device is most likely restricted to booting purposes. The boot kernel assumes these conditions for the flash boot mode (BMODE = 001):

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

Since the EBIU on the Blackfin processor is 16 bits wide (hence no ADDR[0]), an 8-bit flash/PROM occupies only the lower 8 bits of the data bus (D[7:0]). Figure 19-11 illustrates the pin-to-pin connections between the Blackfin processor and an 8-bit flash/PROM.

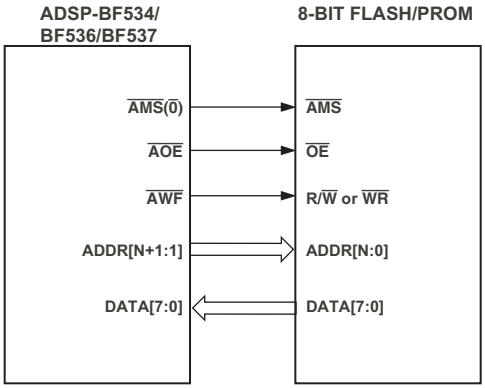


Figure 19-11. Connections Between a Blackfin Processor and an 8-Bit Flash/PROM

In some cases an additional pull-up resistor on the AMS0 strobe might protect the content of the flash device from being corrupted when the processor is in undefined state (that is, during powerup).

Figure 19-12 shows a loader file created for an 8-bit flash/PROM in Intel hex format. It is split into different sections to illustrate the loader file's structure:

1. Intel hex overhead
2. 10-Byte header for INIT code DXE count block, consisting of an ADDRESS of 10-byte seader, COUNT of 10-byte header, and FLAG of 10-byte header
3. INIT code DXE count block
4. 10-Byte header for block 1 of INIT code DXE, consisting of an ADDRESS of 10-byte header, count of 10-byte header, and FLAG of 10-byte header

5. 10-Byte header for block 2 of INIT code DXE, consisting of an ADDRESS of 10-byte header, count of 10-byte header, and FLAG of 10-byte header
6. DXE1 count block
7. 10-Byte header for block 1 of DXE1, consisting of an ADDRESS of 10-byte header, count of 10-byte header, and FLAG of 10-byte header
8. Block 1 of DXE1

```
020000040000FA
:0A0000004080A0FF0400000012006 1
:04000A005C000000 96
:20000E000000A0FF460000000200660167014005C00408E1180A48E1C0FF80E14A070097D7
:20002E00240008E1140A48E1C0FF80E101000097240008E1100A48E1C0FF00E18D9940E16F
:10004E00910000932400800400052701260110007 2
:0C005E000000A0FF020000000A00660184
:0A006A004080A0FF040000001200F7
:04007400 88020000FE
:20007800 0000A0FF7E020000028 0B161113E08E1082048E1E0FF00E1D00040E1A0FF00927A
:20009800 00E1D20040E1A0FF0092009200E1F60040E1A0FF009200E1F80040E1A0FF00925D
:2000B800 00E1FA0040E1A0FF009200E1FC0040E1A0FF009200E1FE0040E1A0FF009200E1BA
:2000D800 000140E1A0FF009200E1020140E1A0FF009200E1040140E1A0FF009200E106015F
:2000F800 40E1A0FF009200E1080140E1A0FF009200E10A0140E1A0FF00924EE1B0FF0EE14F
:20011800 00047E3208E1CC0148E1A0FF583E08E13C2048E1E0FF00E1C00040E1A0FF0092BF
:20013800 80E1008040009F0011007B0108E1CC0148E1A0FF50002000FF2F002008E13007FE
:20015800 48E1C0FF0095104A009708E1080748E1C0FF80E10000104A00923090709000200C
:20017800 11000020002000200020002000200020002000200020002080E1000081E10100A2E0D0
:20019800 062001900A901108010208501000 08E1300748E1C0FF0095004A009708E10807FC
:2001B800 48E1C0FF80E10000004A009230907090002008E1300748E1C0FF0095084A00979C
:2001D800 08E1080748E1C0FF80E10000084A009230907090002000000000048E1C0FF08E131
:2001F800 040A00E1B07B40E1B07B0093240048E1C0FF08E1080A00E1B07B40E1B07B0093FC
:20021800 240048E1C0FF08E1000A008279600856009724004 9E1272009E1040000600897F5
:20023800 240049E1272009E1060080E1FFFF0897240010000000FFE3D2FF08E1300748E1F3
:20025800 C0FF00E100000097240008E1340748E1C0FF00600097240008E1400748E1C0FFEC
:20027800 009501E1000F08550097240049E1272009E105000C9923E1C0001C55FB61AA6027
:20029800 9A5494560A9B2400D243056000E3260008E1000748E1C0FF009521E10001485416
:2002B800 010C0318AA60056021E100024854010C0318FA61056021E100044854010C03183D
:2002D800 9A61056021E100084854010C0410050CD81BDF2F0D60DD2F0AE100004AE18003BB
:0802F800 A2E00220000010004A
:00000001FF
```

Figure 19-12. Example Intel Hex Loader File

When this loader file is programmed into an 8-bit flash connected to asynchronous bank 0 of the Blackfin processor, the contents of memory (starting at location 0x2000 0000) viewed from the Blackfin processor look like Figure 19-13.

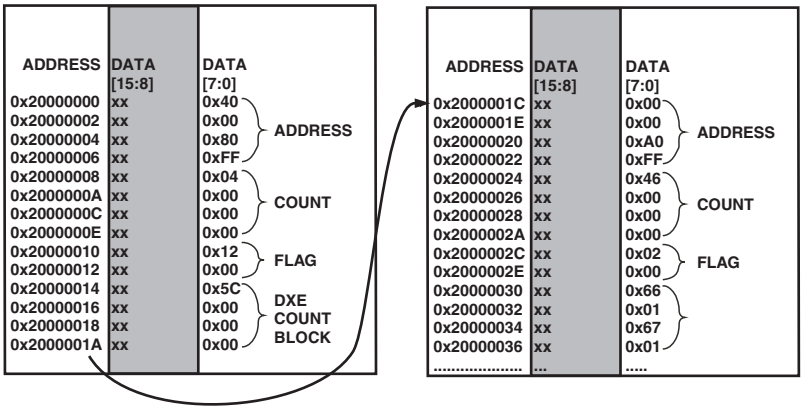


Figure 19-13. 8-Bit Flash/PROM Memory Contents Viewed From Blackfin Memory Window

Figure 19-14 shows the start of a boot sequence for an 8-bit flash/PROM boot.

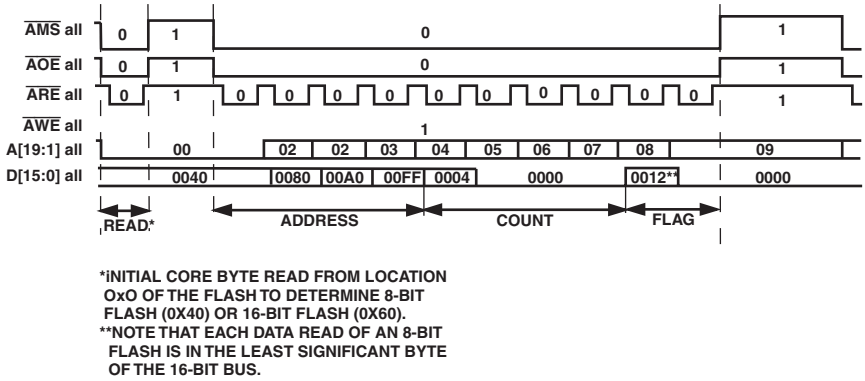


Figure 19-14. Timing Diagram for 8-Bit Flash Boot Sequence

The processor performs an initial core byte read of location 0x0 of the flash to determine the memory width of the flash. When booting from a FIFO in this mode, the first byte (which is part of the first 10-byte header contained within the loader file) must be sent twice—once for this initial core read and once for the actual boot sequence.

16-Bit Flash/PROM Boot (BMODE = 001)

The hardware connection for this boot scenario has already been shown in Figure 19-10. It is one of the advantages of the 16-bit flash boot mode that it requires the same hardware as the bypass mode. This not only enables the user to support two booting methods on a given board by a single jumper on the `BMODE0` pin, but it also enables programming to execute slow subroutines directly out of the flash/PROM at runtime.

A loader file for a 16-bit flash/PROM will be exactly the same as the one shown in Figure 19-12, except that the ADDRESS of the 10-byte header for the DXE count blocks will be 0xFF80 0060 or 0xFF80 0020 instead of

0xFF80 0040 as in the case of an 8-bit flash/PROM. This causes the first byte of the loader file to be 0x60 or 0x20 instead of 0x40. The on-chip boot ROM uses this first byte to determine whether an 8- or a 16-bit flash/PROM is connected. If the first byte is 0x20, it assumes a 16-bit flash/PROM device and performs 16-bit DMA operations. Note that 16-bit DMA requires the `ADDRESS` and `COUNT` fields to be even values. This is ensured by the VisualDSP++ elfloader utility. If the boot stream was generated by any other tool, and either `ADDRESS` or `COUNT` might be odd values, the first byte should be 0x60. Then, a 16-bit flash/PROM device is still assumed, but 8-bit DMA is performed, resulting in almost twice the boot time. If the first byte is 0x40, an 8-bit flash/PROM device is assumed.

When this loader file is programmed into a 16-bit flash connected to asynchronous bank 0 of the Blackfin processor, the contents of memory (starting at location 0x2000 0000) viewed from the Blackfin processor look like Figure 19-15.

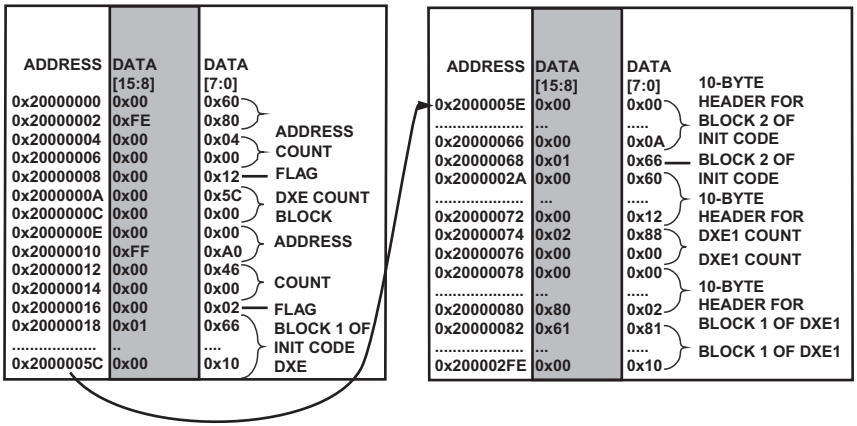


Figure 19-15. 16-Bit Flash/PROM Memory Contents Viewed From Blackfin Memory Window

Figure 19-16 shows the start of a boot sequence for a 16-bit flash/PROM boot.

The processor performs an initial core byte read of location 0x0 of the flash to determine the memory width of the flash. When booting from a FIFO, the first 16-bit word (which is part of first 10-byte header contained within the loader file) must be sent twice—once for this initial core read and once for the actual boot sequence.

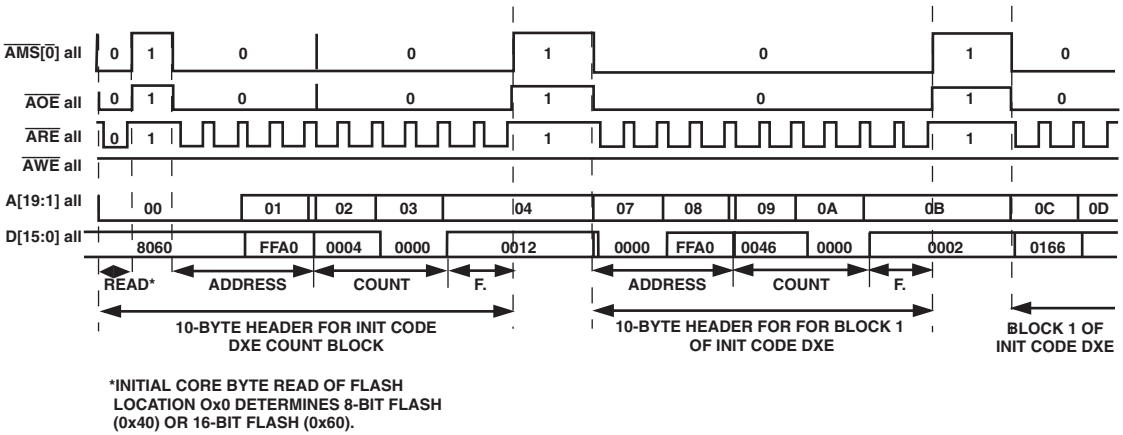


Figure 19-16. Timing Diagram for 16-Bit Flash Boot Sequence

SPI Master Mode Boot from SPI Memory (BMODE = 011)

For SPI master mode boot (`BMODE = 011`), the boot kernel assumes that the SPI baud rate is `SCLK/(2 x 133)` Kbit/s. SPI serial EEPROMs that are 8-bit, 16-bit, and 24-bit addressable are supported. The SPI uses the `PF10` output pin to select a single SPI EEPROM device. The SPI controller submits successive read commands at addresses 0x00, 0x0000, and

0x000000 until a valid 8-, 16-, or 24-bit addressable EEPROM is detected. It then begins clocking data into the beginning of L1 instruction memory.

For SPI master mode booting, the processor is configured as an SPI master connected to an SPI memory. Figure 19-17 shows the pin-to-pin connections needed for this mode.

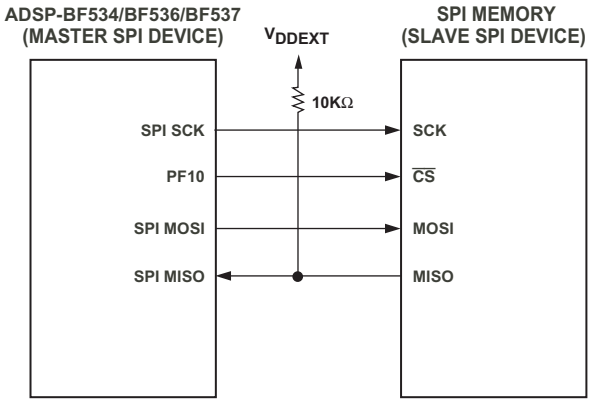


Figure 19-17. Blackfin - SPI Memory Pin-to-Pin Connections

A pull-up resistor on MISO is required for this boot mode to work properly. For this reason, the Blackfin processor reads a 0xFF on the MISO pin if the SPI memory is not responding (that is, no data written on the MISO pin by the SPI memory). This enables the boot kernel to automatically determine the type of SPI memory connected prior to the boot procedure.

Although the pull-up resistor on the MISO line is mandatory, additional pull-up resistors might also be worthwhile as well—pull up the chip select signal on PF10 to ensure the SPI memory is not activated while the Blackfin processor is in reset. Also, experience has shown that a pull-down resistor on the SCK line results in nicer plots on the oscilloscope in case of debugging the boot process.

The SPI memories supported by this interface are standard 8-,16-, and 24-bit addressable SPI memories (read sequence explained below) and these Atmel SPI DataFlash devices: AT45DB041B, AT45DB081B, AT45DB161B, AT45DB321, AT45DB642 or AT45DB1282.

Standard 8-, 16-, and 24-bit addressable SPI memories are memories that take in a read command byte of 0x03 followed by one address byte (for 8-bit addressable SPI memories), two address bytes (for 16-bit addressable SPI memories), or three address bytes (for 24-bit addressable SPI memories). After the correct read command and address are sent, the data stored in the memory at the selected address is shifted out on the `MISO` pin. Data is sent out sequentially from that address with continuing clock pulses. Analog Devices has tested these standard SPI memory devices:

- 8-bit addressable SPI memory: 25LC040 from Microchip
- 16-bit addressable SPI memory: 25LC640 from Microchip
- 24-bit addressable SPI memory: M25P80 from STMicroelectronics
- 24-bit addressable SPI dataflash: AT45DB321 from Atmel

All these devices are compatible with products from different vendors.

SPI Memory Detection Routine

Since `BMODE = 011` supports booting from various SPI memories, the on-chip boot ROM detects what type of memory is connected. To determine the type of memory (8-, 16-, or 24-bit addressable) connected to the

processor, the on-chip boot ROM sends the following sequence of bytes to the SPI memory until the memory responds back. The SPI memory does not respond back until it is properly addressed. The on-chip boot ROM:

1. Sends the read command, 0x03, on the `MOSI` pin then does a dummy read of the `MISO` pin.
2. Sends an address byte, 0x00, on the `MOSI` pin then does a dummy read of the `MISO` pin.
3. Sends another byte, 0x00, on the `MOSI` pin and checks whether the incoming byte on the `MISO` pin is anything other than 0xFF (value from the pull-up resistor). An incoming byte that is not 0xFF means that the SPI memory has responded back after one address byte and an 8-bit addressable SPI memory device is assumed to be connected.
4. If the incoming byte is 0xFF, the on-chip boot ROM sends another byte, 0x00, on the `MOSI` pin and checks whether the incoming byte on the `MISO` pin is anything other than 0xFF. An incoming byte other than 0xFF means that the SPI memory has responded back after two address bytes and a 16-bit addressable SPI memory device is assumed to be connected.
5. If the incoming byte is 0xFF, the on-chip boot ROM sends another byte, 0x00, on the `MOSI` pin and checks whether the incoming byte on the `MISO` pin is anything other than 0xFF. An incoming byte other than 0xFF means that the SPI memory has responded back after three address bytes and a 24-bit addressable SPI memory device is assumed to be connected.
6. If an incoming byte is 0xFF (meaning no devices have responded back), the on-chip boot ROM assumes that one of these Atmel DataFlash devices is connected: AT45DB041B, AT45DB081B, AT45DB161B, AT45DB321, AT45DB642 or AT45DB1282. These DataFlash devices have a different read sequence than the one described above for standard SPI memories. For more

information, refer to the data sheet for the device. The on-chip Boot ROM determines which of the above Atmel DataFlash memories is connected by reading the status register.

The SPI baud rate register is set to 133, which, when based on a 54 MHz system clock, results in a 54 MHz/(2 x 133) = 203 kbit/s bit rate.

Figure 19-18 through Figure 19-21 show the boot sequence for an SPI master mode boot using a 24-bit addressable SPI memory (25LC640 from Microchip). The loader file used is the same as shown in Figure 19-12 on page 19-38.

Initially, the on-chip boot ROM determines the SPI memory type connected–an 8-, 16-, or 24-bit addressable or an Atmel DataFlash.

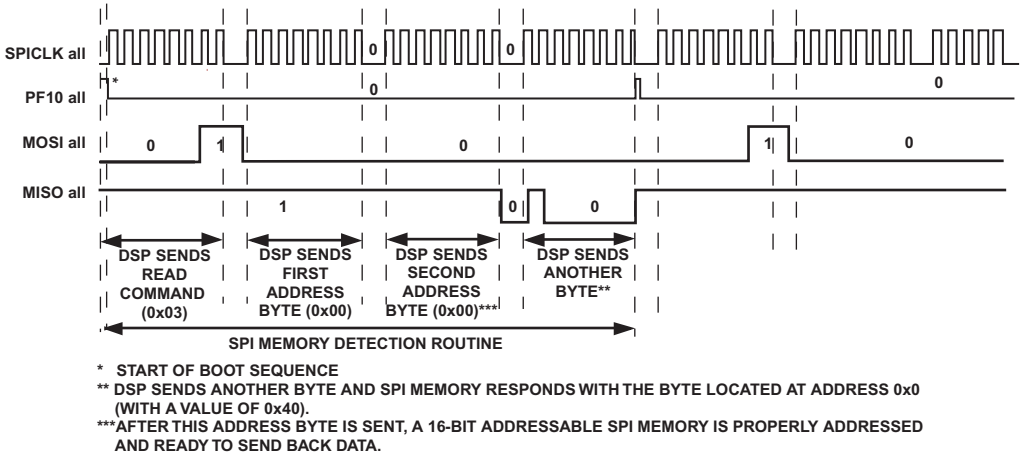


Figure 19-18. SPI Master Mode Boot Sequence: SPI Memory Detection Sequence

The on-chip boot ROM has detected that a 16-bit addressable SPI memory is connected at this point. Next, it issues the read command and sends out address 0x0000 to read in the first 10-byte header for the INIT code DXE count block.

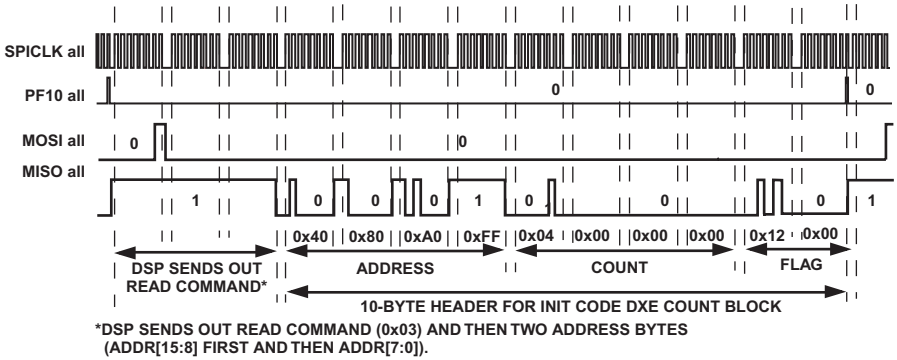


Figure 19-19. SPI Master Mode Boot Sequence: Boot 10-Byte Header for INIT Code DXE Count Block

Since the INIT code DXE count block is a 4-byte IGNORE block, the on-chip boot ROM then issues the read command and sends out address 0x000E for the 10-byte header for block 1 of the INIT code DXE. After this header is read in, the on-chip boot ROM knows where block 1 resides in memory and how many bytes to boot into that location.

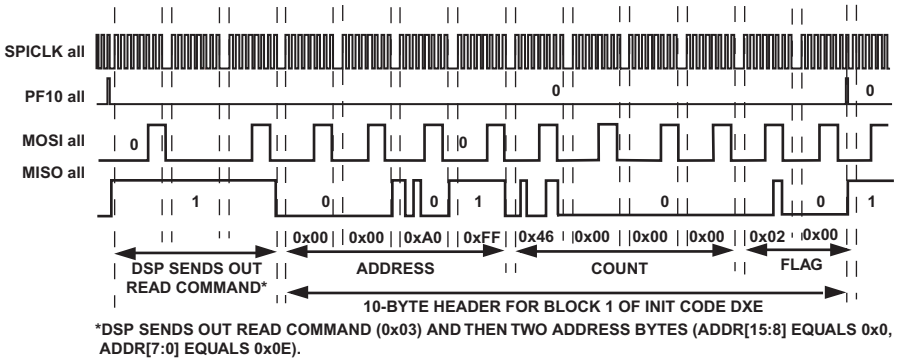


Figure 19-20. SPI Master Mode Boot Sequence: Boot 10-Byte Header for Block 1 of INIT Code DXE

Once this information is processed, the on-chip boot ROM again issues a read command and sends out address 0x0018 to boot in block 1 of the INIT code DXE.

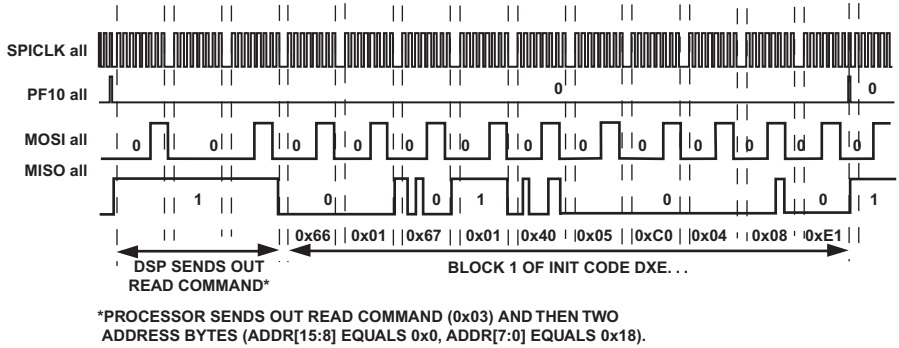


Figure 19-21. SPI Master Mode Boot Sequence: Boot Block 1 of INIT Code DXE

SPI Slave Mode Boot From SPI Host (BMODE = 100)

For SPI slave mode boot (BMODE = 100), the Blackfin processor is configured as an SPI slave device and a host is used to boot the processor. The hardware configuration shown in Figure 19-22 is assumed.

The host drives the SPI clock and is therefore responsible for the timing. The host must provide an active-low chip select signal that connects to the $\overline{\text{SPISS}}$ input of the Blackfin processor on pin PF14. It can toggle with each byte transferred or remain low during the entire procedure. In SPI slave boot mode, the boot kernel sets the CPHA bit and clears the CPOL bit in the SPI_CTL register. Therefore, the MISO pin is latched on the falling edge of the MOSI pin. See "SPI Transfer Protocols" on page 10-14 for details. Eight-bit data is expected; 16-bit mode is not supported.

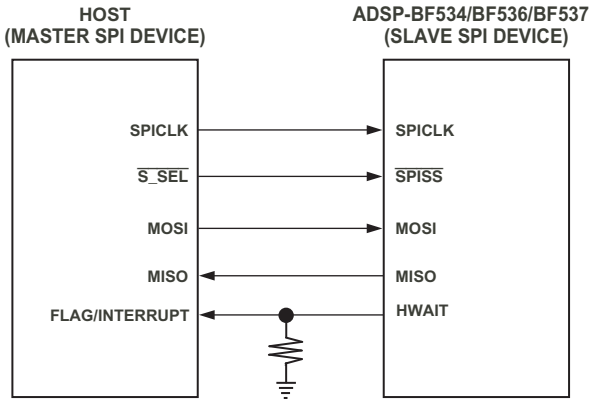


Figure 19-22. Connections Between Host (SPI Master) and Blackfin Processor (SPI Slave)

In SPI slave mode the HWAIT functionality must be activated. The HWAIT handshake signal can operate on any GPIO pin of port F, G or H. The resistor shown in Figure 19-15 on page 19-41 programs HWAIT to hold off the host when high. See "Host Wait Feedback Strobe (HWAIT)" on page 19-19 for further details. The SPI module does not provide extremely large receive FIFOs, so the host is requested to test the HWAIT signal at every byte.

Below are timing diagrams of an SPI slave mode boot using an ADSP-BF536 processor as the host and an ADSP-BF537 processor as the slave SPI device. On the host side, PF4 is used as the $\overline{\text{CS}}$ which is connected to the $\overline{\text{SPISS}}$ of the slave ADSP-BF537. The SPI slave's PG0 (ADSP-BF537 processor) functions as HWAIT and connects to PG1 of the host (ADSP-BF536 processor). All the timing diagrams are from the SPI slave point of view.

The loader file used (SPI_Slave_HostFile.ldr) is the same one as in Figure 19-12 on page 19-38, except two more blocks are added to show the functionality of this boot mode—a ZEROFILL block going to location 0xFFA0 0300 with a byte count of 0x4000 and a data block going to

location 0xFFA0 4300 with these values: 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, 0x99, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF, and 0x19.

After the SPI slave receives the first 10-byte header from the host, it knows which PG flag to configure as the HWAIT feedback strobe. In this case, PG0 is used. Note the deassertion of HWAIT in Figure 19-23 after bits 8–5 of the FLAG word are processed.

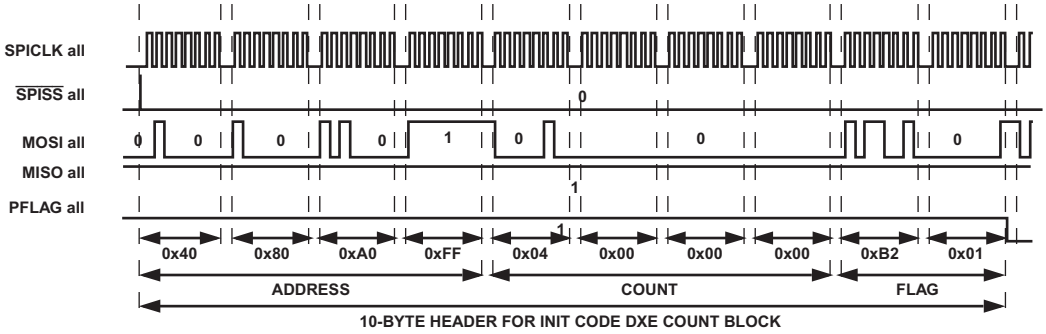


Figure 19-23. SPI Slave Mode Boot Sequence: Start of Boot Sequence

After that, the host sends out the 4-byte INIT code DXE count block, the 10-byte header for block 1 of the INIT code DXE, and then block 1 itself. See Figure 19-24.

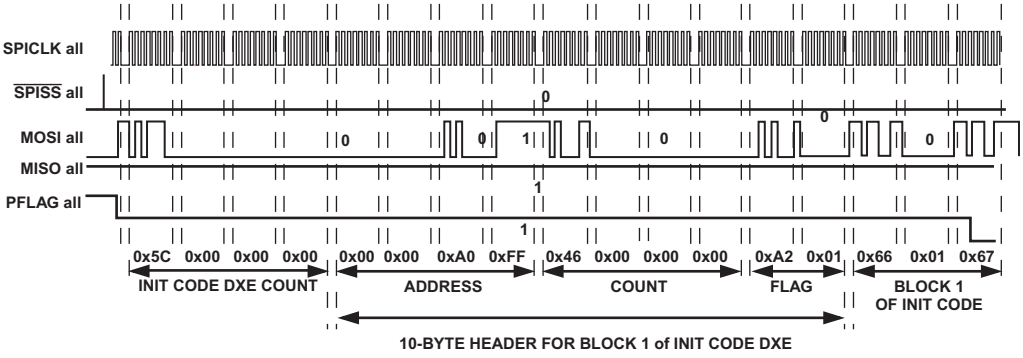


Figure 19-24. SPI Slave Mode Boot Sequence: Boot Block 1 of INIT Code DXE

After the full INIT code DXE is booted into Blackfin memory, the slave SPI Blackfin processor then asserts the feedback strobe, HWAIT, to indicate to the host not to send any more bytes during INIT code execution. Since the Blackfin processor core is running much faster than the SPI interface, the INIT code executes at a much faster rate compared to the rate at which bytes are sent from the host. See Figure 19-25.

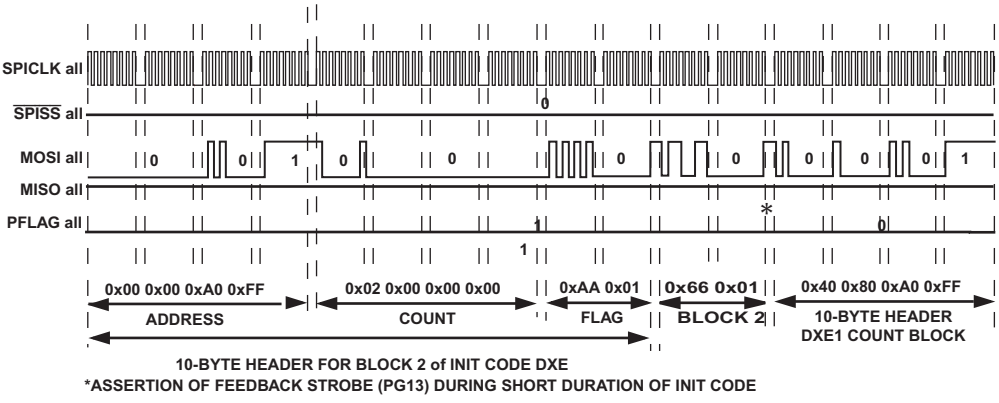
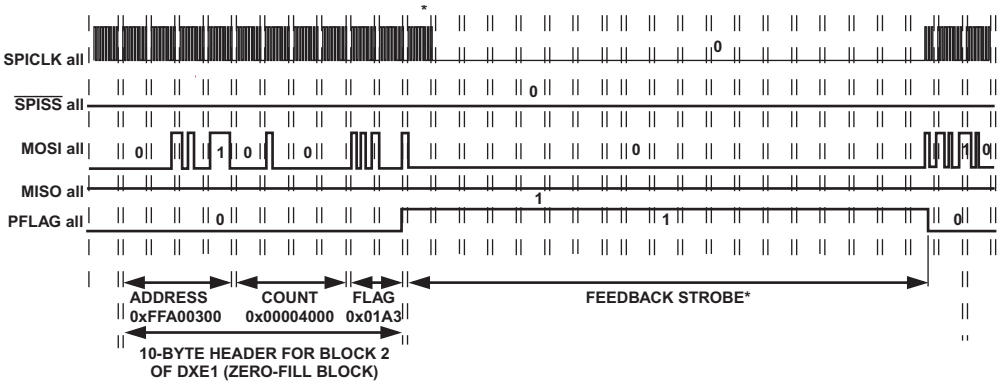


Figure 19-25. SPI Slave Mode Boot Sequence: Boot Block 2 of INIT Code DXE

Figure 19-26 shows the processing of a ZEROFILL block for this boot mode. When the on-chip boot ROM encounters a ZEROFILL block, it asserts the feedback strobe, HWAIT, to hold off the host from sending any more bytes. During this time, it fills 0x4000 zeros to locations 0xFFA0 0300 – 0xFFA0 4300 via MDMA. When complete, the on-chip boot ROM deasserts the feedback strobe and the host continues to send the remaining bytes of the boot process (10-byte header for block 3 and block 3 itself), as shown in Figure 19-27.



* NOTE THAT THE NEXT BYTE OF DATA WILL BE SENT BY THE HOST. THE DATA WILL SIT IN THE FIFO UNTIL THE FEEDBACK STROBE IS DEASSERTED. IN THIS CASE, IT IS THE FIRST BYTE (0x00) OF THE 10-BYTE HEADER FOR BLOCK 3 OF DXE1.

**ASSERTION OF FEEDBACK STROBE (PFLAG) DURING ZERO-FILL BLOCK PROCESSING. PROCESSOR IS ZERO-FILLING MEMORY VIA MemDMA DURING THIS TIME.

Figure 19-26. SPI Slave Mode Boot Sequence: Boot ZEROFILL Block (Block 2 of DXE1)

Figure 19-27. SPI Slave Mode Boot Sequence: Boot Block 3 of DXE1 (Last Block)

TWI Master Boot Mode (BMODE = 101)

The Blackfin processor selects the slave EEPROM with the unique id 0xA0, and submits successive read commands to the device starting at two byte internal address 0x0000 and begins clocking data into the processor. The serial EEPROM must be two-byte addressable. Please note the EEPROM's device select bits `A2–A0` must be 0s (tied low). The I^2C 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 19-28.

The TWI controller is programmed so as to generate a 30% duty cycle clock in accordance with the I^2C clock specification for fast-mode operation.

In both TWI master and slave boot modes, the upper 256 bytes starting at address 0xFF90 3F00 must not be used. 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.

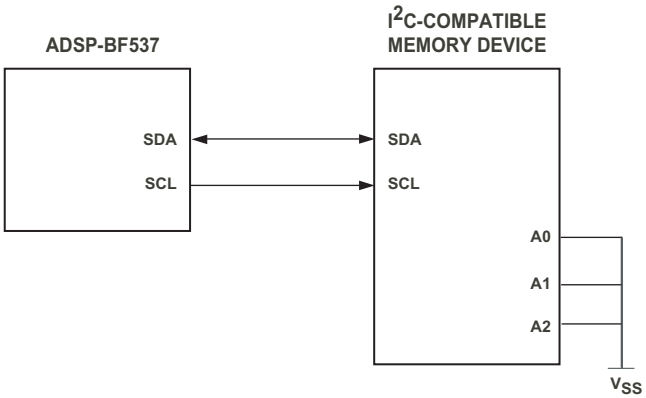


Figure 19-28. TWI Master Boot Mode

In Figure 19-29, The BF534/6/7 TWI controller outputs on the bus the address of the I^2C device to boot from: 0xA0 where the least significant bit indicates the direction of the transfer. In this case it is a write (0) in order to write the first 2 bytes of the internal address from which to start booting (0x00). Figure 19-30 shows the TWI init and zero fill blocks.

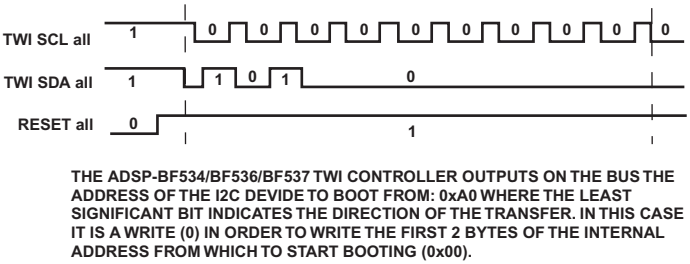


Figure 19-29. TWI Master Booting

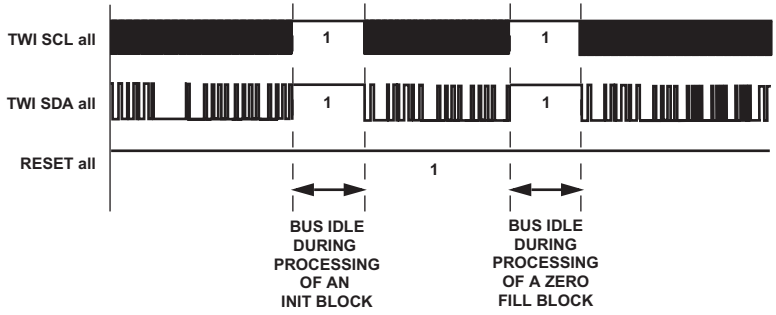


Figure 19-30. TWI Init and Zero Fill Blocks

TWI Slave Boot Mode (BMODE = 110)

The I^{2}C host agent selects the slave (Blackfin processor) with the 7-bit slave address of 0x5F. The processor replies with an acknowledgement and the host can then download the boot stream. The I^{2}C host agent should comply with *Philips I2C Bus Specification version 2.1*.
The host device supplies the serial clock. See Figure 19-31 and Figure 19-32.

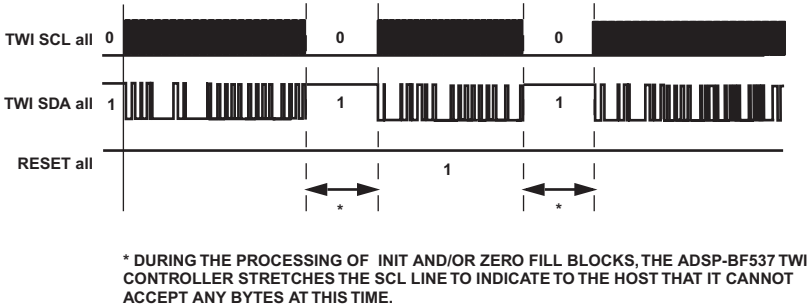


Figure 19-31. TWI Slave Booting

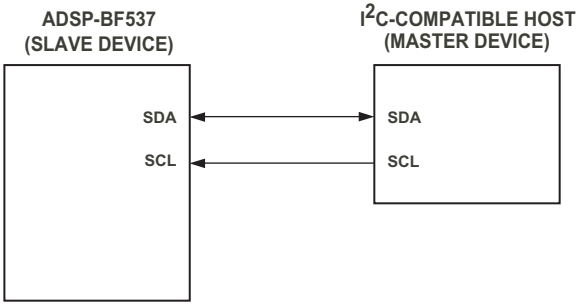


Figure 19-32. TWI Slave Boot Mode

On the Blackfin processor, in both TWI master and slave boot modes, the upper 256 bytes of data bank A starting at address 0xFF90 3F00 must not be used. 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.

UART Slave Mode Boot via Master Host (BMODE = 111)

UART booting on the Blackfin processor is supported only through UART0, and the Blackfin processor is always a slave.

Using an autobaud detection sequence, a boot-stream-formatted program is downloaded by the host. The host agent selects a bit rate within the UART's clocking capabilities. When performing the autobaud, the UART expects an "@" character (0x40, eight bits data, one start bit, one stop bit, no parity bit) on the UART0 `RXD` input to determine the bit rate. The hardware support and the mathematical operations to perform for this autobaud detection is explained in "Autobaud Mode" on page 15-34 of Chapter 15, "General-Purpose Timers". The boot kernel then replies with an acknowledgement, and the host can then download the boot stream. The acknowledgement consists of the following four bytes: `0xBF`, `UART0_DLL`, `UART0_DLH`, `0x00`. The host is requested to not send further

bytes until it has received the complete acknowledge string. Once the 0x00 byte has been received, the host can send the entire boot stream at once. 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.

When the boot kernel is processing ZEROFILL or INIT 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 which can operate on any GPIO of port G. The GPIO which is actually used is encoded in the FLAG word of all block headers. See Figure 19-6 on page 19-14 for details. The boot kernel is not permitted to drive any of the GPIOs before the first block header has been received and evaluated completely. Therefore, a pulling resistor on the HWAIT signal is recommended. If the resistor pulls down to ground, the host is always permitted to send when the HWAIT signal is low and must pause transmission when HWAIT is high. The Blackfin UART module does not provide extremely large receive FIFOs, so the host is requested to test HWAIT at every transmitted byte.

As indicated in Figure 19-33, the HWAIT feedback may connect to the Clear-To-Send (CTS) input of an EIA-232E compatible host device, resulting in a subset of a so-called hardware handshake protocol. At boot time the Blackfin does not evaluate any Request-To-Send (RTS) signal driven by the host.

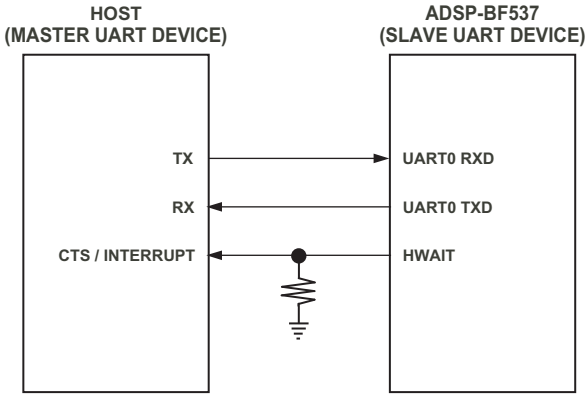


Figure 19-33. UART Slave Boot Mode

Figure 19-33 shows the logical interconnection between the UART host and the Blackfin device as 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 19-34 and Figure 19-35 provide more information about UART booting.

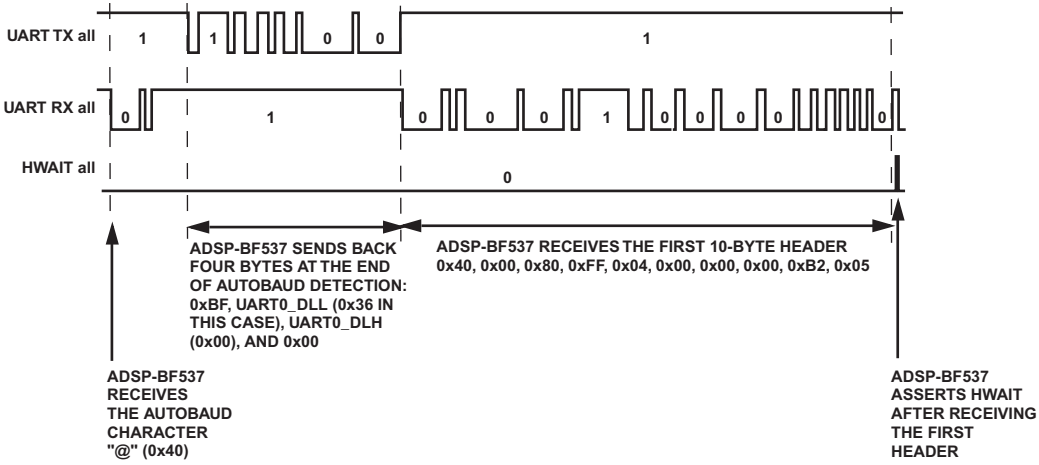


Figure 19-34. UART Slave Booting

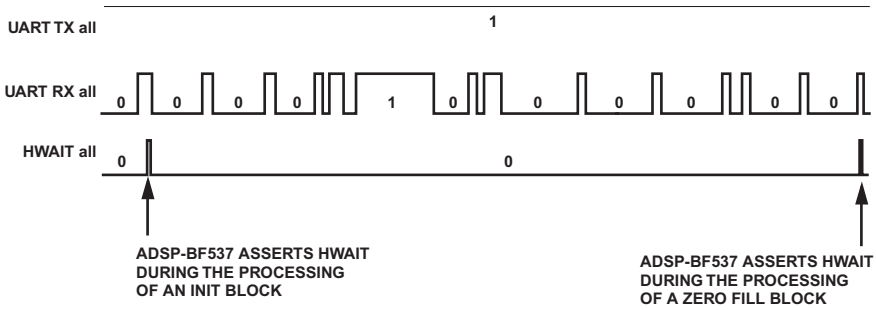


Figure 19-35. UART Init and Zero Fill Blocks

Blackfin Loader File Viewer

The Blackfin Loader File Viewer (LdrViewer) available from http://www.blackfin.org/tools) is a very useful utility that takes a loader file as an input and breaks it down and categorizes it into individual .DXE files and displays it as individual blocks with headers (ADDRESS, COUNT, and FLAG). This handy utility can help to view a loader file's content. When the file in Figure 19-12 is loaded into the LdrViewer, the GUI contents look like Figure 19-36.

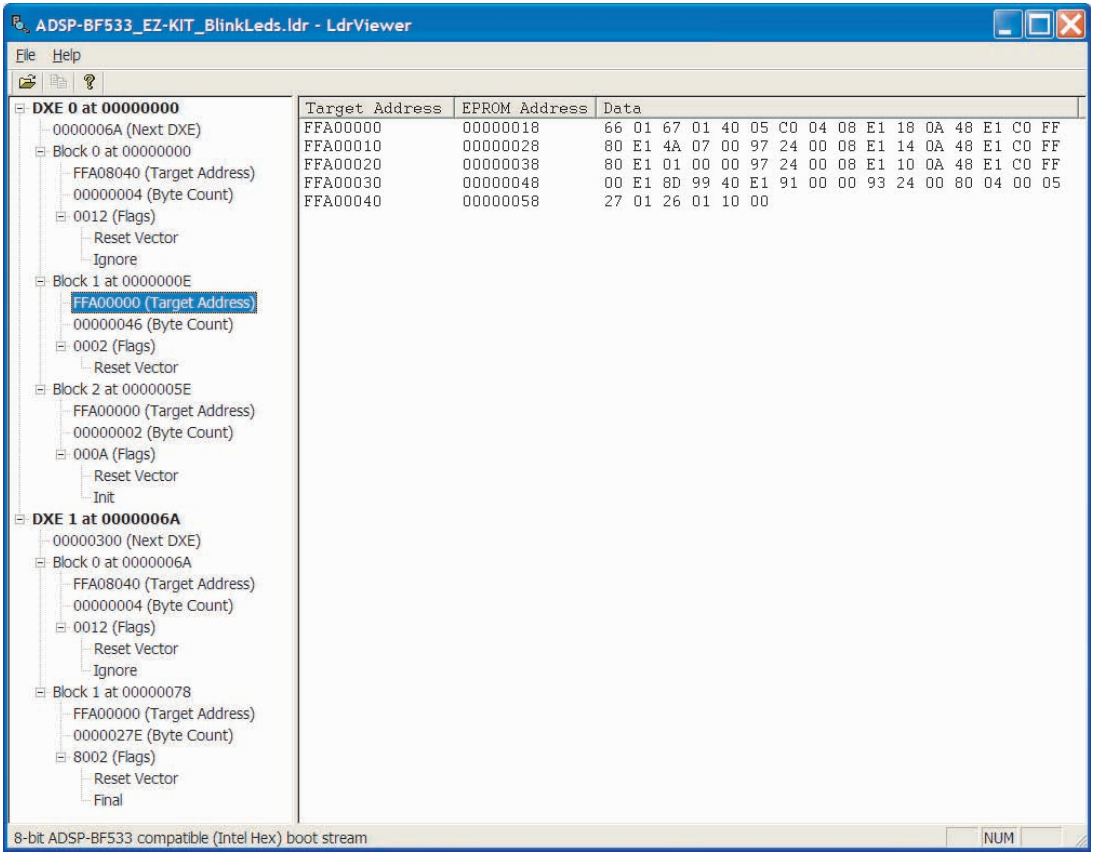


Figure 19-36. Blackfin Loader File Viewer Utility

Note that the LdrViewer utility is a 3rd-party freeware product and is not part of the standard VisualDSP++ software toolset.

20 DYNAMIC POWER MANAGEMENT

This chapter describes the dynamic power management functionality of the processor. This functionality includes:

- Phase Locked Loop (PLL) and clock control
- Dynamic power management controller
    - Operating modes
    - Voltage control

Following a description of the above functionality are consolidated register definitions and programming examples.

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 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).

A 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, 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 be changed 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 20-7.

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 20-1 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.

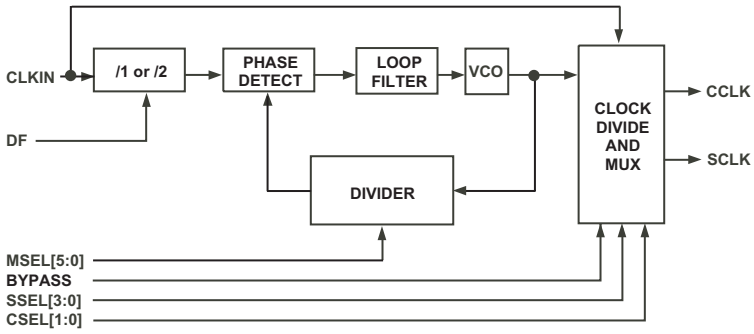


Figure 20-1. PLL Block Diagram

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_CTL Register" on page 20-26.

The divide frequency (DF) bit and multiplier select (MSEL[5:0]) field configure the various PLL clock dividers:

- DF enables the input divider
- MSEL[5:0] controls the feedback dividers

The reset value of MSEL is 0xA. This value can be reprogrammed at startup in the boot code.

Table 20-1 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 20-1. 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 20-5 on page 20-26). Note that changes to the PLL_CTL register do not take effect immediately. In general, the PLL_CTL register is first programmed with a new value, and then a specific PLL programming sequence must be executed to implement the changes. See "PLL Programming Sequence" on page 20-15.

Core Clock/System Clock Ratio Control

Table 20-2 describes the programmable relationship between the VCO frequency and the core clock. Table 20-3 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_DIV Register" on page 20-25.

The reset value of CSEL[1:0] is 0x0, and the reset value of SSEL[3:0] is 0x5. These values can be reprogrammed at startup by the boot code.

By writing the appropriate value to PLL_DIV, you can change the CSEL and SSEL value dynamically. Note the divider ratio of the core clock can never be greater than the divider ratio of the system clock. If the PLL_DIV register is programmed to illegal values, the SCLK divider is automatically increased to be greater than or equal to the core clock divider.

Unlike writing the PLL_CTL register, the PLL_DIV register can be programmed at any time to change the CCLK and SCLK divide values without entering the idle state.

Table 20-2. 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
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 20-3. 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 via writing a new SSEL value into PLL_DIV, take care that the enabled peripherals do not suffer data loss due to SCLK frequency changes.

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 20-8. For further information about the PLL programming sequence, see "PLL Programming Sequence" on page 20-15.

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 20-8.
- 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 Vdd (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 20-19.

Operating Modes

The processor works in four operating modes, each with unique performance and power saving benefits. Table 20-4 summarizes the operational characteristics of each mode.

Table 20-4. Operational Characteristics

Operating Mode	Power Savings	PLL Status	PLL Bypassed	CCLK	SCLK	Allowed DMA Access
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_STAT Register" on page 20-26). 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 (See the *ADSP-BF53x/BF56x Blackfin Processor Programming Reference* for details).

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 20-2 on page 20-12.

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.

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 wakeup 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.

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. For more information about hardware reset, see the *ADSP-BF53x/BF56x Blackfin Processor Programming Reference*. 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_IMASK, 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 20-5.

When in deep sleep mode, clocking to the SDRAM is turned off. Before entering deep sleep mode, software should ensure that important information in SDRAM is saved to a non-volatile memory and/or the SDRAM is placed into self-refresh mode.

Table 20-5. PLL_CTL Values after RTC Wakeup 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 20-2. 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 20-22.

Operating Mode Transitions

Figure 20-2 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 (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 20-14.

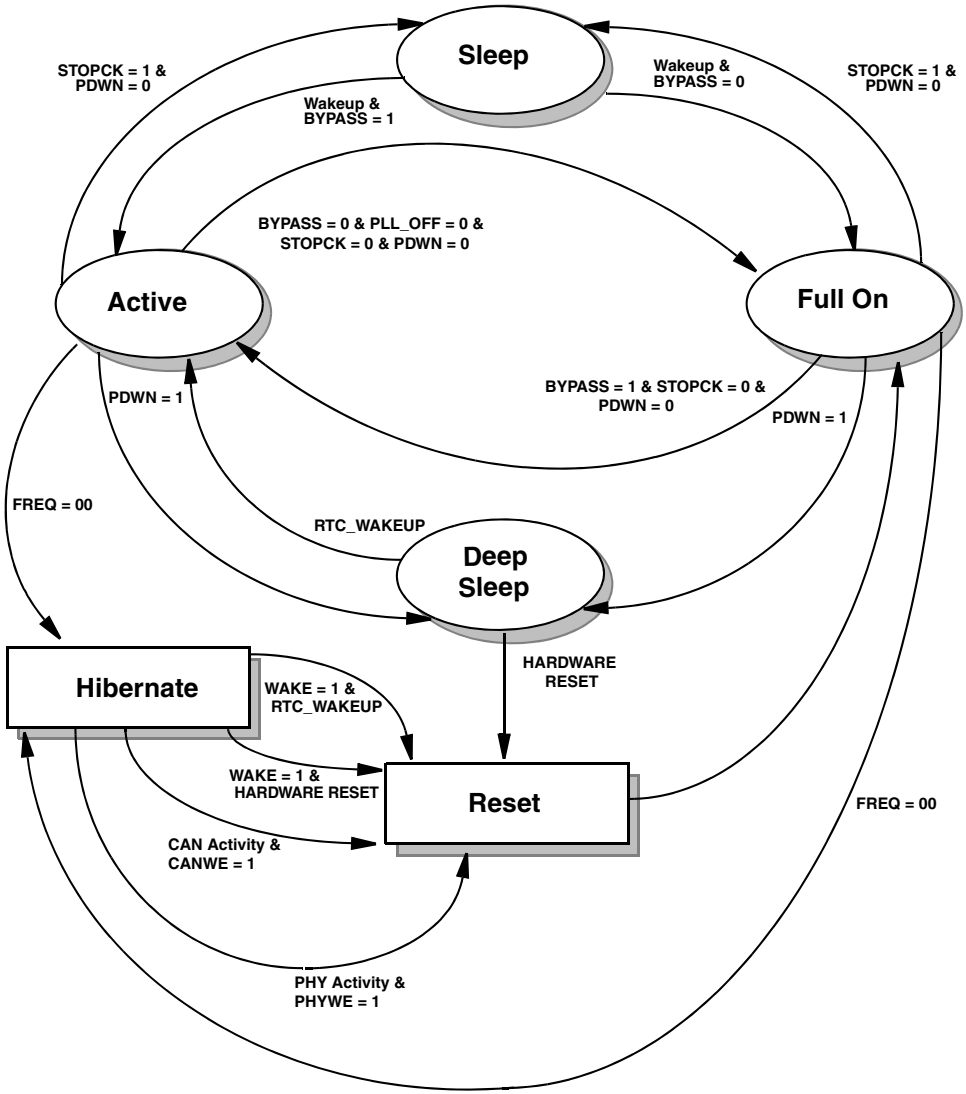


Figure 20-2. Operating Mode Transitions

In addition to the mode transitions shown in Figure 20-2, the PLL can be modified while in active operating mode. Power to the PLL can be applied and removed, and new clock-in to VCO clock (CLKIN to VCO) multiplier ratios can be programmed. Described in detail below, these changes to the PLL do not take effect immediately. As with operating mode transitions, the PLL programming sequence must be executed for these changes to take effect (see "PLL Programming Sequence" on page 20-15).

- 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.

- New multiplier ratio in active mode: New clock-in to VCO clock (CLKIN to VCO) multiplier ratios can be programmed while in active mode.

  Although the CLKIN to VCO multiplier changes are not realized in active mode, forcing the PLL to lock to the new ratio in active mode before transitioning to full on mode reduces the transition time because the PLL is already locked to the new ratio. Note that the PLL must be powered up to lock to the new ratio. To program a new CLKIN to VCO multiplier, write the new MSEL[5:0] and/or DF values to the PLL_CTL register; then execute the PLL programming sequence.

- New multiplier ratio in full on mode: The multiplier ratio can also be changed while in full on mode.

  In this case, the PLL state automatically transitions to active mode while the PLL is locking. After locking, the PLL returns to full on 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 (see information on page 20-15).

Table 20-6 summarizes the allowed operating mode transitions.

Attempting to cause mode transitions other than those shown in Table 20-6 causes unpredictable behavior.

Table 20-6. Allowed Operating Mode Transitions

New Mode	Current Mode			
	Full On	Active	Sleep	Deep Sleep
Full On	–	Allowed	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, which is shown in Listing 20-1 and Listing 20-2. This code sequence first brings the processor to a known, idled state. Once in this idled state, the PLL recognizes

and implements the changes made to the PLL_CTL register. After the changes take effect, the processor operates with the new settings, including the new operating mode, if one is programmed.

PLL Programming Sequence

If new values are assigned to MSEL or DF in the PLL control register (PLL_CTL), the instruction sequence shown in Listing 20-1 and Listing 20-2 puts those changes into effect. The PLL programming sequence is also executed when transitioning between operating modes.

Changes to the divider-ratio bits, CSEL and SSEL, can be made dynamically; they do not require execution of the PLL programming sequence.

Listing 20-1. PLL Programming Sequence (ASM)

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

The first two instructions in the sequence take the core to an idled state with interrupts disabled; the interrupt mask (IMASK) is saved to the R0 register, and the instruction pipeline is halted. The PLL state machine then loads the PLL_CTL register changes into the PLL. In order to break from the idled state, the PLL wakeup event must be enabled in the system interrupt controller interrupt wakeup register (set bit 0 of SIC_IWR).

Listing 20-2. PLL Programming Sequence (C)

```
#include <ccblkfn.h>
int temp;
temp = cli();
idle();
sti(temp);
```

This sequence behaves the same way as the ASM sequence in Listing 20-1. However, the interrupt mask (IMASK) is saved to the data element temp rather than to the R0 register.

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 wakeup interrupt.

Depending on how the PLL_CTL register is programmed, the processor proceeds in one of the following four ways:

- If the PLL_CTL register is programmed to enter either active or full on operating mode, the PLL generates a wakeup signal, and then the processor continues with the STI instruction in the sequence, as described in "PLL Programming Sequence" on page 20-15.

  When the state change enters full on mode from active mode or active from full on, the PLL itself generates a wakeup signal that can be used to exit the idled core state. The wakeup signal is generated by the PLL itself or another peripheral, watchdog or other timer, RTC, or other source. For more information about events that cause the processor to wakeup from being idled, see the "Program Sequencer" chapter of the *ADSP-BF53x/BF56x Blackfin Processor Programming Reference*.

- 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 wakeup signal before continuing.

When the wakeup signal has been asserted, the instruction sequence continues with the STI instruction, as described in "PLL Programming Sequence" on page 20-15, causing the processor to transition to:

—Active mode if BYPASS in the PLL_CTL register is set

—Full on mode if the BYPASS bit is cleared

- If the PLL_CTL register is programmed to enter deep sleep operating mode, the processor immediately transitions to deep sleep mode and waits for an RTC interrupt or hardware reset signal:

  —An RTC interrupt causes the processor to enter active operating mode and continue with the STI instruction in the sequence, as described below.

  —A hardware reset causes the processor to execute the reset sequence. For more information about hardware reset, see the *ADSP-BF53x/BF56x Blackfin Processor Programming Reference*.

- If no operating mode transition is programmed, the PLL generates a wakeup signal, and the processor continues with the STI instruction in the sequence, as described in the following section.

PLL Programming Sequence Continues

The instruction sequence shown in Listing 20-1 and Listing 20-2 then continues with the STI instruction. Interrupts are re-enabled, IMASK is restored, and normal program flow resumes.

To prevent spurious activity, DMA should be suspended while executing this instruction sequence.

Dynamic Supply Voltage Control

In addition to clock frequency control, the processor provides the capability to run the core processor at different voltage levels. As power dissipation is proportional to the voltage squared, significant power reductions can be accomplished when lower voltages are used.

The processor uses three power domains. These power domains are shown in Table 20-7. 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.

Table 20-7. Power Domains

Power Domain	V_{DD} Range
All internal logic except RTC	Variable
Real-Time Clock I/O and internal logic	Variable
All other I/O	Variable

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 20-3. This voltage level can be reduced to save power, depending upon the needs of the system.

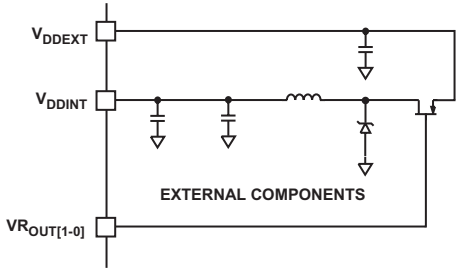


Figure 20-3. Processor Voltage Regulator

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.

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 20-8 on page 20-27). The state of the VR_CTL register is maintained during power down modes and hibernate. It is only set to its reset value by a powerup reset sequence. Writing to VR_CTL initiates a PLL relock sequence, thus, the PLL reprogramming sequence must be followed after modifying VR_CTL.

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 bit controls how quickly the voltage output settles on its final value. In general, higher gain allows for quicker settling times but causes more overshoot in the process.

Table 20-8 lists the gain levels configured by GAIN[1:0].

Table 20-8. GAIN Encodings

GAIN	Value
00	5
01	10
10	20
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).

Table 20-9 lists the switching frequency values configured by FREQ[1:0].

Table 20-9. FREQ Encodings

FREQ	Value
00	Powerdown/Bypass onboard regulation
01	333 kHz
10	667 kHz
11	1MHz

To bypass onboard regulation, program a value of b#00 in the FREQ field and leave the VROUT pins floating.

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 voltage tolerances and allowed rates of change.

Reducing the processor's operating voltage to greatly conserve power or raising the operating voltage to greatly increase performance will probably require significant changes to the operating voltage level. To ensure predictable behavior when varying the operating voltage, the processor should be brought to a known and stable state before the operating voltage is modified.

The recommended procedure is to follow the PLL programming sequence when varying the voltage. 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 20-10 lists the voltage level values for `VLEV[3:0]`.

Table 20-10. VLEV Encodings

VLEV	Voltage
0000–0101	Reserved
0110	.85 volts
0111	.90 volts
1000	.95 volts
1001	1.00 volts
1010	1.05 volts
1011	1.10 volts
1100	1.15 volts
1101	1.20 volts

Table 20-10. VLEV Encodings (Cont'd)

VLEV	Voltage
1110	1.25 volts
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 has been changed to the new level, the processor can safely return to any operational mode so long as the operating parameters, such as core clock frequency (CCLK), are within the limits specified in the processor data sheet for the new operating voltage level.

Powering Down the Core (Hibernate State)

The internal supply regulator for the processor can be shut off by writing 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:

- RTC event or assertion of the $\overline{\text{RESET}}$ pin. Set the wakeup-enable (WAKE) control bit. This bit must be set if no other wakeups are enabled.
- External GP event, or PHY event (ADSP-BF536 or ADSP-BF537 only, if PHY is used). On the ADSP-BF536 and ADSP-BF537 Blackfin processors, set the PHY wakeup enable (PHYWE) control bit

to enable wakeup upon assertion of the PHY_INT/PH6 pin by an external PHY device. If no external PHY interrupt is needed, or if using the ADSP-BF534, set this bit to enable a general-purpose external event via the PH6 pin. (This is the only functionality of this bit on the ADSP-BF534.)

- Activity on the CANRX pin. Set the CAN RX wakeup enable (CANWE) control bit to enable wakeup upon detection of CAN bus activity on the CANRX pin. Please see "CAN Wakeup From Hibernate State" on page 9-38 for more details.

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 thus the internal state of the processor is not maintained, with the exception of the VR_CTL register. Therefore, any critical information stored internally (memory contents, register contents, and so on) must be written to a non-volatile storage device prior to removing power. Be sure to set the drive CKE low during reset (CKELOW) 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 CKE pin going high during reset, which takes the SDRAM out of self-refresh mode, resulting in data decay in the SDRAM due to loss of refresh rate.

Powering down V_{DDINT} does not affect V_{DDEXT}. While V_{DDEXT} is still applied to the processor, external pins are maintained at a three-state level, unless otherwise specified.

To power down the internal supply:

1. If needed, ensure that the RTC, CAN, and/or PHY are enabled for wakeup in the SIC_IWR register.
2. Write to VR_CTL, setting the FREQ bits to b#00, and the appropriate wakeup bit to 1 (CANWE, PHYWE, WAKE on the ADSP-BF536 or ADSP-BF537, and CANWE, WAKE on the ADSP-BF534). Optionally, set the CKELOW bit if SDRAM data should be maintained.
3. Execute the PLL reprogramming sequence.
4. When the idle state is reached, V_{DDINT} will transition to 0 V.
5. When the processor is woken up, whether by RTC, CAN, or PHY on the ADSP-BF536 or ADSP-BF537, or by RTC or CAN on the ADSP-BF534, or by a reset interrupt, the PLL relocks and the boot sequence defined by the BMODE[1:0] pin settings takes effect.

Failure to allow V_{DDINT} to complete the transition to 0 V before waking up the processor can cause undesired results.

PLL Registers

The user interface to the PLL is through four memory-mapped registers (MMRs):

- The PLL divide register (PLL_DIV)
- The PLL control register (PLL_CTL)
- The PLL status register (PLL_STAT)
- The PLL lock count register (PLL_LOCKCNT)

All four registers are 16-bit MMRs and must be accessed with aligned 16-bit reads/writes. These registers are shown in Figure 20-4 through Figure 20-7.

Table 20-11 shows the functions of the PLL/VR registers.

Table 20-11. PLL/VR Register Mapping

Register Name	Function	Notes
PLL_CTL	PLL control register	Requires reprogramming sequence when written
PLL_DIV	PLL divisor register	Can be written freely
PLL_STAT	PLL status register	Monitors active modes of operation
PLL_LOCKCNT	PLL lock count register	Number of SCLKs allowed for PLL to relock
VR_CTL	Voltage regulator control register	Requires PLL reprogramming sequence when written

PLL_DIV Register

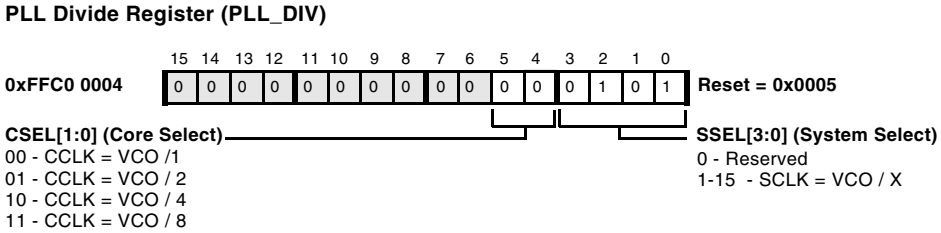


Figure 20-4. PLL Divide Register

PLL_CTL Register

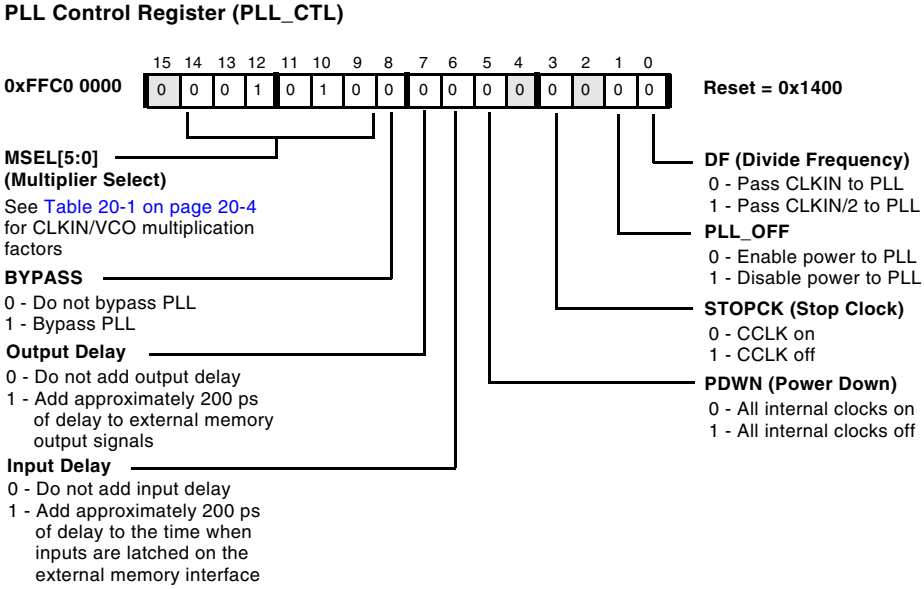


Figure 20-5. PLL Control Register

PLL_STAT Register

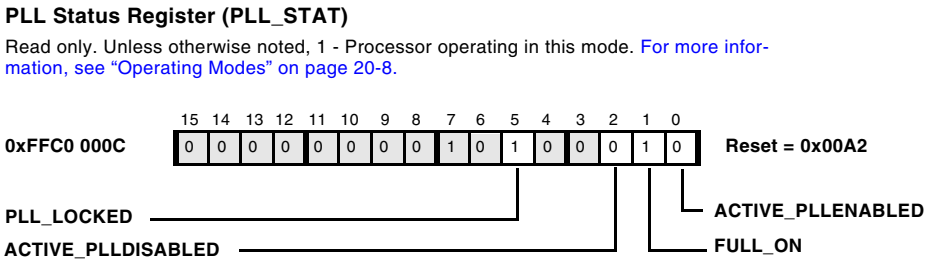


Figure 20-6. PLL Status Register

PLL_LOCKCNT Register

PLL Lock Count Register (PLL_LOCKCNT)

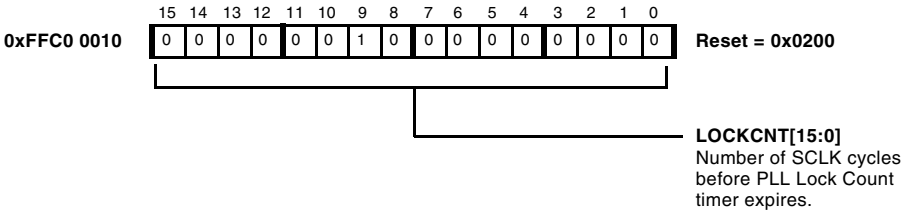


Figure 20-7. PLL Lock Count Register

VR_CTL Register

Voltage Regulator Control Register (VR_CTL)

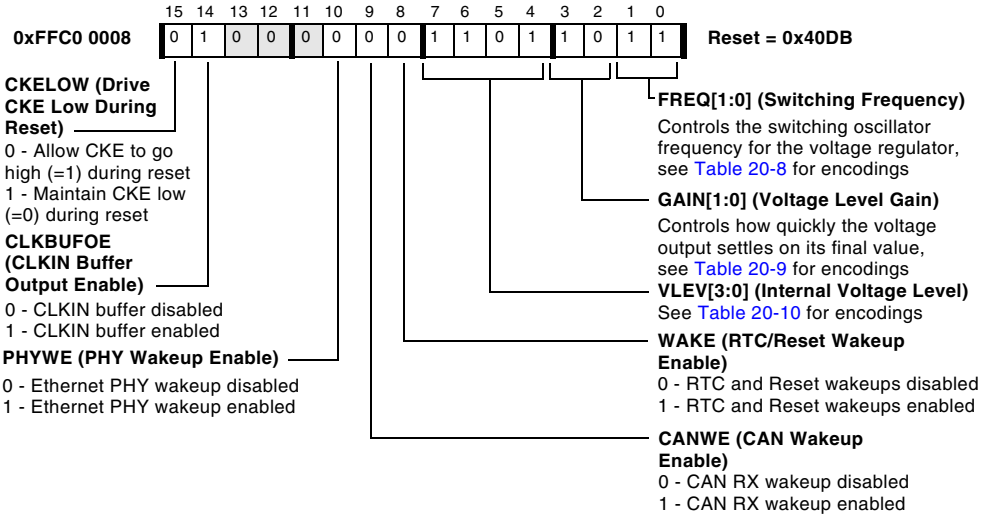


Figure 20-8. Voltage Regulator Control Register

The CLKIN buffer output enable (CLKBUFOE) control bit allows another device, most likely the Ethernet PHY, and the Blackfin processor to run from a single crystal oscillator. Clearing this bit prevents the CLKBUF pin from driving a buffered version of the input clock CLKIN.

Programming Examples

The following code examples illustrate how to use the on-chip voltage regulator and how to effect various operating mode transitions. Some setup code has been removed for clarity, and the following assumptions are made:

- For operating mode transition examples:
  - In the ASM examples, P0 points to the PLL control register (PLL_CTL). P1 points to the PLL divide register (PLL_DIV).
  - In the C examples, the appropriate headers are included and one data element is declared, as follows:

```
#include <cdefBF537.h>   /* sets up MMR access via
*pREGISTER_NAME labels */

#include <ccblkfn.h>   /* contains intrinsics for
Blackfin assembler commands */

int    IMASK_reg;   /* 32-bit data element to tempo-
rarily store IMASK */
```

  - The PLL wakeup interrupt is enabled as a wakeup signal.
  - MSEL[5:0] and DF in PLL_CTL are set to (b#011111) and (b#0) respectively, signifying a CLKIN to VCO multiplier of 31x.

- For voltage regulator examples:
  - In the ASM examples, P0 points to the voltage regulator control register (VR_CTL).
  - In the C examples, the appropriate headers are included and three data elements are declared, as follows.

```
#include <cdefBF537.h>   /* sets up MMR access via
*pREGISTER_NAME labels */

#include <ccblkfn.h>   /* contains intrinsics for
Blackfin assembler commands */

int    IMASK_reg;   /* 32-bit data element to tempo-
rarily store IMASK */

short    VR_CTL_reg;   /* 16-bit data element to tem-
porarily store VR_CTL */

short    VLEV_field;   /* 16-bit data element for
VLEV field */
```

  - The CAN, RTC/$\overline{\text{RESET}}$, and PHY/GP wakeup interrupts are enabled as wakeup signals.

Active Mode to Full On Mode

Listing 20-3 and Listing 20-4 provide code for transitioning from active operating mode to full on mode, in Blackfin assembly and C code, respectively.

Listing 20-3. Transitioning From Active Mode to Full On Mode (ASM)

```
CLI R2;   /* disable interrupts, copy IMASK to R2 */
R1.L = 0x3E00;   /* clear BYPASS bit */
W[P0] = R1;   /* and write to PLL_CTL */
```

```
IDLE;   /* drain pipeline, enter idled state, wait for PLL wakeup
*/
STI R2;   /* after PLL wakeup occurs, restore interrupts and
IMASK */
   /* processor is now in Full On mode */
```

Listing 20-4. Transitioning From Active Mode to Full On Mode (C)

```
IMASK_reg = cli( );   /* disable interrupts, copy IMASK to
IMASK_reg */
*pPLL_CTL &= ~BYPASS;   /* clear BYPASS bit and write to PLL_CTL
*/

idle( );   /* drain pipeline, enter idled state, wait for PLL
wakeup */
sti(IMASK_reg);   /* after PLL wakeup occurs, restore interrupts
and IMASK */
   /* processor is now in Full On mode */
```

Full On Mode to Active Mode

Listing 20-5 and Listing 20-6 provide code for transitioning from full on operating mode to active mode, in Blackfin assembly and C code, respectively.

Listing 20-5. Transitioning From Full On Mode to Active Mode (ASM)

```
CLI R2;   /* disable interrupts, copy IMASK to R2 */
R1.L = 0x3F00;   /* set BYPASS bit */
W[P0] = R1;   /* and write to PLL_CTL */
IDLE;   /* drain pipeline, enter idled state, wait for PLL wakeup
*/
```

```
STI R2;   /* after PLL wakeup occurs, restore interrupts and
IMASK */
   /* processor is now in Active mode */
```

Listing 20-6. Transitioning From Full On Mode to Active Mode (C)

```
IMASK_reg = cli( );   /* disable interrupts, copy IMASK to
IMASK_reg */
*pPLL_CTL | = BYPASS;   /* set BYPASS bit and write to PLL_CTL */
idle( );   /* drain pipeline, enter idled state, wait for PLL
wakeup */
sti(IMASK_reg);   /* after PLL wakeup occurs, restore interrupts
and IMASK */
   /* processor is now in Active mode */
```

In the Full On Mode, Change CLKIN to VCO Multiplier From 31x to 2x

Listing 20-7 and Listing 20-7 provide code for changing the `CLKIN` to `VCO` multiplier from 31x to 2x in full on operating mode, in Blackfin assembly and C code, respectively.

Listing 20-7. Changing CLKIN to VCO Multiplier (ASM)

```
CLI R2;   /* disable interrupts, copy IMASK to R2 */
R1.L = 0x0400;   /* change VCO multiplier to 2x */
W[P0] = R1;   /* by writing to PLL_CTL */
IDLE;   /* drain pipeline, enter idled state, wait for PLL wakeup
*/
STI R2;   /* after PLL wakeup occurs, restore interrupts and
IMASK */
   /* CLKIN to VCO multiplier is now set to 2x */
```

Listing 20-8. Changing CLKIN to VCO Multiplier (C)

```
IMASK_reg = cli( );   /* disable interrupts, copy IMASK to
IMASK_reg */
*pPLL_CTL = 0x0400;   /* change VCO multiplier to 2x by writing
to PLL_CTL */
idle( );   /* drain pipeline, enter idled state, wait for PLL
wakeup */
sti(IMASK_reg);   /* after PLL wakeup occurs, restore interrupts
and IMASK */
   /* CLKIN to VCO multiplier is now set to 2x */
```

Setting Wakeups and Entering Hibernate State

Listing 20-9 and Listing 20-10 provide code for configuring the regulator wakeups and placing the regulator in the hibernate state, in Blackfin assembly and C code, respectively.

Listing 20-9. Configuring Regulator Wakeups and Entering Hibernate State (ASM)

```
R1 = W[P0](Z);   /* read VR_CTL register */
BITSET (R1, 8);   /* enable wakeup from RTC/reset */
BITSET (R1, 15);   /* protect SDRAM contents during reset after
wakeup */
BITCLR (R1, 0);
BITCLR (R1, 1);   /* clear FREQ bits to powerdown core */
W[P0] = R1;   /* write to VR_CTL */
CLI R1;   /* disable interrupts, copy IMASK to R1 */
IDLE;   /* drain pipeline, enter idled state, wait for VR to
hibernate */
   /* Hibernate state: no code executes until wakeup triggers
reset */
```

Listing 20-10. Configuring Regulator Wakeups and Entering Hibernate State (C)

```
VR_reg = *pVR_CTL;   /* read VR_CTL into temporary data */
VR_reg |= (WAKE|CKELOW);   /* enable RTC/Reset and protect SDRAM
*/
VR_reg &= ~FREQ;   /* clear FREQ bits to powerdown core */
*pVR_CTL = VR_reg;   /* write new value to VR_CTL */
IMASK_reg = cli( );   /* disable interrupts, copy IMASK to
IMASK_reg */
idle( );   /* drain pipeline, enter idled state, wait for PLL
wakeup */
   /* Hibernate state: no code executes until wakeup triggers
reset */
```

Changing Internal Voltage Levels

Listing 20-11 and Listing 20-12 provide code for dynamically changing the internal voltage level, in Blackfin assembly and C code, respectively. 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 20-11. Changing Core Voltage via the On-Chip Regulator (ASM)

```
R1 = W[P0](Z);   /* read VR_CTL into backgnd_reg for DEPOSIT */
R7 = 0xC;   /* 0xC = 1.15V (-5% - +10%) in VLEV */
R7 <<= 16;   /* value to be deposited must be upper 16 bits */
R2 = 0x0404(Z);   /* VLEV field position = bit 4, VLEV field
length = 4 bits */
R7 = R7|R2;   /* R7.L now contains position and length for
DEPOSIT */
```

```
R2 = DEPOSIT(R1, R7);   /* R2 contains previous VR_CTL value with
new VLEV field */
W[P0] = R2;   /* write new value to VR_CTL */
CLI R1;   /* disable interrupts, copy IMASK to R1 */
IDLE;   /* drain pipeline, enter idled state, wait for PLL wakeup
*/
STI R1;   /* after PLL wakeup occurs, restore interrupts and
IMASK */
   /* internal voltage now at 1.15V (-5% - +10%) */
```

Listing 20-12. Changing Core Voltage via the On-Chip Regulator (C)

```
VLEV_field = 0xC;   /* 0xC = 1.15V (-5% - +10%) in VLEV */
VR_reg = *pVR_CTL;   /* read VR_CTL into temporary data */
VR_reg &= 0xFF0F;   /* clear out current VLEV field data */
VLEV_field <<= 4;   /* shift VLEV field into proper position */
VR_reg |= VLEV_field;   /* deposit new VLEV field into VR_CTL
word */
*pVR_CTL = VR_reg;   /* write new value to VR_CTL */
IMASK_reg = cli( );   /* disable interrupts, copy IMASK to
IMASK_reg */
idle( );   /* drain pipeline, enter idled state, wait for PLL
wakeup */
sti(IMASK_reg);   /* after PLL wakeup occurs, restore interrupts
and IMASK */
   /* internal voltage now at 1.15V (-5% - +10%) */
```

21 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, design issues cited here are discussed in detail in other sections of this manual. In such cases, a reference appears to the corresponding section of the text, instead of repeating the discussion in this chapter.

Pin Descriptions

Refer to the processor data sheet for pin information, including pin numbers for the 208-ball Pb-free sparse MBGA and 182-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 via the CSEL bit settings in the PLL_DIV register. The system clock is based on a divider ratio that is programmed via the SSEL bit settings in the PLL_DIV register. For detailed information about how to set and change CCLK and SCLK frequencies, see Chapter 20, "Dynamic Power Management".

Configuring and Servicing Interrupts

A variety of interrupts are available. They include both core and peripheral interrupts. The processor assigns default core priorities to system-level interrupts. However, these system interrupts can be remapped via the system interrupt assignment registers (SIC_IARx). For more information, see Chapter 4, "System Interrupts".

The processor core supports nested and non-nested interrupts, as well as self-nested interrupts. For explanations of the various modes of servicing events, please see the *ADSP-BF53x/BF56x Blackfin Processor Programming Reference Manual*.

Semaphores

Semaphores provide a mechanism for communication between multiple processors or processes/threads running in the same system. They are used to coordinate resource sharing. For instance, if a process is using a particular resource and another process requires that same resource, it must wait until the first process signals that it is no longer using the resource. This signalling is accomplished via semaphores.

Semaphore coherency is guaranteed by using the test and set byte (atomic) instruction (TESTSET). The TESTSET instruction performs these functions.

- Loads the half word at memory location pointed to by a P-register. The P-register must be aligned on a half-word boundary.
- Sets CC if the value is equal to zero.
- Stores the value back in its original location (but with the most significant bit (MSB) of the low byte set to 1).

The events triggered by TESTSET are atomic operations. The bus for the memory where the address is located is acquired and not relinquished until the store operation completes. In multithreaded systems, the TESTSET instruction is required to maintain semaphore consistency.

To ensure that the store operation is flushed through any store or write buffers, issue an SSYNC instruction immediately after semaphore release.

The TESTSET instruction can be used to implement binary semaphores or any other type of mutual exclusion method. The TESTSET instruction supports a system-level requirement for a multicycle bus lock mechanism.

The processor restricts use of the TESTSET instruction to the external memory region only. Use of the TESTSET instruction to address any other area of the memory map may result in unreliable behavior.

Example Code for Query Semaphore

Listing 21-1 provides an example of a query semaphore that checks the availability of a shared resource.

Listing 21-1. Query Semaphore

```
/* Query semaphore. Denotes “Busy” if its value is nonzero. Wait
until free (or reschedule thread-- see note below). P0 holds
address of semaphore. */
QUERY:
TESTSET ( P0 ) ;
IF !CC JUMP QUERY ;
/* At this point, semaphore has been granted to current thread,
and all other contending threads are postponed because semaphore
value at [P0] is nonzero. Current thread could write thread_id to
semaphore 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”.

External Memory Design Issues

This section describes design issues related to external memory.

Example Asynchronous Memory Interfaces

This section shows glueless connections to 16-bit wide SRAM. Note this interface does not require external assertion of ARDY, since the internal wait state counter is sufficient for deterministic access times of memories.

Figure 21-1 shows the interface to 8-bit SRAM or flash. Figure 21-2 shows the interface to 16-bit SRAM or flash

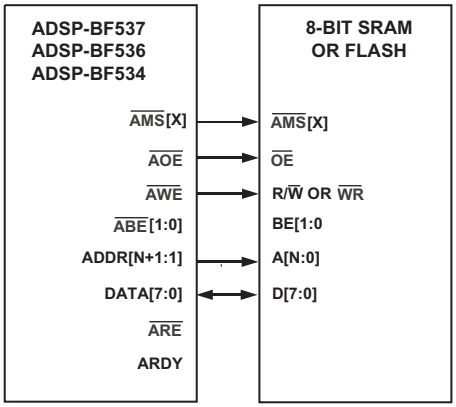


Figure 21-1. Interface to 8-Bit SRAM or Flash

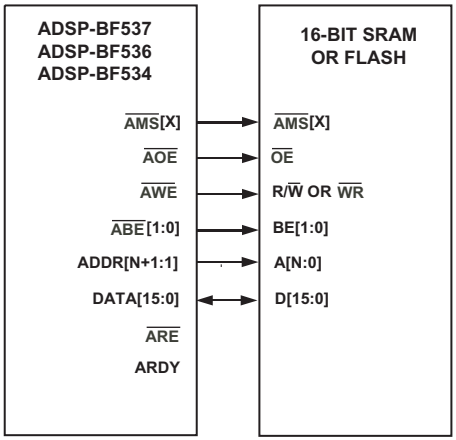


Figure 21-2. Interface to 16-Bit SRAM or Flash

Figure 21-3 shows the system interconnect required to support 16-bit memories. Note this application requires the 16-bit packing mode be enabled for this bank of memory. Otherwise, the programming model must ensure that every other 16-bit memory location is accessed starting on an even (byte address[1:0] = 00) 16-bit address.

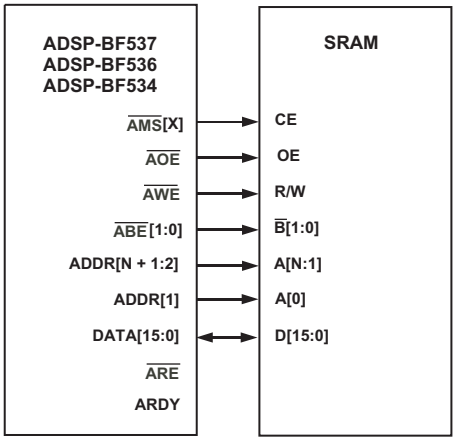


Figure 21-3. Interface to 16-Bit SRAM

Avoiding Bus Contention

Because the three-stated data bus is shared by multiple devices in a system, be careful to avoid contention. Contention causes excessive power dissipation and can lead to device failure. Contention occurs during the time one device is getting off the bus and another is getting on. If the first device is slow to three-state and the second device is quick to drive, the devices contend.

There are two cases where contention can occur. The first case is a read followed by a write to the same memory space. In this case, the data bus drivers can potentially contend with those of the memory device addressed by the read. The second case is back-to-back reads from two different memory spaces. In this case, the two memory devices addressed by the two reads can potentially contend at the transition between the two read operations.

To avoid contention, program the turnaround time (bank transition time) appropriately in the asynchronous memory bank control registers. This feature allows software to set the number of clock cycles between these types of accesses on a bank-by-bank basis. Minimally, the External Bus Interface Unit (EBIU) provides one cycle for the transition to occur.

High Frequency Design Considerations

Because the processor can operate at very fast clock frequencies, signal integrity and noise problems must be considered for circuit board design and layout. The following sections discuss these topics and suggest various techniques to use when designing and debugging signal processing systems.

Signal Integrity

In addition to reducing signal length and capacitive loading, critical signals should be treated like transmission lines.

Capacitive loading and signal length of buses can be reduced by using a buffer for devices that operate with wait states (for example, SDRAMs). This reduces the capacitance on signals tied to the zero-wait-state devices, allowing these signals to switch faster and reducing noise-producing current spikes. Extra care should be taken with certain signals such as external memory, read, write, and acknowledge strobes.

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 will be 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 21-13 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 VDDEXT and VDDINT pins of the package as shown in Figure 21-4. 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.

VDDINT 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 capacitance will raise the resonant frequency of the LC circuit. While a cluster of .1uF is acceptable below 50 MHz, a mix of .1, .01, .001uF and even 100pF 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 and larger values may be placed further from the chip.

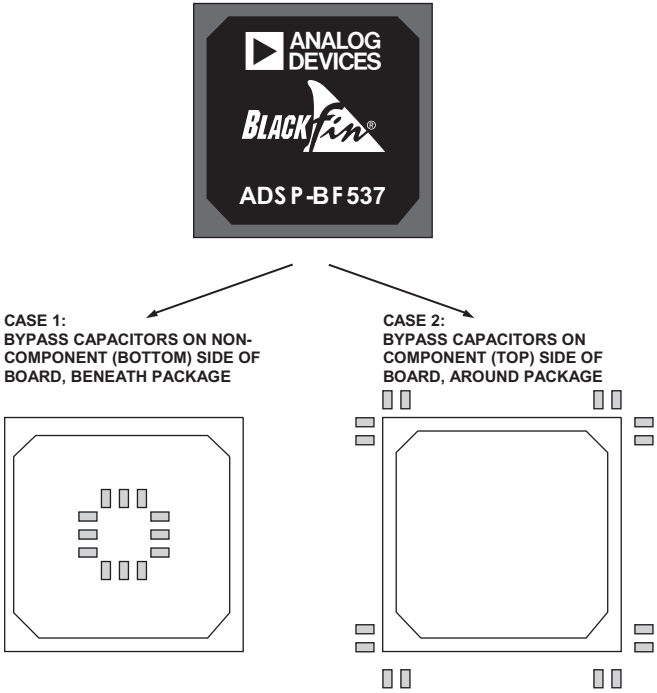


Figure 21-4. Bypass Capacitor Placement

5 Volt Tolerance

Outputs that connect to inputs on 5 V devices can float or be pulled up to 5 V. Most Blackfin pins are not 5 V tolerant. There are a few exceptions such as the TWI pins. Level shifters are required on all other Blackfin pins to keep the pin voltage at or below absolute maximum ratings.

Resetting the Processor

Our processor pins have no hysteresis and therefore require a monotonic rise and fall. Therefore even the $\overline{\text{RESET}}$ pin should not be connected directly to an R/C time delay because such a circuit would be noise sensitive.

In addition to the hardware reset mode provided via the $\overline{\text{RESET}}$ pin, the processor supports several software reset modes. For detailed information on the various modes, see the *ADSP-BF53x/BF56x Blackfin Processor Programming Reference Manual*. The processor state after reset is also described in the *ADSP-BF53x/BF56x Blackfin Processor Programming Reference Manual*.

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.

Programmable Outputs

Programmable pins used as output pins should be connected to a pullup or pulldown resistor to prevent damage to other devices during reset and before these pins are programmed as outputs.

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.

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 will ring and have overshoot and undershoot
- Simple termination will control signal integrity problems

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. Registers are listed in order by their memory-mapped address. 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.

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 Register" on page 20-26
0xFFC0 0004	PLL_DIV	"PLL Divide Register" on page 20-25
0xFFC0 0008	VR_CTL	"Voltage Regulator Control Register" on page 20-27
0xFFC0 000C	PLL_STAT	"PLL Status Register" on page 20-26
0xFFC0 0010	PLL_LOCKCNT	"PLL Lock Count Register" on page 20-27

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 0104	SYSCR	"System Reset Configuration Register" on page 19-5
0xFFC0 010C	SIC_IMASK	"System Interrupt Mask Register" on page 4-20
0xFFC0 0110	SIC_IAR0	"System Interrupt Assignment Register 0" on page 4-18

Table A-2. System Reset and Interrupt Control Registers (Cont'd)

Memory-mapped Address	Register Name	See Page
0xFFC0 0114	SIC_IAR1	"System Interrupt Assignment Register 1" on page 4-18
0xFFC0 0118	SIC_IAR2	"System Interrupt Assignment Register 2" on page 4-19
0xFFC0 011C	SIC_IAR3	"System Interrupt Assignment Register 3" on page 4-19
0xFFC0 0120	SIC_ISR	"System Interrupt Status Register" on page 4-21
0xFFC0 0124	SIC_IWR	"System Interrupt Wakeup-enable Register" on page 4-22

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 Register" on page 17-8
0xFFC0 0204	WDOG_CNT	"Watchdog Count Register" on page 17-6
0xFFC0 0208	WDOG_STAT	"Watchdog Status Register" on page 17-7

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 Register" on page 18-20
0xFFC0 0304	RTC_ICTL	"RTC Interrupt Control Register" on page 18-20
0xFFC0 0308	RTC_ISTAT	"RTC Interrupt Status Register" on page 18-21
0xFFC0 030C	RTC_SWCNT	"RTC Stopwatch Count Register" on page 18-21
0xFFC0 0310	RTC_ALARM	"RTC Alarm Register" on page 18-22
0xFFC0 0314	RTC_PREN	"Prescaler Enable Register" on page 18-22

UART0 Controller Registers

UART0 controller registers (0xFFC0 0400 – 0xFFC0 04FF) are listed in Table A-5. For UART1 registers, see Table A-16 on page A-20.

Table A-5. UART0 Controller Registers

Memory-mapped Address	Register Name	See Page
0xFFC0 0400	UART0_THR	"UART Transmit Holding Registers" on page 13-26
0xFFC0 0400	UART0_RBR	"UART Receive Buffer Registers" on page 13-26

Table A-5. UART0 Controller Registers (Cont'd)

Memory-mapped Address	Register Name	See Page
0xFFC0 0400	UART0_DLL	"UART Divisor Latch Registers" on page 13-30
0xFFC0 0404	UART0_DLH	"UART Divisor Latch Registers" on page 13-30
0xFFC0 0404	UART0_IER	"UART Interrupt Enable Registers" on page 13-28
0xFFC0 0408	UART0_IIR	"UART Interrupt Identification Registers" on page 13-29
0xFFC0 040C	UART0_LCR	"UART Line Control Registers" on page 13-21
0xFFC0 0410	UART0_MCR	"UART Modem Control Registers" on page 13-23
0xFFC0 0414	UART0_LSR	"UART Line Status Registers" on page 13-24
0xFFC0 041C	UART0_SCR	"UART Scratch Registers" on page 13-31
0xFFC0 0424	UART0_GCTL	"UART Global Control Registers" on page 13-31

SPI Controller Registers

SPI controller registers (0xFFC0 0500 – 0xFFC0 05FF) are listed in Table A-6.

Table A-6. SPI Controller Registers

Memory-mapped Address	Register Name	See Page
0xFFC0 0500	SPI_CTL	"SPI Control Register" on page 10-41
0xFFC0 0504	SPI_FLG	"SPI Flag Register" on page 10-42
0xFFC0 0508	SPI_STAT	"SPI Status Register" on page 10-43

Table A-6. SPI Controller Registers (Cont'd)

Memory-mapped Address	Register Name	See Page
0xFFC0 050C	SPI_TDBR	"SPI Transmit Data Buffer Register" on page 10-44
0xFFC0 0510	SPI_RDBR	"SPI Receive Data Buffer Register" on page 10-44
0xFFC0 0514	SPI_BAUD	"SPI Baud Rate Register" on page 10-41
0xFFC0 0518	SPI_SHADOW	"SPI RDBR Shadow Register" on page 10-44

Timer Registers

Timer registers (0xFFC0 0600 – 0xFFC0 06FF) are listed in Table A-7.

Table A-7. Timer Registers

Memory-mapped Address	Register Name	See Page
0xFFC0 0600	TIMER0_CONFIG	"Timer Configuration Registers" on page 15-44
0xFFC0 0604	TIMER0_COUNTER	"Timer Counter Registers" on page 15-46
0xFFC0 0608	TIMER0_PERIOD	"Timer Period Registers" on page 15-48
0xFFC0 060C	TIMER0_WIDTH	"Timer Width Registers" on page 15-49
0xFFC0 0610	TIMER1_CONFIG	"Timer Configuration Registers" on page 15-44
0xFFC0 0614	TIMER1_COUNTER	"Timer Counter Registers" on page 15-46
0xFFC0 0618	TIMER1_PERIOD	"Timer Period Registers" on page 15-48
0xFFC0 061C	TIMER1_WIDTH	"Timer Width Registers" on page 15-49
0xFFC0 0620	TIMER2_CONFIG	"Timer Configuration Registers" on page 15-44

Table A-7. Timer Registers (Cont'd)

Memory-mapped Address	Register Name	See Page
0xFFC0 0624	TIMER2_COUNTER	"Timer Counter Registers" on page 15-46
0xFFC0 0628	TIMER2_PERIOD	"Timer Period Registers" on page 15-48
0xFFC0 062C	TIMER2_WIDTH	"Timer Width Registers" on page 15-49
0xFFC0 0630	TIMER3_CONFIG	"Timer Configuration Registers" on page 15-44
0xFFC0 0634	TIMER3_COUNTER	"Timer Counter Registers" on page 15-46
0xFFC0 0638	TIMER3_PERIOD	"Timer Period Registers" on page 15-48
0xFFC0 063C	TIMER3_WIDTH	"Timer Width Registers" on page 15-49
0xFFC0 0640	TIMER4_CONFIG	"Timer Configuration Registers" on page 15-44
0xFFC0 0644	TIMER4_COUNTER	"Timer Counter Registers" on page 15-46
0xFFC0 0648	TIMER4_PERIOD	"Timer Period Registers" on page 15-48
0xFFC0 064C	TIMER4_WIDTH	"Timer Width Registers" on page 15-49
0xFFC0 0650	TIMER5_CONFIG	"Timer Configuration Registers" on page 15-44
0xFFC0 0654	TIMER5_COUNTER	"Timer Counter Registers" on page 15-46
0xFFC0 0658	TIMER5_PERIOD	"Timer Period Registers" on page 15-48
0xFFC0 065C	TIMER5_WIDTH	"Timer Width Registers" on page 15-49
0xFFC0 0660	TIMER6_CONFIG	"Timer Configuration Registers" on page 15-44
0xFFC0 0664	TIMER6_COUNTER	"Timer Counter Registers" on page 15-46
0xFFC0 0668	TIMER6_PERIOD	"Timer Period Registers" on page 15-48
0xFFC0 066C	TIMER6_WIDTH	"Timer Width Registers" on page 15-49
0xFFC0 0670	TIMER7_CONFIG	"Timer Configuration Registers" on page 15-44
0xFFC0 0674	TIMER7_COUNTER	"Timer Counter Registers" on page 15-46

Table A-7. Timer Registers (Cont'd)

Memory-mapped Address	Register Name	See Page
0xFFC0 0678	TIMER7_PERIOD	"Timer Period Registers" on page 15-48
0xFFC0 067C	TIMER7_WIDTH	"Timer Width Registers" on page 15-49
0xFFC0 0680	TIMER_ENABLE	"Timer Enable Register" on page 15-39
0xFFC0 0684	TIMER_DISABLE	"Timer Disable Register" on page 15-40
0xFFC0 0688	TIMER_STATUS	"Timer Status Register" on page 15-42

Ports Registers

Ports registers (port F: 0xFFC0 0700 – 0xFFC0 07FF, port G: 0xFFC0 1500 – 0xFFC0 15FF, port H: 0xFFC0 1700 – 0xFFC0 17FF, pin control: 0xFFC0 3200 – 0xFFC0 32FF) are listed in Table A-8.

Table A-8. Ports Registers

Memory-mapped Address	Register Name	See Page
0xFFC0 0700	PORTFIO	"GPIO Data Registers" on page 14-24
0xFFC0 0704	PORTFIO_CLEAR	"GPIO Clear Registers" on page 14-25
0xFFC0 0708	PORTFIO_SET	"GPIO Set Registers" on page 14-25
0xFFC0 070C	PORTFIO_TOGGLE	"GPIO Toggle Registers" on page 14-26
0xFFC0 0710	PORTFIO_MASKA	"GPIO Mask Interrupt A Registers" on page 14-28
0xFFC0 0714	PORTFIO_MASKA_ CLEAR	"GPIO Mask Interrupt A Clear Registers" on page 14-31
0xFFC0 0718	PORTFIO_MASKA_ SET	"GPIO Mask Interrupt A Set Registers" on page 14-29
0xFFC0 071C	PORTFIO_MASKA_ TOGGLE	"GPIO Mask Interrupt A Toggle Registers" on page 14-33

Table A-8. Ports Registers (Cont'd)

Memory-mapped Address	Register Name	See Page
0xFFC0 0720	PORTFIO_MASKB	"GPIO Mask Interrupt B Registers" on page 14-28
0xFFC0 0724	PORTFIO_MASKB_ CLEAR	"GPIO Mask Interrupt B Clear Registers" on page 14-32
0xFFC0 0728	PORTFIO_MASKB_ SET	"GPIO Mask Interrupt B Set Registers" on page 14-30
0xFFC0 072C	PORTFIO_MASKB_ TOGGLE	"GPIO Mask Interrupt B Toggle Registers" on page 14-34
0xFFC0 0730	PORTFIO_DIR	"GPIO Direction Registers" on page 14-23
0xFFC0 0734	PORTFIO_POLAR	"GPIO Polarity Registers" on page 14-26
0xFFC0 0738	PORTFIO_EDGE	"Interrupt Sensitivity Registers" on page 14-27
0xFFC0 073C	PORTFIO_BOTH	"GPIO Set on Both Edges Registers" on page 14-27
0xFFC0 0740	PORTFIO_INEN	"GPIO Input Enable Registers" on page 14-24
0xFFC0 1500	PORTGIO	"GPIO Data Registers" on page 14-24
0xFFC0 1504	PORTGIO_CLEAR	"GPIO Clear Registers" on page 14-25
0xFFC0 1508	PORTGIO_SET	"GPIO Set Registers" on page 14-25
0xFFC0 150C	PORTGIO_TOGGLE	"GPIO Toggle Registers" on page 14-26
0xFFC0 1510	PORTGIO_MASKA	"GPIO Mask Interrupt A Registers" on page 14-28
0xFFC0 1514	PORTGIO_MASKA_ CLEAR	"GPIO Mask Interrupt A Clear Registers" on page 14-31
0xFFC0 1518	PORTGIO_MASKA_ SET	"GPIO Mask Interrupt A Set Registers" on page 14-29
0xFFC0 151C	PORTGIO_MASKA_ TOGGLE	"GPIO Mask Interrupt A Toggle Registers" on page 14-33
0xFFC0 1520	PORTGIO_MASKB	"GPIO Mask Interrupt B Registers" on page 14-28

Table A-8. Ports Registers (Cont'd)

Memory-mapped Address	Register Name	See Page
0xFFC0 1524	PORTGIO_MASKB_CLEAR	"GPIO Mask Interrupt B Clear Registers" on page 14-32
0xFFC0 1528	PORTGIO_MASKB_SET	"GPIO Mask Interrupt B Set Registers" on page 14-30
0xFFC0 152C	PORTGIO_MASKB_TOGGLE	"GPIO Mask Interrupt B Toggle Registers" on page 14-34
0xFFC0 1530	PORTGIO_DIR	"GPIO Direction Registers" on page 14-23
0xFFC0 1534	PORTGIO_POLAR	"GPIO Polarity Registers" on page 14-26
0xFFC0 1538	PORTGIO_EDGE	"Interrupt Sensitivity Registers" on page 14-27
0xFFC0 153C	PORTGIO_BOTH	"GPIO Set on Both Edges Registers" on page 14-27
0xFFC0 1540	PORTGIO_INEN	"GPIO Input Enable Registers" on page 14-24
0xFFC0 1700	PORTHIO	"GPIO Data Registers" on page 14-24
0xFFC0 1704	PORTHIO_CLEAR	"GPIO Clear Registers" on page 14-25
0xFFC0 1708	PORTHIO_SET	"GPIO Set Registers" on page 14-25
0xFFC0 170C	PORTHIO_TOGGLE	"GPIO Toggle Registers" on page 14-26
0xFFC0 1710	PORTHIO_MASKA	"GPIO Mask Interrupt A Registers" on page 14-28
0xFFC0 1714	PORTHIO_MASKA_CLEAR	"GPIO Mask Interrupt A Clear Registers" on page 14-31
0xFFC0 1718	PORTHIO_MASKA_SET	"GPIO Mask Interrupt A Set Registers" on page 14-29
0xFFC0 171C	PORTHIO_MASKA_TOGGLE	"GPIO Mask Interrupt A Toggle Registers" on page 14-33
0xFFC0 1720	PORTHIO_MASKB	"GPIO Mask Interrupt B Registers" on page 14-28
0xFFC0 1724	PORTHIO_MASKB_CLEAR	"GPIO Mask Interrupt B Clear Registers" on page 14-32

Table A-8. Ports Registers (Cont'd)

Memory-mapped Address	Register Name	See Page
0xFFC0 1728	PORTHIO_MASKB_SET	"GPIO Mask Interrupt B Set Registers" on page 14-30
0xFFC0 172C	PORTHIO_MASKB_TOGGLE	"GPIO Mask Interrupt B Toggle Registers" on page 14-34
0xFFC0 1730	PORTHIO_DIR	"GPIO Direction Registers" on page 14-23
0xFFC0 1734	PORTHIO_POLAR	"GPIO Polarity Registers" on page 14-26
0xFFC0 1738	PORTHIO_EDGE	"Interrupt Sensitivity Registers" on page 14-27
0xFFC0 173C	PORTHIO_BOTH	"GPIO Set on Both Edges Registers" on page 14-27
0xFFC0 1740	PORTHIO_INEN	"GPIO Input Enable Registers" on page 14-24
0xFFC0 3200	PORTF_FER	"Function Enable Registers" on page 14-23
0xFFC0 3204	PORTG_FER	"Function Enable Registers" on page 14-23
0xFFC0 3208	PORTH_FER	"Function Enable Registers" on page 14-23
0xFFC0 320C	PORT_MUX	"Port Multiplexer Control Register" on page 14-22

SPORT0 Controller Registers

SPORT0 controller registers (0xFFC0 0800 – 0xFFC0 08FF) are listed in Table A-9.

Table A-9. SPORT0 Controller Registers

Memory-mapped Address	Register Name	See Page
0xFFC0 0800	SPORT0_TCR1	"SPORTx Transmit Configuration 1 Register" on page 12-48
0xFFC0 0804	SPORT0_TCR2	"SPORTx Transmit Configuration 2 Register" on page 12-49
0xFFC0 0808	SPORT0_TCLKDIV	"SPORTx Transmit Serial Clock Divider Register" on page 12-64
0xFFC0 080C	SPORT0_TFSDIV	"SPORTx Transmit Frame Sync Divider Register" on page 12-65
0xFFC0 0810	SPORT0_TX	"SPORTx Transmit Data Register" on page 12-60
0xFFC0 0818	SPORT0_RX	"SPORTx Receive Data Register" on page 12-62
0xFFC0 0820	SPORT0_RCR1	"SPORTx Receive Configuration 1 Register" on page 12-54
0xFFC0 0824	SPORT0_RCR2	"SPORTx Receive Configuration 2 Register" on page 12-55
0xFFC0 0828	SPORT0_RCLKDIV	"SPORTx Receive Serial Clock Divider Register" on page 12-65
0xFFC0 082C	SPORT0_RFSDIV	"SPORTx Receive Frame Sync Divider Register" on page 12-66
0xFFC0 0830	SPORT0_STAT	"SPORTx Status Register" on page 12-64
0xFFC0 0834	SPORT0_CHNL	"SPORTx Current Channel Register" on page 12-68

Table A-9. SPORT0 Controller Registers (Cont'd)

Memory-mapped Address	Register Name	See Page
0xFFC0 0838	SPORT0_MCMC1	"SPORTx Multichannel Configuration Register 1" on page 12-66
0xFFC0 083C	SPORT0_MCMC2	"SPORTx Multichannel Configuration Register 2" on page 12-67
0xFFC0 0840	SPORT0_MTCS0	"SPORTx Multichannel Transmit Select Registers" on page 12-71
0xFFC0 0844	SPORT0_MTCS1	"SPORTx Multichannel Transmit Select Registers" on page 12-71
0xFFC0 0848	SPORT0_MTCS2	"SPORTx Multichannel Transmit Select Registers" on page 12-71
0xFFC0 084C	SPORT0_MTCS3	"SPORTx Multichannel Transmit Select Registers" on page 12-71
0xFFC0 0850	SPORT0_MRCS0	"SPORTx Multichannel Receive Select Registers" on page 12-69
0xFFC0 0854	SPORT0_MRCS1	"SPORTx Multichannel Receive Select Registers" on page 12-69
0xFFC0 0858	SPORT0_MRCS2	"SPORTx Multichannel Receive Select Registers" on page 12-69
0xFFC0 085C	SPORT0_MRCS3	"SPORTx Multichannel Receive Select Registers" on page 12-69

SPORT1 Controller Registers

SPORT1 controller registers (0xFFC0 0900 – 0xFFC0 09FF) are listed in Table A-10.

Table A-10. SPORT 1 Controller Registers

Memory-mapped Address	Register Name	See Page
0xFFC0 0900	SPORT1_TCR1	"SPORTx Transmit Configuration 1 Register" on page 12-48
0xFFC0 0904	SPORT1_TCR2	"SPORTx Transmit Configuration 2 Register" on page 12-49
0xFFC0 0908	SPORT1_TCLKDIV	"SPORTx Transmit Serial Clock Divider Register" on page 12-64
0xFFC0 090C	SPORT1_TFSDIV	"SPORTx Transmit Frame Sync Divider Register" on page 12-65
0xFFC0 0910	SPORT1_TX	"SPORTx Transmit Data Register" on page 12-60
0xFFC0 0918	SPORT1_RX	"SPORTx Receive Data Register" on page 12-62
0xFFC0 0920	SPORT1_RCR1	"SPORTx Receive Configuration 1 Register" on page 12-54
0xFFC0 0924	SPORT1_RCR2	"SPORTx Receive Configuration 2 Register" on page 12-55
0xFFC0 0928	SPORT1_RCLKDIV	"SPORTx Receive Serial Clock Divider Register" on page 12-65
0xFFC0 092C	SPORT1_RFSDIV	"SPORTx Receive Frame Sync Divider Register" on page 12-66
0xFFC0 0930	SPORT1_STAT	"SPORTx Status Register" on page 12-64
0xFFC0 0934	SPORT1_CHNL	"SPORTx Current Channel Register" on page 12-68

Table A-10. SPORT 1 Controller Registers (Cont'd)

Memory-mapped Address	Register Name	See Page
0xFFC0 0938	SPORT1_MCMC1	"SPORTx Multichannel Configuration Register 1" on page 12-66
0xFFC0 093C	SPORT1_MCMC2	"SPORTx Multichannel Configuration Register 2" on page 12-67
0xFFC0 0940	SPORT1_MTCS0	"SPORTx Multichannel Transmit Select Registers" on page 12-71
0xFFC0 0944	SPORT1_MTCS1	"SPORTx Multichannel Transmit Select Registers" on page 12-71
0xFFC0 0948	SPORT1_MTCS2	"SPORTx Multichannel Transmit Select Registers" on page 12-71
0xFFC0 094C	SPORT1_MTCS3	"SPORTx Multichannel Transmit Select Registers" on page 12-71
0xFFC0 0950	SPORT1_MRCS0	"SPORTx Multichannel Receive Select Registers" on page 12-69
0xFFC0 0954	SPORT1_MRCS1	"SPORTx Multichannel Receive Select Registers" on page 12-69
0xFFC0 0958	SPORT1_MRCS2	"SPORTx Multichannel Receive Select Registers" on page 12-69
0xFFC0 095C	SPORT1_MRCS3	"SPORTx Multichannel Receive Select Registers" on page 12-69

External Bus Interface Unit Registers

External bus interface unit registers (0xFFC0 0A00 – 0xFFC0 0AFF) are listed in Table A-11.

Table A-11. External Bus Interface Unit Registers

Memory-mapped Address	Register Name	See Page
0xFFC0 0A00	EBIU_AMGCTL	"Asynchronous Memory Global Control Register" on page 6-21
0xFFC0 0A04	EBIU_AMBCTL0	"Asynchronous Memory Bank Control 0 Register" on page 6-22
0xFFC0 0A08	EBIU_AMBCTL1	"Asynchronous Memory Bank Control 1 Register" on page 6-23
0xFFC0 0A10	EBIU_SDGCTL	"SDRAM Memory Global Control Register" on page 6-68
0xFFC0 0A14	EBIU_SDBCTL	"SDRAM Memory Bank Control Register" on page 6-63
0xFFC0 0A18	EBIU_SDRRC	"SDRAM Refresh Rate Control Register" on page 6-60
0xFFC0 0A1C	EBIU_SDSTAT	"SDRAM Control Status Register" on page 6-78

DMA/Memory DMA Control Registers

DMA control registers (0xFFC0 0B00 – 0xFFC0 0FFF) are listed in Table A-12.

Table A-12. DMA Traffic Control Registers

Memory-mapped Address	Register Name	See Page
0xFFC0 0B0C	DMA_TC_PER	"DMA Traffic Control Counter Period Register" on page 5-105
0xFFC0 0B10	DMA_TC_CNT	"DMA Traffic Control Counter Register" on page 5-106

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-13 and Table A-14. Table A-13 identifies the base address of each DMA channel, as well as the register prefix that identifies the channel. Table A-14 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-13. 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-13. 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-14. DMA Register Suffix and Offset

Register Suffix	Offset From Base	See Page
NEXT_DESC_PTR	0x00	"Next Descriptor Pointer Registers" on page 5-95
START_ADDR	0x04	"Start Address Registers" on page 5-82
CONFIG	0x08	"Configuration Registers" on page 5-74
X_COUNT	0x10	"Inner Loop Count Registers" on page 5-85
X_MODIFY	0x14	"Inner Loop Address Increment Registers" on page 5-89
Y_COUNT	0x18	"Outer Loop Count Registers" on page 5-90

Table A-14. DMA Register Suffix and Offset (Cont'd)

Register Suffix	Offset From Base	See Page
Y_MODIFY	0x1C	"Outer Loop Address Increment Registers" on page 5-93
CURR_DESC_PTR	0x20	"Current Descriptor Pointer Registers" on page 5-97
CURR_ADDR	0x24	"Current Address Registers" on page 5-84
IRQ_STATUS	0x28	"Interrupt Status Registers" on page 5-79
PERIPHERAL_MAP	0x2C	"Peripheral Map Registers" on page 5-71
CURR_X_COUNT	0x30	"Current Inner Loop Count Registers" on page 5-87
CURR_Y_COUNT	0x38	"Current Outer Loop Count Registers" on page 5-92

PPI Registers

PPI registers (0xFFC0 1000 – 0xFFC0 10FF) are listed in Table A-15.

Table A-15. PPI Registers

Memory-mapped Address	Register Name	See Page
0xFFC0 1000	PPI_CONTROL	"PPI Control Register" on page 7-27
0xFFC0 1004	PPI_STATUS	"PPI Status Register" on page 7-33
0xFFC0 1008	PPI_COUNT	"Transfer Count Register" on page 7-35
0xFFC0 100C	PPI_DELAY	"Delay Count Register" on page 7-34
0xFFC0 1010	PPI_FRAME	"Lines Per Frame Register" on page 7-36

UART1 Controller Registers

UART1 controller registers (0xFFC0 2000 – 0xFFC0 20FF) are listed in Table A-16. For UART0 registers, see Table A-5 on page A-4.

Table A-16. UART1 Controller Registers

Memory-mapped Address	Register Name	See Page
0xFFC0 2000	UART1_THR	"UART Transmit Holding Registers" on page 13-26
0xFFC0 2000	UART1_RBR	"UART Receive Buffer Registers" on page 13-26
0xFFC0 2000	UART1_DLL	"UART Divisor Latch Registers" on page 13-30
0xFFC0 2004	UART1_DLH	"UART Divisor Latch Registers" on page 13-30
0xFFC0 2004	UART1_IER	"UART Interrupt Enable Registers" on page 13-28
0xFFC0 2008	UART1_IIR	"UART Interrupt Identification Registers" on page 13-29
0xFFC0 200C	UART1_LCR	"UART Line Control Registers" on page 13-21
0xFFC0 2010	UART1_MCR	"UART Modem Control Registers" on page 13-23
0xFFC0 2014	UART1_LSR	"UART Line Status Registers" on page 13-24
0xFFC0 201C	UART1_SCR	"UART Scratch Registers" on page 13-31
0xFFC0 2024	UART1_GCTL	"UART Global Control Registers" on page 13-31

CAN Registers

CAN registers (0xFFC0 2A00 – 0xFFC0 2FFF) are listed in Table A-17 through Table A-20.

Table A-17. CAN Control and Configuration Registers

Memory-mapped Address	Register Name	See Page
0xFFC0 2A00	CAN_MC1	"Mailbox Configuration Register 1" on page 9-69
0xFFC0 2A04	CAN_MD1	"Mailbox Direction Register 1" on page 9-70
0xFFC0 2A08	CAN_TRS1	"Transmission Request Set Register 1" on page 9-74
0xFFC0 2A0C	CAN_TRR1	"Transmission Request Reset Register 1" on page 9-75
0xFFC0 2A10	CAN_TA1	"Transmission Acknowledge Register 1" on page 9-77
0xFFC0 2A14	CAN_AA1	"Abort Acknowledge Register 1" on page 9-76
0xFFC0 2A18	CAN_RMP1	"Receive Message Pending Register 1" on page 9-71
0xFFC0 2A1C	CAN_RML1	"Receive Message Lost Register 1" on page 9-72
0xFFC0 2A20	CAN_MBTIF1	"Mailbox Transmit Interrupt Flag Register 1" on page 9-80
0xFFC0 2A24	CAN_MBRIF1	"Mailbox Receive Interrupt Flag Register 1" on page 9-81
0xFFC0 2A28	CAN_MBIM1	"Mailbox Interrupt Mask Register 1" on page 9-79
0xFFC0 2A2C	CAN_RFH1	"Remote Frame Handling Register 1" on page 9-78
0xFFC0 2A30	CAN_OPSS1	"Overwrite Protection/Single Shot Transmission Register 1" on page 9-73

Table A-17. CAN Control and Configuration Registers (Cont'd)

Memory-mapped Address	Register Name	See Page
0xFFC0 2A40	CAN_MC2	"Mailbox Configuration Register 2" on page 9-69
0xFFC0 2A44	CAN_MD2	"Mailbox Direction Register 2" on page 9-70
0xFFC0 2A48	CAN_TRS2	"Transmission Request Set Register 2" on page 9-74
0xFFC0 2A4C	CAN_TRR2	"Transmission Request Reset Register 2" on page 9-75
0xFFC0 2A50	CAN_TA2	"Transmission Acknowledge Register 2" on page 9-77
0xFFC0 2A54	CAN_AA2	"Abort Acknowledge Register 2" on page 9-76
0xFFC0 2A58	CAN_RMP2	"Receive Message Pending Register 2" on page 9-71
0xFFC0 2A5C	CAN_RML2	"Receive Message Lost Register 2" on page 9-72
0xFFC0 2A60	CAN_MBTIF2	"Mailbox Transmit Interrupt Flag Register 2" on page 9-81
0xFFC0 2A64	CAN_MBRIF2	"Mailbox Receive Interrupt Flag Register 2" on page 9-82
0xFFC0 2A68	CAN_MBIM2	"Mailbox Interrupt Mask Register 2" on page 9-80
0xFFC0 2A6C	CAN_RFH2	"Remote Frame Handling Register 2" on page 9-79
0xFFC0 2A70	CAN_OPSS2	"Overwrite Protection/Single Shot Transmission Register 2" on page 9-73
0xFFC0 2A80	CAN_CLOCK	"CAN Clock Register" on page 9-45
0xFFC0 2A84	CAN_TIMING	"CAN Timing Register" on page 9-46
0xFFC0 2A8C	CAN_STATUS	"Global Status Register" on page 9-44
0xFFC0 2A90	CAN_CEC	"CAN Error Counter Register" on page 9-85

Table A-17. CAN Control and Configuration Registers (Cont'd)

Memory-mapped Address	Register Name	See Page
0xFFC0 2A94	CAN_GIS	"Global Interrupt Status Register" on page 9-47
0xFFC0 2A98	CAN_GIM	"Global Interrupt Mask Register" on page 9-47
0xFFC0 2A9C	CAN_GIF	"Global Interrupt Flag Register" on page 9-48
0xFFC0 2AA0	CAN_CONTROL	"Master Control Register" on page 9-43
0xFFC0 2AA4	CAN_INTR	"CAN Interrupt Register" on page 9-46
0xFFC0 2AAC	CAN_MBTD	"Temporary Mailbox Disable Register" on page 9-78
0xFFC0 2AB0	CAN_EWR	"CAN Error Counter Warning Level Register" on page 9-86
0xFFC0 2AB4	CAN_ESR	"Error Status Register" on page 9-85
0xFFC0 2AC4	CAN_UCCNT	"Universal Counter Register" on page 9-84
0xFFC0 2AC8	CAN_UCRC	"Universal Counter Reload/Capture Register" on page 9-84
0xFFC0 2ACC	CAN_UCCNF	"Universal Counter Configuration Mode Register" on page 9-83

Table A-18. CAN Mailbox Acceptance Mask Registers

Memory-mapped Address	Register Name	See Section
0xFFC0 2B00	CAN_AM00L	"Acceptance Mask Register (L)" on page 9-51
0xFFC0 2B04	CAN_AM00H	"Acceptance Mask Register (H)" on page 9-49
0xFFC0 2B08	CAN_AM01L	"Acceptance Mask Register (L)" on page 9-51
0xFFC0 2B0C	CAN_AM01H	"Acceptance Mask Register (H)" on page 9-49
0xFFC0 2B10	CAN_AM02L	"Acceptance Mask Register (L)" on page 9-51
0xFFC0 2B14	CAN_AM02H	"Acceptance Mask Register (H)" on page 9-49

Table A-18. CAN Mailbox Acceptance Mask Registers (Cont'd)

Memory-mapped Address	Register Name	See Section
0xFFC0 2B18	CAN_AM03L	"Acceptance Mask Register (L)" on page 9-51
0xFFC0 2B1C	CAN_AM03H	"Acceptance Mask Register (H)" on page 9-49
0xFFC0 2B20	CAN_AM04L	"Acceptance Mask Register (L)" on page 9-51
0xFFC0 2B24	CAN_AM04H	"Acceptance Mask Register (H)" on page 9-49
0xFFC0 2B28	CAN_AM05L	"Acceptance Mask Register (L)" on page 9-51
0xFFC0 2B2C	CAN_AM05H	"Acceptance Mask Register (H)" on page 9-49
0xFFC0 2B30	CAN_AM06L	"Acceptance Mask Register (L)" on page 9-51
0xFFC0 2B34	CAN_AM06H	"Acceptance Mask Register (H)" on page 9-49
0xFFC0 2B38	CAN_AM07L	"Acceptance Mask Register (L)" on page 9-51
0xFFC0 2B3C	CAN_AM07H	"Acceptance Mask Register (H)" on page 9-49
0xFFC0 2B40	CAN_AM08L	"Acceptance Mask Register (L)" on page 9-51
0xFFC0 2B44	CAN_AM08H	"Acceptance Mask Register (H)" on page 9-49
0xFFC0 2B48	CAN_AM09L	"Acceptance Mask Register (L)" on page 9-51
0xFFC0 2B4C	CAN_AM09H	"Acceptance Mask Register (H)" on page 9-49
0xFFC0 2B50	CAN_AM10L	"Acceptance Mask Register (L)" on page 9-51
0xFFC0 2B54	CAN_AM10H	"Acceptance Mask Register (H)" on page 9-49
0xFFC0 2B58	CAN_AM11L	"Acceptance Mask Register (L)" on page 9-51
0xFFC0 2B5C	CAN_AM11H	"Acceptance Mask Register (H)" on page 9-49
0xFFC0 2B60	CAN_AM12L	"Acceptance Mask Register (L)" on page 9-51
0xFFC0 2B64	CAN_AM12H	"Acceptance Mask Register (H)" on page 9-49
0xFFC0 2B68	CAN_AM13L	"Acceptance Mask Register (L)" on page 9-51
0xFFC0 2B6C	CAN_AM13H	"Acceptance Mask Register (H)" on page 9-49
0xFFC0 2B70	CAN_AM14L	"Acceptance Mask Register (L)" on page 9-51
0xFFC0 2B74	CAN_AM14H	"Acceptance Mask Register (H)" on page 9-49

Table A-18. CAN Mailbox Acceptance Mask Registers (Cont'd)

Memory-mapped Address	Register Name	See Section
0xFFC0 2B78	CAN_AM15L	"Acceptance Mask Register (L)" on page 9-51
0xFFC0 2B7C	CAN_AM15H	"Acceptance Mask Register (H)" on page 9-49
0xFFC0 2B80	CAN_AM16L	"Acceptance Mask Register (L)" on page 9-51
0xFFC0 2B84	CAN_AM16H	"Acceptance Mask Register (H)" on page 9-49
0xFFC0 2B88	CAN_AM17L	"Acceptance Mask Register (L)" on page 9-51
0xFFC0 2B8C	CAN_AM17H	"Acceptance Mask Register (H)" on page 9-49
0xFFC0 2B90	CAN_AM18L	"Acceptance Mask Register (L)" on page 9-51
0xFFC0 2B94	CAN_AM18H	"Acceptance Mask Register (H)" on page 9-49
0xFFC0 2B98	CAN_AM19L	"Acceptance Mask Register (L)" on page 9-51
0xFFC0 2B9C	CAN_AM19H	"Acceptance Mask Register (H)" on page 9-49
0xFFC0 2BA0	CAN_AM20L	"Acceptance Mask Register (L)" on page 9-51
0xFFC0 2BA4	CAN_AM20H	"Acceptance Mask Register (H)" on page 9-49
0xFFC0 2BA8	CAN_AM21L	"Acceptance Mask Register (L)" on page 9-51
0xFFC0 2BAC	CAN_AM21H	"Acceptance Mask Register (H)" on page 9-49
0xFFC0 2BB0	CAN_AM22L	"Acceptance Mask Register (L)" on page 9-51
0xFFC0 2BB4	CAN_AM22H	"Acceptance Mask Register (H)" on page 9-49
0xFFC0 2BB8	CAN_AM23L	"Acceptance Mask Register (L)" on page 9-51
0xFFC0 2BBC	CAN_AM23H	"Acceptance Mask Register (H)" on page 9-49
0xFFC0 2BC0	CAN_AM24L	"Acceptance Mask Register (L)" on page 9-51
0xFFC0 2BC4	CAN_AM24H	"Acceptance Mask Register (H)" on page 9-49
0xFFC0 2BC8	CAN_AM25L	"Acceptance Mask Register (L)" on page 9-51
0xFFC0 2BCC	CAN_AM25H	"Acceptance Mask Register (H)" on page 9-49
0xFFC0 2BD0	CAN_AM26L	"Acceptance Mask Register (L)" on page 9-51
0xFFC0 2BD4	CAN_AM26H	"Acceptance Mask Register (H)" on page 9-49

Table A-18. CAN Mailbox Acceptance Mask Registers (Cont'd)

Memory-mapped Address	Register Name	See Section
0xFFC0 2BD8	CAN_AM27L	"Acceptance Mask Register (L)" on page 9-51
0xFFC0 2BDC	CAN_AM27H	"Acceptance Mask Register (H)" on page 9-49
0xFFC0 2BE0	CAN_AM28L	"Acceptance Mask Register (L)" on page 9-51
0xFFC0 2BE4	CAN_AM28H	"Acceptance Mask Register (H)" on page 9-49
0xFFC0 2BE8	CAN_AM29L	"Acceptance Mask Register (L)" on page 9-51
0xFFC0 2BEC	CAN_AM29H	"Acceptance Mask Register (H)" on page 9-49
0xFFC0 2BF0	CAN_AM30L	"Acceptance Mask Register (L)" on page 9-51
0xFFC0 2BF4	CAN_AM30H	"Acceptance Mask Register (H)" on page 9-49
0xFFC0 2BF8	CAN_AM31L	"Acceptance Mask Register (L)" on page 9-51
0xFFC0 2BFC	CAN_AM31H	"Acceptance Mask Register (H)" on page 9-49

Since each CAN mailbox has an identical MMR set, with fixed offsets from the base address associated with that mailbox, it is convenient to view the MMR information as provided in Table A-19 and Table A-20. Table A-19 identifies the base address of each CAN mailbox, as well as the register prefix that identifies mailbox. Table A-20 then lists the register suffix and provides its offset from the base address.

As an example, the CAN mailbox 2 length register is called `CAN_MB02_LENGTH`, and its address is 0xFFC0 2C50. Likewise, the CAN mailbox 17 timestamp register is called `CAN_MB17_TIMESTAMP`, and its address is 0xFFC0 2E34.

Table A-19. CAN Mailbox Base Addresses

Mailbox Identifier	MMR Base Address	Register Prefix
0	0xFFC0 2C00	CAN_MB00_
1	0xFFC0 2C20	CAN_MB01_

Table A-19. CAN Mailbox Base Addresses (Cont'd)

Mailbox Identifier	MMR Base Address	Register Prefix
2	0xFFC0 2C40	CAN_MB02_
3	0xFFC0 2C60	CAN_MB03_
4	0xFFC0 2C80	CAN_MB04_
5	0xFFC0 2CA0	CAN_MB05_
6	0xFFC0 2CC0	CAN_MB06_
7	0xFFC0 2CE0	CAN_MB07_
8	0xFFC0 2D00	CAN_MB08_
9	0xFFC0 2D20	CAN_MB09_
10	0xFFC0 2D40	CAN_MB10_
11	0xFFC0 2D60	CAN_MB11_
12	0xFFC0 2D80	CAN_MB12_
13	0xFFC0 2DA0	CAN_MB13_
14	0xFFC0 2DC0	CAN_MB14_
15	0xFFC0 2DE0	CAN_MB15_
16	0xFFC0 2E00	CAN_MB16_
17	0xFFC0 2E20	CAN_MB17_
18	0xFFC0 2E40	CAN_MB18_
19	0xFFC0 2E60	CAN_MB19_
20	0xFFC0 2E80	CAN_MB20_
21	0xFFC0 2EA0	CAN_MB21_
22	0xFFC0 2EC0	CAN_MB22_
23	0xFFC0 2EE0	CAN_MB23_
24	0xFFC0 2F00	CAN_MB24_
25	0xFFC0 2F20	CAN_MB25_

Table A-19. CAN Mailbox Base Addresses (Cont'd)

Mailbox Identifier	MMR Base Address	Register Prefix
26	0xFFC0 2F40	CAN_MB26_
27	0xFFC0 2F60	CAN_MB27_
28	0xFFC0 2F80	CAN_MB28_
29	0xFFC0 2FA0	CAN_MB29_
30	0xFFC0 2FC0	CAN_MB30_
31	0xFFC0 2FE0	CAN_MB31_

Table A-20. CAN Mailbox Register Suffix and Offset

Register Suffix	Offset From Base	See Page
DATA0	0x00	"Mailbox Word 0 Register" on page 9-67
DATA1	0x04	"Mailbox Word 1 Register" on page 9-65
DATA2	0x08	"Mailbox Word 2 Register" on page 9-63
DATA3	0x0C	"Mailbox Word 3 Register" on page 9-61
LENGTH	0x10	"Mailbox Word 4 Register" on page 9-59
TIMESTAMP	0x14	"Mailbox Word 5 Register" on page 9-57
ID0	0x18	"Mailbox Word 6 Register" on page 9-55
ID1	0x1C	"Mailbox Word 7 Register" on page 9-53

Ethernet MAC Registers

Ethernet MAC registers (0xFFC0 3000 – 0xFFC0 31FF) are listed in Table A-21.

Table A-21. Ethernet MAC Registers

Memory-mapped Address	Register Name	See Page
0xFFC0 3000	EMAC_OPMODE	"MAC Operating Mode Register" on page 8-64
0xFFC0 3004	EMAC_ADDRLO	"MAC Address Low Register" on page 8-71
0xFFC0 3008	EMAC_ADDRHI	"MAC Address High Register" on page 8-71
0xFFC0 300C	EMAC_HASHLO	"MAC Multicast Hash Table Low Register" on page 8-74
0xFFC0 3010	EMAC_HASHHI	"MAC Multicast Hash Table High Register" on page 8-75
0xFFC0 3014	EMAC_STAADD	"MAC Station Management Address Register" on page 8-76
0xFFC0 3018	EMAC_STADAT	"MAC Station Management Data Register" on page 8-78
0xFFC0 301C	EMAC_FLC	"MAC Flow Control Register" on page 8-79
0xFFC0 3020	EMAC_VLAN1	"MAC VLAN1 Tag Register" on page 8-81
0xFFC0 3024	EMAC_VLAN2	"MAC VLAN2 Tag Register" on page 8-81
0xFFC0 302C	EMAC_WKUP_CTL	"MAC Wakeup Frame Control and Status Register" on page 8-82
0xFFC0 3030	EMAC_WKUP_ FFMSK0	"MAC Wakeup Frame0 Byte Mask Register" on page 8-85
0xFFC0 3034	EMAC_WKUP_ FFMSK1	"MAC Wakeup Frame1 Byte Mask Register" on page 8-86
0xFFC0 3038	EMAC_WKUP_ FFMSK2	"MAC Wakeup Frame2 Byte Mask Register" on page 8-87

Table A-21. Ethernet MAC Registers (Cont'd)

Memory-mapped Address	Register Name	See Page
0xFFC0 303C	EMAC_WKUP_ FFMSK3	"MAC Wakeup Frame3 Byte Mask Register" on page 8-88
0xFFC0 3040	EMAC_WKUP_ FFCMD	"MAC Wakeup Frame Filter Commands Register" on page 8-89
0xFFC0 3044	EMAC_WKUP_FFOFF	"Ethernet MAC Wakeup Frame Filter Offsets Register" on page 8-91
0xFFC0 3048	EMAC_WKUP_ FFCRC0/1	"MAC Wakeup Frame Filter CRC0/1 Register" on page 8-92
0xFFC0 304C	EMAC_WKUP_ FFCRC2/3	"MAC Wakeup Frame Filter CRC2/3 Register" on page 8-92
0xFFC0 3060	EMAC_SYSCTL	"MAC System Control Register" on page 8-93
0xFFC0 3064	EMAC_SYSTAT	"MAC System Status Register" on page 8-95
0xFFC0 3068	EMAC_RX_STAT	"Ethernet MAC RX Current Frame Status Register" on page 8-98
0xFFC0 306C	EMAC_RX_STKY	"Ethernet MAC RX Sticky Frame Status Register" on page 8-103
0xFFC0 3070	EMAC_RX_IRQE	"Ethernet MAC RX Frame Status Interrupt Enable Register" on page 8-107
0xFFC0 3074	EMAC_TX_STAT	"Ethernet MAC TX Current Frame Status Register" on page 8-108
0xFFC0 3078	EMAC_TX_STKY	"Ethernet MAC TX Sticky Frame Status Register" on page 8-112
0xFFC0 307C	EMAC_TX_IRQE	"Ethernet MAC TX Frame Status Interrupt Enable Register" on page 8-114
0xFFC0 3080	EMAC_MMC_CTL	"MAC Management Counters Control Register" on page 8-124
0xFFC0 3084	EMAC_MMC_RIRQS	"Ethernet MAC MMC RX Interrupt Status Register" on page 8-116

Table A-21. Ethernet MAC Registers (Cont'd)

Memory-mapped Address	Register Name	See Page
0xFFC0 3088	EMAC_MMC_RIRQE	"Ethernet MAC MMC RX Interrupt Enable Register" on page 8-118
0xFFC0 308C	EMAC_MMC_TIRQS	"Ethernet MAC MMC TX Interrupt Status Register" on page 8-120
0xFFC0 3090	EMAC_MMC_TIRQE	"Ethernet MAC MMC TX Interrupt Enable Register" on page 8-122
0xFFC0 3100	EMAC_RXC_OK	"MAC Management Counter Registers" on page 8-54
0xFFC0 3104	EMAC_RXC_FCS	"MAC Management Counter Registers" on page 8-54
0xFFC0 3108	EMAC_RXC_ALIGN	"MAC Management Counter Registers" on page 8-54
0xFFC0 310C	EMAC_RXC_OCTET	"MAC Management Counter Registers" on page 8-54
0xFFC0 3110	EMAC_RXC_ DMAOVF	"MAC Management Counter Registers" on page 8-54
0xFFC0 3114	EMAC_RXC_UNICST	"MAC Management Counter Registers" on page 8-54
0xFFC0 3118	EMAC_RXC_MULTI	"MAC Management Counter Registers" on page 8-54
0xFFC0 311C	EMAC_RXC_BROAD	"MAC Management Counter Registers" on page 8-54
0xFFC0 3120	EMAC_RXC_LNERRI	"MAC Management Counter Registers" on page 8-54
0xFFC0 3124	EMAC_RXC_LNERRO	"MAC Management Counter Registers" on page 8-54
0xFFC0 3128	EMAC_RXC_LONG	"MAC Management Counter Registers" on page 8-54
0xFFC0 312C	EMAC_RXC_MACCTL	"MAC Management Counter Registers" on page 8-54

Table A-21. Ethernet MAC Registers (Cont'd)

Memory-mapped Address	Register Name	See Page
0xFFC0 3130	EMAC_RXC_OPCODE	"MAC Management Counter Registers" on page 8-54
0xFFC0 3134	EMAC_RXC_PAUSE	"MAC Management Counter Registers" on page 8-54
0xFFC0 3138	EMAC_RXC_ALLFRM	"MAC Management Counter Registers" on page 8-54
0xFFC0 313C	EMAC_RXC_ALLOCT	"MAC Management Counter Registers" on page 8-54
0xFFC0 3140	EMAC_RXC_TYPED	"MAC Management Counter Registers" on page 8-54
0xFFC0 3144	EMAC_RXC_SHORT	"MAC Management Counter Registers" on page 8-54
0xFFC0 3148	EMAC_RXC_EQ64	"MAC Management Counter Registers" on page 8-54
0xFFC0 314C	EMAC_RXC_LT128	"MAC Management Counter Registers" on page 8-54
0xFFC0 3150	EMAC_RXC_LT256	"MAC Management Counter Registers" on page 8-54
0xFFC0 3154	EMAC_RXC_LT512	"MAC Management Counter Registers" on page 8-54
0xFFC0 3158	EMAC_RXC_LT1024	"MAC Management Counter Registers" on page 8-54
0xFFC0 315C	EMAC_RXC_GE1024	"MAC Management Counter Registers" on page 8-54
0xFFC0 3180	EMAC_TXC_OK	"MAC Management Counter Registers" on page 8-54
0xFFC0 3184	EMAC_TXC_1COL	"MAC Management Counter Registers" on page 8-54
0xFFC0 3188	EMAC_TXC_GT1COL	"MAC Management Counter Registers" on page 8-54

Table A-21. Ethernet MAC Registers (Cont'd)

Memory-mapped Address	Register Name	See Page
0xFFC0 318C	EMAC_TXC_OCTET	"MAC Management Counter Registers" on page 8-54
0xFFC0 3190	EMAC_TXC_DEFER	"MAC Management Counter Registers" on page 8-54
0xFFC0 3194	EMAC_TXC_LATECL	"MAC Management Counter Registers" on page 8-54
0xFFC0 3198	EMAC_TXC_XS_COL	"MAC Management Counter Registers" on page 8-54
0xFFC0 319C	EMAC_TXC_ DMAUND	"MAC Management Counter Registers" on page 8-54
0xFFC0 31A0	EMAC_TXC_CRSERR	"MAC Management Counter Registers" on page 8-54
0xFFC0 31A4	EMAC_TXC_UNICST	"MAC Management Counter Registers" on page 8-54
0xFFC0 31A8	EMAC_TXC_MULTI	"MAC Management Counter Registers" on page 8-54
0xFFC0 31AC	EMAC_TXC_BROAD	"MAC Management Counter Registers" on page 8-54
0xFFC0 31B0	EMAC_TXC_ES_DFR	"MAC Management Counter Registers" on page 8-54
0xFFC0 31B4	EMAC_TXC_MACCTL	"MAC Management Counter Registers" on page 8-54
0xFFC0 31B8	EMAC_TXC_ALLFRM	"MAC Management Counter Registers" on page 8-54
0xFFC0 31BC	EMAC_TXC_ALLOCT	"MAC Management Counter Registers" on page 8-54
0xFFC0 31C0	EMAC_TXC_EQ64	"MAC Management Counter Registers" on page 8-54
0xFFC0 31C4	EMAC_TXC_LT128	"MAC Management Counter Registers" on page 8-54

Table A-21. Ethernet MAC Registers (Cont'd)

Memory-mapped Address	Register Name	See Page
0xFFC0 31C8	EMAC_TXC_LT254	"MAC Management Counter Registers" on page 8-54
0xFFC0 31CC	EMAC_TXC_LT512	"MAC Management Counter Registers" on page 8-54
0xFFC0 31D0	EMAC_TXC_LT1024	"MAC Management Counter Registers" on page 8-54
0xFFC0 31D4	EMAC_TXC_GE1024	"MAC Management Counter Registers" on page 8-54
0xFFC0 31D8	EMAC_TXC_ABORT	"MAC Management Counter Registers" on page 8-54

Handshake MDMA Control Registers

HMDMA registers (0xFFC0 3300 – 0xFFC0 33FF) are listed in Table A-22.

Table A-22. HMDMA Registers

Memory-mapped Address	Register Name	See Page
0xFFC0 3300	HMDMA0_CONTROL	"Handshake MDMA Control Registers" on page 5-100
0xFFC0 3304	HMDMA0_ECINIT	"Handshake MDMA Initial Edge Count Registers" on page 5-103
0xFFC0 3308	HMDMA0_BCINIT	"Handshake MDMA Initial Block Count Registers" on page 5-101
0xFFC0 330C	HMDMA0_ ECURGENT	"Handshake MDMA Edge Count Urgent Registers" on page 5-104
0xFFC0 3310	HMDMA0_ ECOVERFLOW	"Handshake MDMA Edge Count Overflow Interrupt Registers" on page 5-104

Table A-22. HMDMA Registers (Cont'd)

Memory-mapped Address	Register Name	See Page
0xFFC0 3314	HMDMA0_ECOUNT	"Handshake MDMA Current Edge Count Registers" on page 5-103
0xFFC0 3318	HMDMA0_BCOUNT	"Handshake MDMA Current Block Count Registers" on page 5-102
0xFFC0 3340	HMDMA1_CONTROL	"Handshake MDMA Control Registers" on page 5-100
0xFFC0 3344	HMDMA1_ECINIT	"Handshake MDMA Initial Edge Count Registers" on page 5-103
0xFFC0 3348	HMDMA1_BCINIT	"Handshake MDMA Initial Block Count Registers" on page 5-101
0xFFC0 334C	HMDMA1_ ECURGENT	"Handshake MDMA Edge Count Urgent Registers" on page 5-104
0xFFC0 3350	HMDMA1_ ECOVERFLOW	"Handshake MDMA Edge Count Overflow Interrupt Registers" on page 5-104
0xFFC0 3354	HMDMA1_ECOUNT	"Handshake MDMA Current Edge Count Registers" on page 5-103
0xFFC0 3358	HMDMA1_BCOUNT	"Handshake MDMA Current Block Count Registers" on page 5-102

Core Timer Registers

Core timer registers (0xFFE0 3000 – 0xFFE0 300C) are listed in Table A-23.

Table A-23. Core Timer Registers

Memory-mapped Address	Register Name	See Page
0xFFE0 3000	TCNTL	"Core Timer Control Register" on page 16-4
0xFFE0 3004	TPERIOD	"Core Timer Period Register" on page 16-6
0xFFE0 3008	TSCALE	"Core Timer Scale Register" on page 16-7
0xFFE0 300C	TCOUNT	"Core Timer Count Register" on page 16-5

Processor-Specific Memory Registers

Processor-specific memory registers (0xFFE0 0004 – 0xFFE0 0300) are listed in Table A-24.

Table A-24. Processor-Specific Memory Registers

Memory-mapped Address	Register Name	See Page
0xFFE0 0004	DMEM_CONTROL	"L1 Data Memory Control Register" on page 3-9
0xFFE0 0300	DTEST_COMMAND	"Data Test Command Register" on page 3-10

B TEST FEATURES

This chapter discusses the test features of the processor.

JTAG Standard

The processor is fully compatible with the IEEE 1149.1 standard, also known as the Joint Test Action Group (JTAG) standard.

The JTAG standard defines circuitry that may be built to assist in the test, maintenance, and support of assembled printed circuit boards.The circuitry includes a standard interface through which instructions and test data are communicated. A set of test features is defined, including a boundary-scan register, such that the component can respond to a minimum set of instructions designed to help test printed circuit boards.

The standard defines test logic that can be included in an integrated circuit to provide standardized approaches to:

- Testing the interconnections between integrated circuits once they have been assembled onto a printed circuit board
- Testing the integrated circuit itself
- Observing or modifying circuit activity during normal component operation

The test logic consists of a boundary-scan register and other building blocks. The test logic is accessed through a Test Access Port (TAP).

Full details of the JTAG standard can be found in the document *IEEE Standard Test Access Port and Boundary-Scan Architecture*, ISBN 1-55937-350-4.

Boundary-Scan Architecture

The boundary-scan test logic consists of:

- A TAP comprised of five pins (see Table B-1)
- A TAP controller that controls all sequencing of events through the test registers
- An Instruction register (IR) that interprets 5-bit instruction codes to select the test mode that performs the desired test operation
- Several data registers defined by the JTAG standard

Table B-1. Test Access Port Pins

Pin Name	Input/Output	Description
TDI	Input	Test Data Input
TMS	Input	Test Mode Select
TCK	Input	Test Clock
$\overline{\text{TRST}}$	Input	Test Reset
TDO	Output	Test Data Out

The TAP controller is a synchronous, 16-state, finite-state machine controlled by the TCK and TMS pins. Transitions to the various states in the diagram occur on the rising edge of TCK and are defined by the state of the TMS pin, here denoted by either a logic 1 or logic 0 state. For full details of the operation, see the JTAG standard.

Figure B-1 shows the state diagram for the TAP controller.

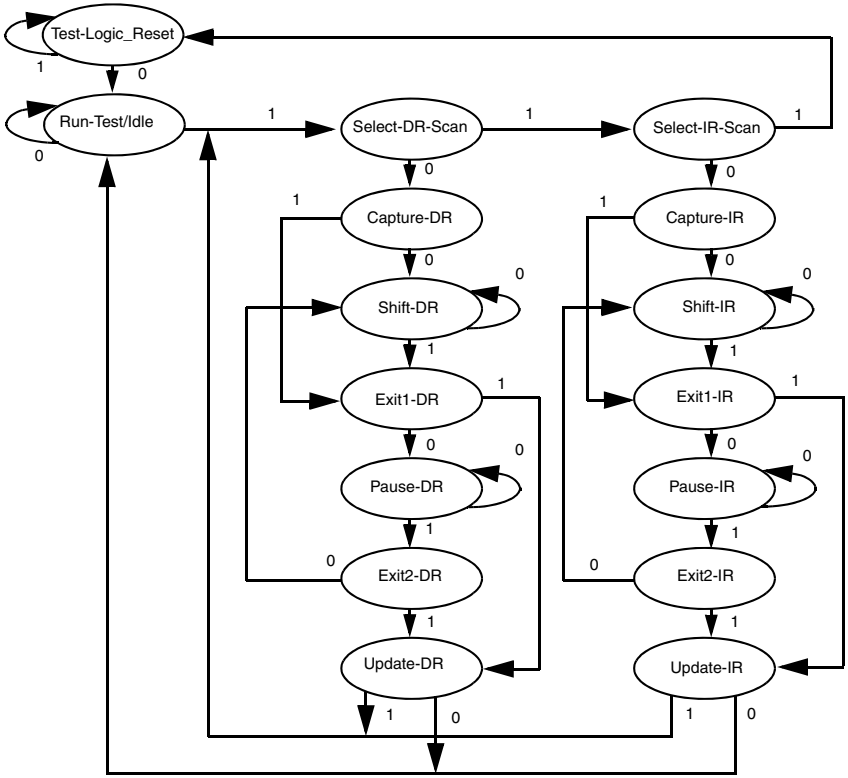


Figure B-1. TAP Controller State Diagram

Note:

- The TAP controller enters the test-logic-reset state when TMS is held high after five TCK cycles.
- The TAP controller enters the test-logic-reset state when TRST is asynchronously asserted.
- An external system reset does not affect the state of the TAP controller, nor does the state of the TAP controller affect an external system reset.

Instruction Register

The instruction register is five bits wide and accommodates up to 32 boundary-scan instructions.

The instruction register holds both public and private instructions. The JTAG standard requires some of the public instructions; other public instructions are optional. Private instructions are reserved for the manufacturer's use.

The binary decode column of Table B-2 lists the decode for the public instructions. The register column lists the serial scan paths.

Table B-2. Decode for Public JTAG-Scan Instructions

Instruction Name	Binary Decode 01234	Register
EXTEST	00000	Boundary-Scan
SAMPLE/PRELOAD	10000	Boundary-Scan
BYPASS	11111	Bypass

Figure B-2 shows the instruction bit scan ordering for the paths shown in Table B-2.

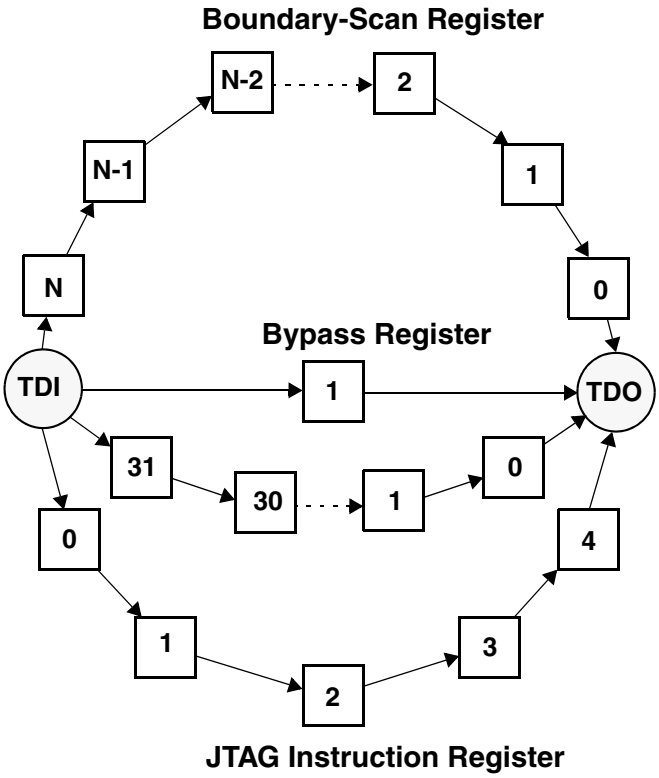


Figure B-2. Serial Scan Paths

Public Instructions

The following sections describe the public JTAG scan instructions.

EXTEST – Binary Code 00000

The `EXTEST` instruction selects the boundary-scan register to be connected between the `TDI` and `TDO` pins. This instruction allows testing of on-board circuitry external to the device.

The EXTEST instruction allows internal data to be driven to the boundary outputs and external data to be captured on the boundary inputs.

To protect the internal logic when the boundary outputs are over-driven or signals are received on the boundary inputs, make sure that nothing else drives data on the processor's output pins.

SAMPLE/PRELOAD – Binary Code 10000

The SAMPLE/PRELOAD instruction performs two functions and selects the Boundary-Scan register to be connected between TDI and TDO. The instruction has no effect on internal logic.

The SAMPLE part of the instruction allows a snapshot of the inputs and outputs captured on the boundary-scan cells. Data is sampled on the rising edge of TCK.

The PRELOAD part of the instruction allows data to be loaded on the device pins and driven out on the board with the EXTEST instruction. Data is preloaded on the pins on the falling edge of TCK.

BYPASS – Binary Code 11111

The BYPASS instruction selects the BYPASS register to be connected to TDI and TDO. The instruction has no effect on the internal logic. No data inversion should occur between TDI and TDO.

Boundary-Scan Register

The boundary-scan register is selected by the EXTEST and SAMPLE/PRELOAD instructions. These instructions allow the pins of the processor to be controlled and sampled for board-level testing.

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 = 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 will be testable.

Digital Signal Processor (DSP).

An integrated circuit designated for high-speed manipulation of analog information that has been 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 has been 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.

I^2C.

A bus standard specified in the *Philips I^2C 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 via 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 $\overline{\text{SMS}}$ pin.

Self-Refresh.

When the SDRAM is in self-refresh mode, the SDRAM's internal timer initiates auto-refresh cycles periodically, without external control input. The SDRAM Controller (SDC) must issue a series of commands including the self-refresh command to put SDRAM into low power mode, and it must issue another series of commands to exit self-refresh mode. Entering self-refresh mode is programmed in the SDRAM memory global control

register (EBIU_SDGCTL) and any access to the SDRAM address space causes the SDC to exit SDRAM from self-refresh mode. See "Enter Self-Refresh Mode" on page 6-35 and "Exit Self-Refresh Mode" on page 6-36.

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 I^2C 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.

I INDEX

Symbols

µ-law companding, 12-24, 12-29

Numerics

DMC, 3-8
16-bit flash, booting from, 19-40
16-bit flash interface, 21-5
16-bit SRAM, interface, 21-5
24 hours event flag bit, 18-21
24 hours interrupt enable bit, 18-20
2D DMA, 5-14
8-bit flash, booting from, 19-36
8-bit flash interface, 21-5
8-bit SRAM interface, 21-5

A

A[10] pin, 6-55
AAIF bit, 9-25, 9-48
AAIM bit, 9-25, 9-47
AAIS bit, 9-25, 9-47
AAn bit, 9-76
$\overline{\text{ABE[1:0]}}$ pins, 6-18
ABO bit, 9-43
abort acknowledge interrupt, CAN, 9-25
abort acknowledge register 1 (CAN_AA1), 9-76
abort acknowledge register 2 (CAN_AA2), 9-76
aborted frames, MAC, 8-16
aborts, DMA, 5-32
acceptance mask filtering, CAN, 9-16
acceptance mask register (CAN_AMxxH), 9-49
acceptance mask register (CAN_AMxxL), 9-51
access denied interrupt, CAN, 9-24
accesses
  off-core, 2-5
  to internal memory, 3-2
access to unimplemented address interrupt, CAN, 9-25
access way/ instruction address bit 11 bit, 3-10
ACKE bit, 9-85
active descriptor queue, and DMA synchronization, 5-64
active low/high frame syncs, serial port, 12-33
active mode, 1-23, 20-9
active mode to full on mode example, 20-29
ACTIVE_PLLDISABLED bit, 20-26
ACTIVE_PLLENABLED bit, 20-26
active video only mode, PPI, 7-10, 7-11
ADCs, connecting to, 12-1
address bus, 6-8
address filter evaluation, MAC, 8-14
address mapping, SDRAM, 6-27
ADIF bit, 9-24, 9-48
ADIM bit, 9-24, 9-47
ADIS bit, 9-24, 9-47
alarm clock, RTC, 18-2
alarm event flag bit, 18-21
alarm interrupt enable bit, 18-20
A-law companding, 12-24, 12-29
AlignmentErrors register, 8-55
alternate frame sync mode, 12-36
alternate timing, serial port, 12-35
AMBEN[2:0] field, 6-19, 6-21

AMC, 1-6, 1-9
bus contention, 6-11
EBIU block diagram, 6-4
features, 6-9
signals, 6-10
timing parameters, 6-20
AMCKEN bit, 6-19, 6-21
AME bit, 9-53
AMIDE bit, 9-49
AMS, 6-10
ANAK bit, 11-14, 11-38
application data, loading, 19-1
arbitration
DAB, 2-8, 2-9
DCB, 2-8, 2-9
DEB, 2-8, 2-9
EAB, 2-11
latency, 2-10
TWI, 11-7
architecture, memory, 3-1
ARDY pin, 6-12, 6-16
array access bit, 3-10
ASIC/FPGA designs, 6-1
ASTP bit, 8-17, 8-64, 8-70
asynchronous controller, 1-9
asynchronous FIFO connection, 5-42
asynchronous interfaces supported, 6-1
asynchronous memory, 3-2, 6-2, 6-9
asynchronous memory bank address range (table), 6-10
asynchronous memory bank control 0 register (EBIU_AMBCTL0), 6-22
asynchronous memory bank control 1 register (EBIU_AMBCTL1), 6-23
asynchronous memory bank control registers (EBIU_AMBCTLx), 6-21
asynchronous memory controller, *See* AMC
asynchronous memory global control register (EBIU_AMGCTL), 6-18
asynchronous read, 6-13
asynchronous serial communications, 13-4
asynchronous write, 6-14
ASYNC memory banks, 6-3
autobaud, and general-purpose timers, 15-34
autobaud detection, 13-4, 13-13, 15-34
autobuffer mode, 5-14, 5-31, 5-76
auto-refresh
command, 6-35, 6-52
timing, 6-60
auto-transmit mode, CAN, 9-15
avoiding bus contention, 6-11

B

B0HT[1:0] field, 6-22
B0RAT[3:0] field, 6-22
B0RDYEN bit, 6-22
B0RDYPOL bit, 6-22
B0ST[1:0] field, 6-22
B0TT[1:0] field, 6-22
B0WAT[3:0] field, 6-22
B1HT[1:0] field, 6-22
B1RAT[3:0] field, 6-22
B1RDYEN bit, 6-22
B1RDYPOL bit, 6-22
B1ST[1:0] field, 6-22
B1TT[1:0] field, 6-22
B1WAT[3:0] field, 6-22
B2HT[1:0] field, 6-23
B2RAT[3:0] field, 6-23
B2RDYEN bit, 6-23
B2RDYPOL bit, 6-23
B2ST[1:0] field, 6-23
B2TT[1:0] field, 6-23
B2WAT[3:0] field, 6-23
B3HT[1:0] field, 6-23
B3RAT[3:0] field, 6-23
B3RDYEN bit, 6-23
B3RDYPOL bit, 6-23
B3ST[1:0] field, 6-23

B3TT[1:0] field, 6-23
B3WAT[3:0] field, 6-23
BA[1:0] pins, 6-33
bandwidth, and memory DMA operations, 5-51
bank activate command, 6-35, G-1
bank activation command, 6-51
bank address, 6-64
bank size, 6-26, 6-64
bank size encodings (table), 6-28
bank width, 6-26
BASEID[10:0] field, 9-49, 9-53
baud rate
  SPI, 10-20
  UART, 13-6, 13-12
baud rate[15:0] field, 10-41
BCINIT[15:0] field, 5-101
BCOUNT[15:0] field, 5-102
BDI bit, 5-100
BDIE bit, 5-44, 5-100
BEF bit, 9-85
$\overline{\text{BGH}}$ pin, 6-8
$\overline{\text{BG}}$ signal, 6-8
BGSTAT bit, 6-8, 6-78, 6-79
BI bit, 13-24, 13-25
binary decode, B-4
bin x bit, 8-74, 8-75
bit order, selecting, 12-28
BKPRSEN bit, 8-79
Blackfin Loader File Viewer, 19-59
Blackfin processor family
  I/O memory space, 1-7
  memory architecture, 1-4
block, DMA, 5-11
block count, DMA, 5-40
block diagrams
  bus hierarchy, 2-3
  CAN, 9-3
  core, 2-5
block diagrams *(continued)*
  core timer, 16-2
  DMA controller, 5-5
  EBIU, 6-4
  general-purpose timers, 15-3
  interrupt processing, 4-16
  MAC, 8-3
  PLL, 20-3
  PPI, 7-2
  processor, 1-3, 1-4
  RTC, 18-3
  SDRAM, 6-58, 6-59
  SPI, 10-3
  SPORT, 12-6
  TWI, 11-2
  UART, 13-3
  watchdog timer, 17-2
block done interrupt, DMA, 5-44
block transfers, DMA, 5-41
BMODE[2:0] field, 19-5
BMODE[2:0] pins, 19-4
BMODE pins, 19-1
BOIF bit, 9-25, 9-48
BOIM bit, 9-25, 9-47
BOIS bit, 9-25, 9-47
BOLMT[1:0] field, 8-64, 8-66
booting, 19-1 to 19-60
  16-bit flash boot mode, 19-40
  boot block flag processing, 19-18
  booting a different application, 19-27
  boot ROM functions, 19-27
  boot stream, 19-11
  bypass mode, 19-34
  control bits, 19-14
  determining boot stream start addresses, 19-29
  from 8-bit flash, 19-36
  header, 19-13
  host boot scenarios, 19-12
  IGNORE block, 19-27

booting *(continued)*
initialization code, 19-22
Intel hex loader file, 19-38
loader file utility, 19-59
memory locations, 19-12
multi-DXE, 19-26
PORTF_FER, 19-20
PORT_MUX, 19-20
slave boot modes, 19-19
SPI master boot from flash, 19-28
SPI master mode, 19-42
SPI slave mode, 19-48
TWI master boot from flash, 19-29
TWI master mode, 19-53
TWI slave mode, 19-55
UART slave mode, 19-56
boot kernel, 19-1
boot modes, 1-25
boot ROM
internal, 19-1
memory space, 3-8
_BOOTROM_Boot_DXE_SPI function, 19-31
_BOOTROM_Get_DXE_Address_SPI function, 19-31
boot stream, 19-1, 19-11
boundary-scan architecture, B-2
boundary-scan register, B-6
BroadcastFramesReceivedOK register, 8-56
BroadcastFramesXmittedOK register, 8-62
broadcast mode, 10-10, 10-17, 10-18
BRP[9:0] field, 9-10, 9-45
$\overline{\text{BR}}$ signal, 6-8
buffer registers, timers, 15-47
BUFRDERR bit, 11-13, 11-38
BUFWRERR bit, 11-13, 11-38
burst length, 6-33, G-2
burst type, 6-34, G-2
bus agents
DAB, 2-9
PAB, 2-6
BUSBUSY bit, 11-12, 11-38
bus contention, avoiding, 6-11, 21-7
bus cycles
asynchronous read, 6-13
asynchronous write, 6-14
bus error, EBIU, 6-7
buses
See also DAB, DCB, DEB, EAB, EPB, PAB
bandwidth, 1-2
core, 2-4
and DMA, 5-45
hierarchy, 2-3
on-chip, 2-1
PAB, 2-6
peripheral, 2-6
and peripherals, 1-2
prioritization and DMA, 5-53
busmastership, granting to external SDC, 6-46
bus-off interrupt, CAN, 9-25
bus request and grant, 6-8
bus standard, I^2C, 1-11
BYPASS bit, 20-26
bypass capacitor placement, 21-11
BYPASS instruction, B-6
bypass mode, 19-34
bypass register, B-6
byte address, 6-64
byte enables, 6-18
byte enable x bit, 8-85, 8-86, 8-87, 8-88

C

CAN, 1-12, 9-1 to 9-93
acceptance mask filtering, 9-16
acceptance mask registers, 9-5
acknowledge error, 9-29
architecture, 9-4
autobaud detection, 15-34
auto-transmit mode, 9-15
bit error, 9-28
bit rate detection, 15-5
bit timing, 9-9
block diagram, 9-3
bus interface, 9-2
clock, 9-9
code examples, 9-87
configuration mode, 9-9, 9-12
CRC error, 9-29
data field filtering, 9-18
debug and test modes, 9-33
disabling mailboxes, 9-21
enabling mailboxes, 9-89
error frames, 9-26, 9-29
error levels, 9-31
errors, 9-28
event counter, 9-26
extended frame, 9-9
features, 9-1
form error, 9-28
global interupts, 9-24
hibernate state, 9-38
identifier frame, 9-8
initializing, 9-87
initializing mailboxes, 9-89
initiating transfers, 9-91
interrupts, 9-23, 9-91
lost arbitration, 9-27
and low power designs, 9-38
low power features, 9-37
mailbox area registers, 9-5
mailbox control, 9-6

CAN *(continued)*
mailboxes, 9-4
mailbox interrupts, 9-23
mailbox RAM, 9-4
message buffers, 9-4
message received, 9-27
message stored, 9-27
multiplexing of signals, 9-2
nominal bit rate, 9-11
nominal bit time, 9-10
overload frame, 9-26
pins, 14-3
port J, 14-8
propagation segment, 9-11
protocol basics, 9-7
receive message lost, 9-27
receive message rejected, 9-27
receive operation, 9-15
receive operation flow chart, 9-17
registers, table, 9-39
remote frames, 9-19
resynchronization, 9-11
retransmission, 9-13
sampling, 9-11
single shot transmission, 9-15
sleep mode, 9-38
software reset, 9-12
standard frame, 9-8
stuff error, 9-29
suspend mode, 9-37
test modes, 9-36
time quantum, 9-9
time stamps, 9-20
transceiver interconnection, 9-2
transmission, 9-8
transmission aborted, 9-27
transmission succeeded, 9-27
transmit operation, 9-12
transmit operation flow chart, 9-14
universal counter as event counter, 9-26

CAN *(continued)*
valid message, 9-27
wakeup from hibernate, 9-38
warnings, 9-28
watchdog mode, 9-19
CAN_AA1 (abort acknowledge register 1), 9-76
CAN_AA2 (abort acknowledge register 2), 9-76
CAN_AAx registers, 9-41
CAN_AMxxH (acceptance mask register), 9-40, 9-49
CAN_AMxxL (acceptance mask register), 9-40, 9-51
CAN_CEC (CAN error counter register), 9-35, 9-42, 9-85
CAN_CLOCK (CAN clock register), 9-10, 9-39, 9-45
CAN clock register (CAN_CLOCK), 9-45
CAN_CONTROL (master control register), 9-39, 9-43
CAN_DEBUG (CAN debug register), 9-33, 9-34, 9-39, 9-45
CAN debug register (CAN_DEBUG), 9-45
CAN error counter register (CAN_CEC), 9-85
CAN error counter warning level register (CAN_EWR), 9-86
CAN_ESR (error status register), 9-42, 9-85
CAN_EWR (CAN error counter warning level register), 9-42, 9-86
CAN_GIF (global interrupt flag register), 9-40, 9-48
CAN_GIM (global interrupt mask register), 9-40, 9-47
CAN_GIS (global interrupt status register), 9-40, 9-47
CAN interrupt register (CAN_INTR), 9-46
CAN_INTR (CAN interrupt register), 9-40, 9-46
CAN_MBIM1 (mailbox interrupt mask register 1), 9-79
CAN_MBIM2 (mailbox interrupt mask register 2), 9-80
CAN_MBIMx registers, 9-41
CAN_MBRIF1 (mailbox receive interrupt flag register 1), 9-81
CAN_MBRIF2 (mailbox receive interrupt flag register 2), 9-82
CAN_MBRIFx registers, 9-42
CAN_MBTD (temporary mailbox disable feature register), 9-41, 9-78
CAN_MBTIF1 (mailbox transmit interrupt flag register 1), 9-80
CAN_MBTIF2 (mailbox transmit interrupt flag register 2), 9-81
CAN_MBTIFx registers, 9-42
CAN_MBxx_DATA0 (mailbox word 0 register), 9-40, 9-67
CAN_MBxx_DATA1 (mailbox word 1 register), 9-40, 9-65
CAN_MBxx_DATA2 (mailbox word 2 register), 9-40, 9-63
CAN_MBxx_DATA3 (mailbox word 3 register), 9-40, 9-61
CAN_MBxx_DATA registers, 9-5
CAN_MBxx_ID0 (mailbox word 6 register), 9-5, 9-40, 9-55
CAN_MBxx_ID1 (mailbox word 7 register), 9-5, 9-40, 9-53
CAN_MBxx_LENGTH (mailbox word 4 register), 9-5, 9-40, 9-59
CAN_MBxx_TIMESTAMP (mailbox word 5 register), 9-5, 9-40, 9-57
CAN_MC1 (mailbox configuration register 1), 9-69

CAN_MC2 (mailbox configuration register 2), 9-69
CAN_MCx registers, 9-41
CAN_MD1 (mailbox direction register 1), 9-70
CAN_MD2 (mailbox direction register 2), 9-70
CAN_MDx (mailbox direction registers), 9-41
CAN_OPSS1 (overwrite protection/single shot transmission register 1), 9-73
CAN_OPSS2 (overwrite protection/single shot transmission register 2), 9-73
CAN_OPSSx (overwrite protection/single shot transmission registers), 9-41
CAN_RFH1 (remote frame handling register 1), 9-78
CAN_RFH2 (remote frame handling register 2), 9-79
CAN_RFHx registers, 9-41
CAN_RML1 (receive message lost register 1), 9-72
CAN_RML2 (receive message lost register 2), 9-72
CAN_RMLx registers, 9-41
CAN_RMP1 (receive message pending register 1), 9-71
CAN_RMP2 (receive message pending register 2), 9-71
CAN_RMPx registers, 9-41
CANRX bit, 9-46
CANRX pin, 9-7
CAN_STATUS (global status register), 9-39, 9-44
CAN_TA1 (transmission acknowledge register 1), 9-77
CAN_TA2 (transmission acknowledge register 2), 9-77
CAN_TAx registers, 9-41
CAN_TIMING (CAN timing register), 9-10, 9-40, 9-46
CAN timing register (CAN_TIMING), 9-46
CAN_TRR1 (transmission request reset register 1), 9-75
CAN_TRR2 (transmission request reset register 2), 9-75
CAN_TRRx registers, 9-41
CAN_TRS1 (transmission request set register 1), 9-74
CAN_TRS2 (transmission request set register 2), 9-74
CAN_TRSx registers, 9-41
CANTX bit, 9-46
CANTX pin, 9-7
CAN_UCCNF (universal counter configuration mode register), 9-42, 9-83
CAN_UCCNT (universal counter register), 9-42, 9-84
CAN_UCRC (universal counter reload/capture register), 9-42, 9-84
CANWE bit, 20-27
capacitors, 21-10
capture mode, *See* WDTH_CAP mode
CAPWKFRM bit, 8-82, 8-83
CarrierSenseErrors register, 8-62
CAS latency, 6-34, 6-37
CAW, 6-64
CCA bit, 9-44
CCIR-656, *See* ITU-R 656
CCITT G.711 specification, 12-29
CCLK (core clock), 2-4, 20-5
  disabling, 20-22
  status by operating mode, 20-8
CCOR bit, 8-44, 8-125
CCR bit, 9-43
CDDBG bit, 6-46, 6-68, 6-77
CDE bit, 9-33, 9-45

CDPRIO bit, 6-19, 6-21, 6-44
channels
  defined, serial, 12-23
  serial port TDM, 12-23
  serial select offset, 12-23
CHNL[9:0] field, 12-67, 12-68
CIF, 7-9
circuit board testing, B-1, B-5
circular addressing, 5-61
CKELOW bit, 20-27
CL, 6-37
CL[1:0] field, 6-68, 6-69
clearing interrupt requests, 4-23
clear Pxn bit, 14-25
clear Pxn interrupt A enable bit, 14-31
clear Pxn interrupt B enable bit, 14-32
CLKBUFOE bit, 20-27, 20-28
CLKBUF pin, 8-4, 8-47
CLKHI[7:0] field, 11-30
CLKIN, 1-22, 20-1
CLKIN (input clock), 2-4, 20-3
CLKIN to VCO, changing the multiplier, 20-16
CLKIN to VCO, changing the multiplier, example, 20-31
CLKLOW[7:0] field, 11-30
CLKOUT pin, 6-7, 6-69
  disabling, 6-74
CLK_SEL bit, 15-13, 15-22, 15-44, 15-51
clock
  clock signals, 1-22
  EBIU, 6-2
  frequency for SPORT, 12-64
  internal, 2-4
  MAC, 8-4
  managing, 21-1
  RTC, 18-4
  source for general-purpose timers, 15-5
  SPI clock signal, 10-4
  types, 21-1
clock divide modulus register, 12-64
clock domain synchronization, PPI, 7-16
clock input (CLKIN) pin, 21-1
clock phase, SPI, 10-14, 10-15
clock polarity, SPI, 10-14
clock rate
  core timer, 16-1
  SPORT, 12-2
clock ratio, changing, 20-6
codecs, connecting to, 12-1
column address, 6-64
  strobe latency, 6-34, G-4
column read/write, SDRAM, 6-32
command inhibit command, 6-54
commands
  auto-refresh, 6-35, 6-52, 6-60
  bank activate, 6-35, G-1
  bank activation, 6-51
  command inhibit, 6-54
  DMA control, 5-34, 5-35
  EMRS, 6-50
  ERMS, 6-34
  MRS, 6-34, 6-49
  no operation, 6-54
  precharge, 6-35, G-18
  precharge all, 6-35, 6-52
  read, 6-35
  read/write, 6-51
  SDC, 6-47
  self-refresh, 6-53
  single precharge, 6-52
  transfer initiate, 10-24, 10-25
  write, 6-35
  write with data mask, 6-51
companding, 12-16, 12-24
  defined, 12-29
  lengths supported, 12-29
  multichannel operations, 12-24

configuration
CAN, 9-12
regulator wakeups, 20-32
SDC, 6-56
SDRAM, 6-25
SPORT, 12-11
congestion, on DMA channels, 5-50
contention, bus, avoiding, 6-11
continuous polling, and MMC register, 8-44
continuous transition, DMA, 5-30
control bit summary, general-purpose timers, 15-50
control byte sequences, PPI, 7-9
control frames, MAC, 8-16
controller area network, *See* CAN
control register
data memory, 3-9
EBIU, 6-6
conventions, -lv
core
block diagram, 2-5
core bus, 2-4
core clock (CCLK), 20-5, 21-2
core clock/system clock ratio control, 20-5
powering down, 20-22
waking from idle state, 4-9
core and system reset, code example, 19-8
core clock, *See* CCLK
core event controller (CEC), 4-1, 4-2
core-only software reset, 19-3, 19-7
core timer, 16-1 to 16-8
block diagram, 16-2
clock rate, 16-1
features, 16-1
initialization, 16-2
internal interfaces, 16-2
interrupts, 16-3
core timer *(continued)*
low power mode, 16-3
operation, 16-2
registers, 16-4
scaling, 16-7
core timer control register (TCNTL), 16-3, 16-4
core timer count register (TCOUNT), 16-5
core timer period register (TPERIOD), 16-6
core timer scale register (TSCALE), 16-7
core voltage, changing, 20-33
counter, RTC, 18-1
count value[15:0] field, 16-5
count value[31:16] field, 16-5
CPHA bit, 10-41
CPOL bit, 10-41
CRC-16 hash value calculation, 8-38
CRC-32 calculation, MAC, 8-73
CRCE bit, 9-85
CRC state, MAC, 8-37
CROLL bit, 8-44, 8-124, 8-125
CrossCore software, 1-29
crosstalk, 21-9
CSA bit, 9-37, 9-44
CSEL[1:0] field, 20-5, 20-25
CSR bit, 9-37, 9-43
CTYPE bit, 5-71
current address field, 5-84
current address registers
(DMAx_CURR_ADDR), 5-83
(MDMA_yy_CURR_ADDR), 5-83
current descriptor pointer field, 5-97
current descriptor pointer registers
(DMAx_CURR_DESC_PTR), 5-96
(MDMA_yy_CURR_DESC_PTR), 5-96

current inner loop count registers
(DMAx_CURR_X_COUNT), 5-86, 5-87
(MDMA_yy_CURR_X_COUNT), 5-86, 5-87
current outer loop count registers
(DMAx_CURR_Y_COUNT), 5-91
(MDMA_yy_CURR_Y_COUNT), 5-91
CURR_X_COUNT[15:0] field, 5-87
CURR_Y_COUNT[15:0] field, 5-92
customer support, -xlvii

D

DAB, 2-8, 5-7, 5-45, 5-106
arbitration, 2-8, 2-9
bus agents (masters), 2-9
clocking, 20-2
latencies, 2-10
performance, 2-10
throughput, 2-10
DAB_TRAFFIC_COUNT[2:0] field, 5-106
data bank access bit, 3-10
data bus, 6-8
data cache select/address bit 14 bit, 3-10
data corruption, avoiding with SPI, 10-17
data-driven interrupts, 5-80
data field byte 0[7:0] field, 9-61
data field byte 1[7:0] field, 9-61
data field byte 2[7:0] field, 9-63
data field byte 3[7:0] field, 9-63
data field byte 4[7:0] field, 9-65
data field byte 5[7:0] field, 9-65
data field byte 6[7:0] field, 9-67
data field byte 7[7:0] field, 9-67
data field filtering, CAN, 9-18
data formats, SPORT, 12-28
data input modes for PPI, 7-15 to 7-17
data/instruction access bit, 3-10
data I/O mask function, 6-34
data masks, 6-41
data memory control register (DMEM_CONTROL), 3-9
data move, serial port operations, 12-38
data output modes for PPI, 7-18 to 7-19
data sampling, serial, 12-33
data test command register (DTEST_COMMAND), 3-10
data transfers
DMA, 2-10, 5-1
SPI, 10-17
data word, serial data formats, 12-57
day[14:0] field, 18-22
day alarm event flag bit, 18-21
day alarm interrupt enable bit, 18-20
day counter[14:0] field, 18-20
DBF bit, 8-64, 8-68
DCB, 2-8, 5-6, 5-45, 5-107
arbitration, 2-8, 2-9
DC bit, 8-64, 8-67
DCBS bit, 3-9
DCB_TRAFFIC_COUNT field, 5-107
DCB_TRAFFIC_PERIOD field, 5-107
DCNT[7:0] field, 11-34, 11-35
DEB, 2-8, 5-7, 5-45, 5-107
arbitration, 2-8, 2-9
and EBIU, 6-5
frequency, 2-11
performance, 2-11
DEB_TRAFFIC_COUNT field, 5-107
DEB_TRAFFIC_PERIOD field, 5-107
debugging, 1-29
test point access, 21-12
DEC bit, 9-35, 9-45
deep sleep mode, 1-24, 6-46, 20-10
default mapping, peripheral to DMA, 5-7
delay count register (PPI_DELAY), 7-34
descriptor array mode, DMA, 5-18, 5-76
descriptor-based DMA, 5-16

descriptor chains, DMA, 5-30
descriptor list mode, DMA, 5-17, 5-76
descriptor pairs, DMA, 8-12
descriptor queue, 5-62
  management, 5-61
  synchronization, 5-62
descriptor structures
  alternative, 8-26
  DMA, 5-60
  MDMA, 5-67
destination channels, memory DMA, 5-9
development tools, 1-29
DF bit, 20-3, 20-4, 20-26
DFC[15:0] field, 9-55
DFETCH bit, 5-17, 5-24, 5-79
DFM[15:0] field, 9-51
DI_EN bit, 5-17, 5-74, 5-77
DIL bit, 9-35, 9-45
direction errors, MAC, 8-30
direct memory access, *See* DMA
disabling
  general-purpose timers, 15-39
  PLL, 20-13
discarded frames, MAC, 8-16
DI_SEL bit, 5-74, 5-77
DITFS bit, 12-37, 12-48, 12-51, 12-63
divisor latch high byte[15:8] field, 13-30
divisor latch low byte[7:0] field, 13-30
divisor reset, UART, 13-30
DLAB bit, 13-21, 13-26, 13-27
DLC[3:0] field, 9-59
DLEN[2:0] field, 7-26, 7-27
DMA, 1-7, 5-1 to 5-118
  1D interrupt-driven, 5-58
  1D unsynchronized FIFO, 5-60
  2D, polled, 5-59
  2D array, example, 5-108
  2D interrupt-driven, 5-59
  autobuffer mode, 5-14, 5-31, 5-76

DMA *(continued)*
  bandwidth, 5-50
  block count, 5-40
  block diagram, 5-5
  block done interrupt, 5-44
  block transfers, 5-11, 5-41
  buffer size, multichannel SPORT, 12-25
  buses, 2-8
  channel registers, 5-70
  channels, 5-45
  channels and control schemes, 5-55
  channel-specific register names, 5-69
  congestion, 5-50
  connecting asynchronous FIFO, 5-42
  continuous transfers using autobuffering, 5-58
  continuous transition, 5-30
  control command restrictions, 5-38
  control commands, 5-34, 5-35
  data transfers, 5-2
  default peripheral mapping, 5-7
  descriptor array, 5-25
  descriptor array mode, 5-18, 5-76
  descriptor-based, 5-16
  descriptor-based, initializing, 5-111
  descriptor-based vs. register-based transfers, 5-2
  descriptor chains, 5-30
  descriptor element offsets, 5-19
  descriptor list mode, 5-17, 5-76
  descriptor lists, 5-25
  descriptor queue, 5-61, 5-62
  descriptors, and MAC, 8-127
  descriptors, recommended size, 5-19
  descriptor structures, 5-60
  direction, 5-78
  DMA error interrupt, 5-81
  double buffer scheme, 5-59
  and EBIU, 5-6
  errors, 5-32, 5-33

DMA *(continued)*
example connection, receive, 5-43
example connection, transmit, 5-42
external interfaces, 5-6
finish control command, 5-36
first data memory access, 5-24
flow chart, 5-21, 5-22
FLOW mode, 5-19
FLOW value, 5-23
for SPI transmit, 10-12
functions, summary, 5-3
handshake DMA, 1-8
handshake operation, 5-40
header file to define descriptor structures example, 5-112
HMDMA1 block enable example, 5-117
HMDMA pins, 14-2
HMDMA with delayed processing example, 5-118
initializing, 5-20
internal interfaces, 5-6
and L1 memory, 5-6
large model mode, 5-76
latency, 5-27
linked list, 8-127
MAC configuration, 8-126
mapping to peripherals, 4-10
memory conflict, 5-53
memory DMA, 1-8, 5-9
memory DMA streams, 5-10
memory DMA transfers, 5-7
memory read, 5-28
operation flow, 5-20
orphan access, 5-31
overflow interrupt, 5-44
overview, 1-7
performance considerations, 5-46
peripheral, 5-7
peripheral channels, 5-2
peripheral channels priority, 5-8
peripheral interrupts, 4-9

DMA *(continued)*
peripheral priority and default mapping, 5-49
pipelining requests, 5-41
polling DMA status example, 5-110
polling registers, 5-56
and PPI, 7-37
prioritization and traffic control, 5-48 to 5-55
programming examples, 5-107 to 5-118
receive, 5-30
receive restart or finish, 5-39
refresh, 5-25
register-based, 5-12
register-based 2D memory DMA example, 5-108
register naming conventions, 5-70
remapping peripheral assignment, 5-8
request data control command, 5-37
request data urgent control command, 5-38
restart control command, 5-36
round robin operation, 5-52
serial port block transfers, 12-38
single-buffer transfers, 5-58
small model mode, 5-76
software management, 5-55
software-triggered descriptor fetch example, 5-114
and SPI, 10-12
SPI data transmission, 10-13, 10-14
SPI master, 10-30
SPI slave, 10-32
and SPORT, 12-3
startup, 5-20
stop mode, 5-13, 5-75
stopping transfers, 5-31
support for peripherals, 1-2
switching peripherals from, 5-81
and synchronization with PPI, 7-14

DMA *(continued)*
synchronization, 5-55 to 5-65
synchronized transition, 5-30
termination without abort, 5-31
throughput, 5-45
traffic control, 5-53
traffic exceeding available bandwidth, 5-50
transfers, urgent, 5-50
transmit, 5-28
transmit operation, MAC, 8-24
transmit restart or finish, 5-38
triggering transfers, 5-65
two descriptors in small list flow mode, example, 5-111
two-dimensional, 5-14
two-dimensional memory DMA setup example, 5-109
types supported, 1-8
and UART, 13-17, 13-27
using descriptor structures example, 5-113
variable descriptor size, 5-18
with PPI, 7-23
word size, changing, 5-30, 5-31
work units, 5-17, 5-25, 5-27
DMA2D bit, 5-74, 5-77
DMA bus, *See* DAB
DMACFG field, 5-24, 5-67
DMA channel registers, 5-68
DMA configuration registers
(DMAx_CONFIG), 5-74
(MDMA_yy_CONFIG), 5-74
DMA controller, 5-2
DMA core bus, *See* DCB
DMA descriptor pairs, 8-24
DMA_DONE bit, 5-79
DMA_DONE interrupt, 5-78
DMAEN bit, 5-21, 5-66, 5-74, 5-78
DMA_ERR bit, 5-79
DMA_ERROR interrupt, 5-32
DMA error interrupts, 5-80
DMA external bus, *See* DEB
DMA performance optimization, 5-44
DMA queue completion interrupt, 5-64
DMAR0 pin, 5-6
DMAR1 pin, 5-6
DMA registers, 5-67, 5-68
DMA_RUN bit, 5-24, 5-63, 5-66, 5-79
DMARx pin, 5-41
DMA start address field, 5-82
DMA_TC_CNT (DMA traffic control counter register), 5-106
DMA_TC_PER (DMA traffic control counter period register), 5-52, 5-106
DMA traffic control counter period register (DMA_TC_PER), 5-106
DMA traffic control counter register (DMA_TC_CNT), 5-106
DMA_TRAFFIC_PERIOD field, 5-106
DMAx_CONFIG (DMA configuration registers), 5-11, 5-20, 5-28, 5-74
DMAx_CURR_ADDR (current address registers), 5-83
DMAx_CURR_DESC_PTR (current descriptor pointer registers), 5-96
DMAx_CURR_X_COUNT (current inner loop count registers), 5-86, 5-87
DMAx_CURR_Y_COUNT (current outer loop count registers), 5-91
DMAx_IRQ_STATUS (interrupt status registers), 5-78, 5-79
DMAx_NEXT_DESC_PTR (next descriptor pointer registers), 5-20, 5-94
DMAx_PERIPHERAL_MAP (peripheral map registers), 5-71
DMAx_START_ADDR (start address registers), 5-20, 5-82

DMAx_X_COUNT (inner loop count registers), 5-85
DMAx_X_MODIFY (inner loop address increment registers), 5-20, 5-88
DMAx_Y_COUNT (outer loop count registers), 5-90
DMAx_Y_MODIFY (outer loop address increment registers), 5-20, 5-93
DMC[1:0] field, 3-9
DMEM_CONTROL (data memory control register), 3-8, 3-9
DNAK bit, 11-14, 11-38
DNM bit, 9-43
DOUBLE_FAULT bit, 19-7
double word index[1:0] field, 3-10
DPMC, 20-2, 20-7 to 20-34
DR bit, 13-11, 13-24
DR flag, 13-16
DRI bit, 9-35, 9-45
DRO bit, 8-64
DRQ[1:0] field, 5-51, 5-99, 5-100
DRTY bit, 8-64, 8-66
DRxPRI signal, 12-5
DRxPRI SPORT input, 12-6
DRxSEC signal, 12-5
DRxSEC SPORT input, 12-6
DTEST_COMMAND (data test command register), 3-10
DTO bit, 9-35, 9-45
DTXCRC bit, 8-64, 8-67
DTXPAD bit, 8-64, 8-67
DTxPRI signal, 12-5
DTxPRI SPORT output, 12-6
DTxSEC signal, 12-5
DTxSEC SPORT output, 12-6
dynamic power management, 1-23, 20-1 to 20-34
dynamic power management controller, 20-2

E

EAB
  arbitration, 2-11
  clocking, 20-2
  and EBIU, 6-5
  frequency, 2-11
  performance, 2-11
early frame sync, 12-35
EAV signal, 7-7
EBC, 6-5
EBCAW[1:0] field, 6-63
EBE bit, 6-56, 6-62, 6-63
EBIU, 1-9, 6-1 to 6-83
  as slave, 6-5
  asynchronous interfaces supported, 6-1
  block diagram, 6-4
  bus errors, 6-7
  byte enables, 6-18
  clock, 6-2
  clocking, 20-2
  control registers, 6-6
  core transfers to SRAM, 6-24
  and DMA, 5-6
  error detection, 6-7
  external access extension, 6-16
  overview, 6-1
  programmable timing characteristics, 6-12
  read access period, 6-16
  request priority, 6-1
  shared pins, 6-6
  status register, 6-6
  system clock, 6-7
  wait states, 6-16
EBIU_AMBCTL0 (asynchronous memory bank control 0 register), 6-21, 6-22
EBIU_AMBCTL1 (asynchronous memory bank control 1 register), 6-21, 6-23
EBIU_AMGCTL (asynchronous memory global control register), 6-21

EBIU_SDBCTL (SDRAM memory bank control register), 6-62, 6-63
EBIU_SDGCTL (SDRAM memory bank control register), 6-57
EBIU_SDGCTL (SDRAM memory global control register), 6-67, 6-68
EBIU_SDRRC (SDRAM refresh rate control register), 6-60
EBIU_SDSTAT (SDRAM control status register), 6-78
EBO bit, 9-44
EBSZ[2:0] field, 6-27, 6-62, 6-63
EBUFE bit, 6-59, 6-68, 6-75, 6-76
ECINIT[15:0] field, 5-103
ECOUNT[15:0] field, 5-103
edge detection, GPIO, 14-15
elfloader.exe, 19-11
ELSI bit, 13-7, 13-28
EMAC_ADDRHI (MAC address high register), 8-52, 8-71
EMAC_ADDRLO (MAC address low register), 8-52, 8-70
EMAC_FLC (MAC flow control register), 8-52, 8-78
EMAC_HASHHI (EMAC multicast hash table high register), 8-52, 8-72
EMAC_HASHLO (EMAC multicast hash table low register), 8-52, 8-72
EMAC_MMC_CTL (MAC management counters control register), 8-54, 8-124
EMAC_MMC_RIRQE (EMAC MMC RX interrupt enable register), 8-54, 8-117
EMAC_MMC_RIRQS (EMAC MMC RX interrupt status register), 8-54, 8-115
EMAC MMC RX interrupt enable register (EMAC_MMC_RIRQE), 8-117
EMAC MMC RX interrupt status register (EMAC_MMC_RIRQS), 8-115
EMAC_MMC_TIRQE (EMAC MMC TX interrupt enable register), 8-54, 8-121
EMAC_MMC_TIRQS (EMAC MMC TX interrupt status register), 8-54, 8-119
EMAC MMC TX interrupt enable register (EMAC_MMC_TIRQE), 8-121
EMAC MMC TX interrupt status register (EMAC_MMC_TIRQS), 8-119
EMAC multicast hash table high register (EMAC_HASHHI), 8-72
EMAC multicast hash table low register (EMAC_HASHLO), 8-72
EMAC_OPMODE (MAC operating mode register), 8-52, 8-64
EMAC_RXC_ALIGN register, 8-55
EMAC_RXC_ALLFRM register, 8-59
EMAC_RXC_ALLOCT register, 8-59
EMAC_RXC_BROAD register, 8-56
EMAC_RXC_DMAOVF register, 8-55
EMAC_RXC_EQ64 register, 8-59
EMAC_RXC_FCS register, 8-55
EMAC_RXC_GE1024 register, 8-60
EMAC_RXC_LNERRI register, 8-57
EMAC_RXC_LNERRO register, 8-57
EMAC_RXC_LONG register, 8-57
EMAC_RXC_LT1024 register, 8-60
EMAC_RXC_LT128 register, 8-59
EMAC_RXC_LT256 register, 8-59
EMAC_RXC_LT512 register, 8-60
EMAC_RXC_MACCTL register, 8-58
EMAC_RXC_MULTI register, 8-56
EMAC_RXC_OCTET register, 8-55
EMAC_RXC_OK register, 8-54
EMAC_RXC_OPCODE register, 8-58
EMAC_RXC_PAUSE register, 8-58
EMAC_RXC_SHORT register, 8-59
EMAC_RXC_TYPED register, 8-59
EMAC_RXC_UNICST register, 8-56

EMAC RX current frame status register (EMAC_RX_STAT), 8-97
EMAC RX frame status interrupt enable register (EMAC_RX_IRQE), 8-107
EMAC_RX_IRQE (EMAC RX frame status interrupt enable register), 8-53, 8-107
EMAC_RX_STAT (EMAC RX current frame status register), 8-53, 8-97
EMAC RX sticky frame status register (EMAC_RX_STKY), 8-103
EMAC_RX_STKY (EMAC RX sticky frame status register), 8-53, 8-103
EMAC_STAADD (MAC station management address register), 8-10, 8-52, 8-76
EMAC_STADAT (MAC station management data register), 8-10, 8-52, 8-78
EMAC_SYSCTL (MAC system control register), 8-10, 8-53, 8-93
EMAC_SYSTAT (MAC system status register), 8-53, 8-95
EMAC_TXC_1COL register, 8-60
EMAC_TXC_ALLFRM register, 8-63
EMAC_TXC_ALLOCT register, 8-63
EMAC_TXC_BROAD register, 8-62
EMAC_TXC_CRSERR register, 8-62
EMAC_TXC_DEFER register, 8-61
EMAC_TXC_DMAUND register, 8-62
EMAC_TXC_EQ64 register, 8-63
EMAC_TXC_GT1COL register, 8-60
EMAC_TXC_LATECL register, 8-61
EMAC_TXC_LT128 register, 8-63
EMAC_TXC_MACCTL register, 8-63
EMAC_TXC_MULTI register, 8-62
EMAC_TXC_OCTET register, 8-61
EMAC_TXC_OK register, 8-60
EMAC_TXC_UNICST register, 8-62
EMAC TX current frame status register (EMAC_TX_STAT), 8-108
EMAC_TXC_XS_COL register, 8-61
EMAC_TXC_XS_DFR register, 8-62
EMAC TX frame status interrupt enable register (EMAC_TX_IRQE), 8-114
EMAC_TX_IRQE (EMAC TX frame status interrupt enable register), 8-53, 8-114
EMAC_TX_STAT (EMAC TX current frame status register), 8-53, 8-108
EMAC TX sticky frame status register (EMAC_TX_STKY), 8-112
EMAC_TX_STKY (EMAC TX sticky frame status register), 8-53, 8-112
EMAC_VLAN1 (MAC VLAN1 tag register), 8-52, 8-80
EMAC_VLAN2 (MAC VLAN2 tag register), 8-52, 8-80
EMAC wakeup frame filter offsets register (EMAC_WKUP_FFOFF), 8-91
EMAC_WKUP_CTL (MAC wakeup frame control and status register), 8-52, 8-81
EMAC_WKUP_FFCMD (MAC wakeup frame filter commands register), 8-53, 8-89
EMAC_WKUP_FFCRC0 (MAC wakeup frame filter CRC0/1 register), 8-53, 8-92
EMAC_WKUP_FFCRC1 (MAC wakeup frame filter CRC2/3 register), 8-53, 8-92
EMAC_WKUP_FFMSKx (MAC wakeup frameX byte mask registers), 8-52, 8-84
EMAC_WKUP_FFOFF (EMAC wakeup frame filter offsets register), 8-53, 8-91
EMISO bit, 10-19, 10-41
EMREN bit, 6-50, 6-68, 6-76

EMRS command, 6-50
emulation, and timer counter, 15-45
EMU_RUN bit, 15-44, 15-51
enable interrupts (STI) instruction, 20-17
enable Pxn interrupt A bit, 14-28
enable Pxn interrupt B bit, 14-28
enable wakeup filter 0 bit, 8-89, 8-90
enable wakeup filter 1 bit, 8-89, 8-90
enable wakeup filter 2 bit, 8-89, 8-90
enable wakeup filter 3 bit, 8-89
enabling
  general-purpose timers, 15-38
  interrupts, 4-8
  PPI, 7-3
ENDCPLB bit, 3-9
entering hibernate state, 20-32
entire field mode, PPI, 7-10
entire field submode, 7-10
EP bit, 9-44
EPIF bit, 9-26, 9-48
EPIM bit, 9-26, 9-47
EPIS bit, 9-26, 9-47
EPROM, 1-6
EPS bit, 13-21
ERBFI bit, 13-7, 13-11, 13-27, 13-28
ERMS command, 6-34
ERR_DET bit, 7-30, 7-33
ERR_NCOR bit, 7-31, 7-33
error frames, CAN, 9-29
error-passive interrupt, CAN, 9-26
errors
  DMA, 5-32
  not detected by DMA hardware, 5-33
  startup, and timers, 15-11
error signals, SPI, 10-20 to 10-23
error status register (CAN_ESR), 9-85
error warning receive interrupt, CAN, 9-26
error warning transmit interrupt, CAN, 9-26
ERR_TYP[1:0] field, 15-10, 15-43, 15-44, 15-51
ETBEI bit, 13-6, 13-11, 13-18, 13-27, 13-28
Ethernet frame header, 8-50
Ethernet MAC, *See* MAC
event counter, CAN, 9-26
event handling, 4-2
events
  default mapping, 4-11
  definition, 4-3
  types of, 4-2
event vector table (EVT), 4-1
EVT1 register, 19-8
EWLREC[7:0] field, 9-86
EWLTEC[7:0] field, 9-86
EWRIF bit, 9-26, 9-48
EWRIM bit, 9-26, 9-47
EWRIS bit, 9-26, 9-47
EWTIF bit, 9-26, 9-48
EWTIM bit, 9-26, 9-47
EWTIS bit, 9-26, 9-47
EXT_CLK mode, 15-34 to 15-36, 15-47
  control bit and register usage, 15-50
  flow diagram, 15-35
extended mode register, initialization, 6-50
external access bus, *See* EAB
external bank, SDRAM, 6-33
external bus iInterface unit, *See* EBIU
external crystal, 1-22
external memory, 1-6, 3-8
  design issues, 21-5
  interfaces, 6-6
external memory interfaces, 6-6
external memory map, 3-2
  figure, 6-3
external PHY, 8-4
external SDRAM memory, 6-27
external trigger output interrupt, CAN, 9-24

EXTEST instruction, B-5
EXTID[15:0] field, 9-51, 9-55
EXTID[17:16] field, 9-49, 9-53
EXTIF bit, 9-24, 9-48
EXTIM bit, 9-24, 9-47
EXTIS bit, 9-24, 9-47
EZ-KIT Lite, 1-31

F

FAST bit, 11-34, 11-36
fast mode, TWI, 11-10
FBBRW bit, 6-68, 6-76
FDF bit, 9-18, 9-49
FDMODE bit, 8-64, 8-65
FE bit, 13-24, 13-25
FER bit, 9-85
FFE bit, 13-31, 13-32
field, 3-8
FIFO, 6-1
filtering, MAC, 8-13
FINAL bit, 19-14
finish control command, DMA, 5-36
flag word register, 19-14
flash interface, 21-5
flash memory, 1-6, 6-1
FLCBUSY bit, 8-79, 8-80
FLCE bit, 8-16, 8-79, 8-80
FLCPAUSE[15:0] field, 8-79
FLD bit, 7-31, 7-33
FLD_SEL bit, 7-5, 7-27, 7-30
flex descriptors, 5-3
FLGx bit, 10-42, 10-43
FLOW[2:0] field, 5-25, 5-27, 5-60, 5-74, 5-75
flow charts
  boot block flag processing, 19-18
  CAN receive operation, 9-17
  CAN transmit operation, 9-14
  DMA, 5-21, 5-22
flow charts *(continued)*
  general-purpose timers interrupt structure, 15-9
  GPIO, 14-20
  GPIO interrupt generation, 14-17
  PPI, 7-25
  SPI core-driven, 10-36
  SPI DMA, 10-37
  timer EXT_CLK mode, 15-35
  timer PWM_OUT mode, 15-14
  timer WDTH_CAP mode, 15-27
  TWI master mode, 11-29
  TWI slave mode, 11-28
FLOW mode, DMA, 5-19
FLOW value, DMA, 5-23
FLSx bit, 10-10, 10-42
FMD bit, 9-49
FPE bit, 13-31, 13-32
frame buffer,Ethernet, 8-128
FrameCheckSequenceErrors register, 8-55
framed serial transfers, characteristics, 12-31
framed/unframed data, 12-31
frame filter evaluation, MAC, 8-15
FramesAbortedDueToXSColls register, 8-61
FramesLen1024_MaxReceived register, 8-60
FramesLen128_255Received register, 8-59
FramesLen256_511Received register, 8-60
FramesLen512_1023Received register, 8-60
FramesLen65_127Received register, 8-59
FramesLen65_127Transmitted register, 8-63
FramesLenEq64Received register, 8-59
FramesLenEq64Transmitted register, 8-63
FramesLenLt64Received register, 8-59
FramesLostDueToIntMAC RcvError register, 8-55

FramesLostDueToIntMACXmitError register, 8-62
FramesReceivedAll register, 8-59
FramesReceivedOK register, 8-54
frame start detect, PPI, 7-36
FramesTransmittedAll register, 8-63
FramesTransmittedOK register, 8-60
FramesWithDeferredXmissions register, 8-61
FramesWithExcessiveDeferral register, 8-62
frame sync
  active high/low, 12-33
  early, 12-35
  early/late, 12-35
  external/internal, 12-32
  internal, 12-27
  internally generated, 12-65
  late, 12-35
  multichannel mode, 12-19
  sampling edge, 12-33
  SPORT options, 12-30
frame sync divider[15:0] field, 12-65, 12-66
frame synchronization
  PPI in GP modes, 7-20
  and SPORT, 12-3
frame sync polarity, PPI and timer, 7-21
frame sync pulse
  use of, 12-51
  when issued, 12-51
frame sync signal, control of, 12-51, 12-56
FrameTooLongErrors register, 8-57
frame track error, 7-31, 7-35
FREQ[1:0] field, 20-20, 20-27
frequencies, clock and frame sync, 12-26
frequency, DEB, 2-11
frequency, EAB, 2-11
FSDR bit, 12-22, 12-67
F signal, 7-31
FT_ERR bit, 7-31, 7-33, 7-35
full duplex, 12-4, 12-6
  SPI, 10-1
FULL_ON bit, 20-26
full on mode, 1-23, 20-9
full on mode to active mode example, 20-30
function enable register (PORTF_FER), 14-9
function enable register (PORTG_FER), 14-9
function enable register (PORTH_FER), 14-9
function enable registers (PORTx_FER), 14-23

G

GAIN[1:0] field, 20-19, 20-20, 20-27
gain levels, 20-20
GCALL bit, 11-15, 11-33
GEN bit, 11-31
general call address, TWI, 11-9
general-purpose interrupts, 4-2, 4-4
general-purpose I/O, *See* GPIO
general-purpose modes, PPI, 7-13
general-purpose ports, 1-10, 14-1 to 14-35
  assigning interrupt channels, 14-15
  interrupt channels, 14-15
  interrupt generation flow, 14-15
  latency, 14-9
  pin defaults, 14-3
  pins, interrupt, 14-14
  throughput, 14-9

general-purpose timers, 15-1 to 15-61
aborting, 15-25
and startup errors, 15-11
autobaud mode, 15-34
block diagram, 15-3
buffer registers, 15-47
capture mode, 15-7
clock source, 15-5
code examples, 15-52
control bit summary, 15-50
counter, 15-6
disable timing, 15-25
disabling, 15-39
enabling, 15-7, 15-36, 15-38
error detection, 15-10
EXT_CLK mode, 15-47
external interface, 15-4
features, 15-1
flow diagram for EXT_CLK mode, 15-35
generating maximum frequency, 15-18
illegal states, 15-10, 15-11
internal interface, 15-6
internal timer structure, 15-4
interrupts, 15-6, 15-8, 15-17, 15-31
interrupt setup, 15-54
interrupt structure, 15-9
measurement report, 15-28, 15-29, 15-30
non-overlapping clock pulses, 15-58
output pad disable, 15-15
overflow, 15-6
periodic interrupt requests, 15-55
port setup, 15-52
and PPI, 15-5, 15-23
preventing errors in PWM_OUT mode, 15-48
programming model, 15-36
PULSE_HI toggle mode, 15-18
PWM mode, 15-7
PWM_OUT mode, 15-13 to 15-25, 15-47
registers, 15-37
signal generation, 15-53
single pulse generation, 15-15
size of register accesses, 15-38
stopping in PWM_OUT mode, 15-23
three timers with same period, 15-19
timer outputs, 15-4
two timers with non-overlapping clocks, 15-20
waveform generation, 15-16
WDTH_CAP mode, 15-26, 15-47
WDTH_CAP mode configuration, 15-60
WDTH_CAP mode flow diagram, 15-27
Get_DXE_Address function, 19-33
Get_DXE_Address functions, 19-31
GIRQ bit, 9-46
glitch filtering, UART, 13-9
global interrupt flag register (CAN_GIF), 9-48
global interrupt mask register (CAN_GIM), 9-47
global interrupts, CAN, 9-24
global interrupt status register (CAN_GIS), 9-47
global status register (CAN_STATUS), 9-44
glueless connection, 21-5
GM bit, 10-27, 10-41
GPIO, 1-10, 14-1 to 14-35
assigned to same interrupt channel, 14-18
and booting, 19-19
clearing interrupt condition, 14-16
clear registers, 14-13
code examples, 14-34
configuration, 14-10

GPIO *(continued)*
data registers, 14-10, 14-11, 14-12
direction registers, 14-10, 14-11
edge detection, 14-15
edge-sensitive, 14-12
flow chart, 14-20
function enable registers, 14-10, 14-14
high current option, 14-5
input buffers, 14-11
input driver, 14-11
input enable registers, 14-11, 14-14
interrupt channels, 14-19
interrupt generation flow chart, 14-17
interrupt request, 4-24
interrupts at reset, 14-16
interrupt sensitivity registers, 14-15
mask data registers, 14-16
mask interrupt clear registers, 14-17
mask interrupt set registers, 14-16
mask interrupt toggle registers, 14-18
mask registers, 14-15
pins, 14-2, 14-10
polarity registers, 14-14
port F, 14-5
port G, 14-6
registers, 14-22
set registers, 14-12
toggle registers, 14-13
using as input, 14-11
write operations, 14-11
writes to registers, 14-13
GPIO clear registers (PORTxIO_CLEAR), 14-25
GPIO data registers (PORTxIO), 14-24
GPIO direction registers (PORTxIO_DIR), 14-23
GPIO input enable registers (PORTxIO_INEN), 14-24
GPIO mask interrupt A clear registers (PORTxIO_MASKA_CLEAR), 14-31
GPIO mask interrupt A registers (PORTxIO_MASKA), 14-28
GPIO mask interrupt A set registers (PORTxIO_MASKA_SET), 14-29
GPIO mask interrupt A toggle registers (PORTxIO_MASKA_TOGGLE), 14-33
GPIO mask interrupt B clear registers (PORTxIO_MASKB_CLEAR), 14-32
GPIO mask interrupt B registers (PORTxIO_MASKB), 14-28
GPIO mask interrupt B set registers (PORTxIO_MASKB_SET), 14-30
GPIO mask interrupt B toggle registers (PORTxIO_MASKB_TOGGLE), 14-34
GPIO pins, 14-9
GPIO polarity registers (PORTxIO_POLAR), 14-26
GPIO set on both edges registers (PORTxIO_BOTH), 14-27
GPIO set registers (PORTxIO_SET), 14-25
GPIO toggle registers (PORTxIO_TOGGLE), 14-26
GP modes, PPI, 7-15
ground plane, 21-9
GUWKE bit, 8-82, 8-83

H

H.100, 12-22, 12-25
handshake DMA, 1-8
handshake MDMA, 5-11, 5-40
interrupts, 5-43

handshake MDMA control registers (HMDMAx_CONTROL), 5-99, 5-100
handshake MDMA current block count registers (HMDMAx_BCOUNT), 5-101, 5-102
handshake MDMA current edge count registers (HMDMAx_ECOUNT), 5-102, 5-103
handshake MDMA edge count overflow interrupt registers (HMDMAx_ECOVERFLOW), 5-104
handshake MDMA edge count overflow interrupt registers (HMDMAx_ECOVERFLOW), 5-104
handshake MDMA edge count urgent registers (HMDMAx_ECURGENT), 5-104
handshake MDMA initial block count registers (HMDMAx_BCINIT), 5-101
handshake MDMA initial edge count registers (HMDMAx_ECINIT), 5-103
handshake MDMA initial edge count registers (HMDMAx_ECINIT), 5-103
handshake memory DMA, 5-2
hardware reset, 19-8
hibernate state, 1-24, 6-46, 20-11, 20-22
  and CAN, 9-38
  entering, 20-32
high current option, 14-5
high frequency design considerations, 21-8
HM bit, 8-64, 8-69
HMDMA, 5-11
  pins, 14-2
HMDMAEN bit, 5-40, 5-41, 5-100
HMDMAx_BCINIT (handshake MDMA configuration registers), 5-40, 5-101
HMDMAx_BCOUNT (handshake MDMA current block count registers), 5-41, 5-101, 5-102
HMDMAx_CONTROL (handshake MDMA control registers), 5-6, 5-99, 5-100
HMDMAx_ECINIT (handshake MDMA initial edge count registers), 5-41, 5-103
HMDMAx_ECOUNT (handshake MDMA current edge count registers), 5-41, 5-102, 5-103
HMDMAx_ECOVERFLOW (handshake MDMA edge count overflow interrupt registers), 5-104
HMDMAx_ECURGENT (handshake MDMA edge count urgent registers), 5-104
HMVIP, 12-26
hold, for EBIU asynchronous memory controller, 6-20
horizontal blanking, 7-7
horizontal tracking, PPI, 7-32
hours[3:0] field, 18-20, 18-22
hours[4] bit, 18-20, 18-22
hours event flag bit, 18-21
hours interrupt enable bit, 18-20
HU bit, 8-64, 8-70
HWAIT strobe, 19-19

I

I2C, *See* TWI
I^2C bus standard, 1-11, 11-1
I^2S, 1-16
  format, 12-12
  serial devices, 12-2
IDE bit, 9-53

idle state
  waking from, 4-9
IEEE 1149.1 standard, *See* JTAG standard
IEEE 802.3, 1-14, 8-2, 8-7, 8-43
IFE bit, 8-64, 8-69
IGNORE bit, 19-14
IGNORE block, 19-27
iInitialization code, 19-22
INIT bit, 19-23
initializing
  CAN, 9-9
  DMA, 5-20
inner loop address increment registers
  (DMAx_X_MODIFY), 5-88
  (MDMA_yy_X_MODIFY), 5-88
inner loop count registers
  (DMAx_X_COUNT), 5-85
  (MDMA_yy_X_COUNT), 5-85
input buffers, GPIO, 14-11
input clock, *See* CLKIN
input delay bit, 20-26
input driver, GPIO, 14-11
InRangeLengthErrors register, 8-57
instruction bit scan ordering, B-4
instruction register, B-2, B-4
instructions, 1-27
  See also instructions by name
Intel hex loader file, 19-38
interfaces
  external memory, 6-6
  internal memory, 6-5
  on-chip, 2-2
  overview, 2-3
  RTC, 18-2
  system, 2-1
Inter IC bus, 11-1
interlaced video, 7-7
interleaving
  of data in SPORT FIFO, 12-58
  SPORT data, 12-7
internal address mapping (table), 6-64
internal bank, 6-33, G-12
internal boot ROM, 19-1
internal clocks, 2-4
internal/external frame syncs, *See* frame sync
internal memory, 1-6
  accesses, 3-2
  interfaces, 6-5
internal SDRAM banks, 6-29
internal supply regulator, shutting off, 20-22
internal voltage levels, changing, 20-33
interrupt A, 14-15
interrupt B, 14-15
interrupt channels, UART, 13-27
interrupt conditions, UART, 13-28
interrupt handler and DMA synchronization, 5-63
interrupt output, SPI, 10-23
interrupt request lines, peripheral, 4-2
interrupts, 4-1 to 4-25
  assigning priority for UART, 13-12
  CAN, 9-23
  channels, assigning, 14-15
  channels, GPIO, 14-15
  clearing requests, 4-23
  configuring and servicing, 21-2
  configuring for MAC, 8-47
  control of system, 4-2
  core timer, 16-3
  default mapping, 4-4
  definition, 4-4
  determining source, 4-8
  DMA channels, 4-9
  DMA_ERROR, 5-32
  DMA error, 5-81
  DMA overflow, 5-44
  DMA queue completion, 5-64

interrupts *(continued)*
enabling, 4-8
ethernet event, 8-38, 8-39
evaluation of GPIO interrupts, 14-18
general-purpose, 4-2, 4-4
general-purpose timers, 15-6, 15-8, 15-17, 15-31
generated by peripheral, 4-14
GPIO, 14-14, 14-19
handshake MDMA, 5-43
initialization, 4-14
inputs and outputs, 4-7
mapping, 4-7
mask function, 4-10
multiple sources, 4-15
peripheral, 4-2, 4-4, 4-7 to 4-13
peripheral IDs, 4-11
peripheral interrupt events, 4-11
prioritization, 4-7
processing, 4-1, 4-14
programming examples, 4-23 to 4-25
reset, 19-9
routing overview, 4-3
RTC, 18-15
shared, 4-7
software, 4-4
SPI, 10-23, 10-47
SPORT error, 12-38
SPORT RX, 12-38, 12-62
SPORT TX, 12-38, 12-59
system, 4-1
to wake core from idle, 4-9
UART, 13-10
use in managing a descriptor queue, 5-62
interrupt sensitivity registers (PORTxIO_EDGE), 14-27
interrupt service routine, determining source of interrupt, 4-8
interrupt status registers (DMAx_IRQ_STATUS), 5-78, 5-79
(MDMA_yy_IRQ_STATUS), 5-78, 5-79
I/O interface to peripheral serial device, 12-3
I/O memory space, 1-7
I/O pins, general-purpose, 14-10
IP checksum, MAC, 8-19
IRCLK bit, 12-54, 12-56
IrDA, 13-31
receiver, 13-9
transmitter, 13-8
IREN bit, 13-31
IRFS bit, 12-32, 12-54, 12-56
IRPOL bit, 13-9
IRQ_ENA bit, 15-44, 15-50, 15-52
ISR and multiple interrupt sources, 4-15
ISR for the MMC interrupt, structure, 8-45
ITCLK bit, 12-48, 12-50
ITFS bit, 12-20, 12-32, 12-48, 12-51
ITHR[15:0] field, 5-104
ITU-R 601, 7-16
ITU-R 601/656, 1-14
ITU-R 656, 7-10
modes supported, 1-15
ITU-R 656 modes, 7-6, 7-30
and DLEN field, 7-26
frame start detect, 7-36
frame synchronization, 7-12
output, 7-12

J

JPEG compression, PPI, 7-39
JTAG, 1-30, B-1, B-2, B-4
JTAG port, 19-7

L

L1 data cache, 3-8
L1 data memory, 1-6
  subbanks, 3-7
L1 instruction memory, 1-6, 3-5
  address alignment, 3-5
  subbanks, 3-6
L1 memory, 2-4
  and DMA, 5-6
L1 scratchpad RAM, 1-6
LARFS bit, 12-35, 12-54, 12-57
large descriptor mode, DMA, 5-17
large model mode, DMA, 5-76
LateCollisions register, 8-61
late frame sync, 12-18, 12-35
latency
  DAB, 2-10
  DMA, 5-27
  general-purpose ports, 14-9
  powerup, 6-56
  SDC, 6-41
  SDRAM, 6-73
LATFS bit, 12-35, 12-48, 12-52
LB bit, 8-64, 8-65
LCRTE bit, 8-28
LCTRE bit, 8-64, 8-66
LdrViewer, 19-59
lines per frame register (PPI_FRAME), 7-35, 7-36
line terminations, SPORT, 12-9
linked list, DMA, 8-127
little endian byte order, 11-44
loader file, 19-11
loader utility, 19-11
LOCKCNT[15:0] field, 20-27
locked transfers, DMA, 2-10
loopback feature, PPI, 7-11
loopback mode, UART, 13-23
LOOP bit, 13-23
LOSTARB bit, 11-14, 11-38
LRFS bit, 12-31, 12-33, 12-54, 12-57
LSBF bit, 10-41
LT_ERR_OVR bit, 7-31, 7-33
LT_ERR_OVR flag, 7-32
LT_ERR_UNDR bit, 7-31, 7-33
LT_ERR_UNDR flag, 7-32
LTFS bit, 12-20, 12-31, 12-33, 12-48, 12-52

M

MAA bit, 9-45
MAC, 1-14, 8-1 to 8-132
  aborted frames, 8-16
  address filter evaluation, 8-14
  address setup, 8-129
  alternative descriptor structure, 8-26
  block diagram, 8-3
  clocking, 8-4
  code examples, 8-126
  configuration, 8-46
  configuring address registers, 8-48
  configuring interrupts, 8-47
  configuring PHY, 8-50
  continuous polling, 8-44
  control frames, 8-16
  CRC-16 hash value calculation, 8-38
  CRC-32 address calculation, 8-73
  CRC state, 8-37
  descriptor structure, alternative, 8-26
  discarded frames, 8-16
  DMA configuration, 8-126
  DMA data transfer, 8-50
  DMA descriptor pairs, 8-12, 8-24
  DMA descriptors, 8-127
  ethernet event interrupts, 8-38, 8-39
  Ethernet frame buffer, 8-128
  Ethernet frame header, 8-50
  flexible descriptor structure, 8-27
  frame filter evaluation, 8-15

MAC *(continued)*
frame reception and filtering, 8-13
general description, 1-14
internal interface, 8-6
IP checksum, 8-19
late collisions, 8-28
linked list of DMAs, 8-127
Magic Packet detection, 8-34
management counters, 8-43
MII management interface, 8-7
MMC interrupt, 8-40, 8-45
multiplexed pins, 8-47
multiplexing, 8-5
operation in sleep state, 8-33
PHY control routines, 8-130
PHYINT interrupt, 8-40
pins, 8-5, 14-3
and PMAP field, 5-49
port H, 14-7
power management states, 8-7
protocol compliance, 8-7
read access to the PHY, 8-131
receive DMA operation, 8-11
receiving data, 8-51
registers, table, 8-51
remote wakeup frame filters, 8-35
RX/ TX frame status interrupt operation, 8-42
RX automatic pad stripping, 8-17
RX DMA buffer structure, 8-18
RX DMA data alignment, 8-18
RX DMA direction error detected, 8-41
RX DMA direction errors, 8-22
RX frame status buffer, 8-19
RX frame status classification, 8-20
RX frame status interrupt, 8-40
RX frame status register operation, 8-43
RX IP frame checksum calculation, 8-21
RX receive status priority, 8-20
speculative read, 8-44
station management read, 8-9

MAC *(continued)*
station management read transfer, 8-49
station management transfer done, 8-41
station management write, 8-9
station management write transfer, 8-49
transfer frame protocol, 8-8
transmit DMA operation, 8-24
transmitting data, 8-51
TX DMA data alignment, 8-27
TX DMA direction error detected, 8-41
TX DMA direction errors, 8-30
TX frame status classification, 8-29
TX frame status interrupt, 8-41
TX frame status register operation, 8-43
TX transmit status priority, 8-29
type definitions, 8-126
wake from hibernate, 8-31
wake from sleep, 8-32
wakeup frame detected, 8-41
MAC address high[15:0] field, 8-71
MAC address high register (EMAC_ADDRHI), 8-71
MAC address low[15:0] field, 8-71
MAC address low[31:16] field, 8-71
MAC address low register (EMAC_ADDRLO), 8-70
MACControlFramesReceived register, 8-58
MACControlFramesTransmitted register, 8-63
MAC flow control register (EMAC_FLC), 8-78
MAC management counters control register (EMAC_MMC_CTL), 8-124
MAC operating mode register (EMAC_OPMODE), 8-64
MAC station management address register (EMAC_STAADD), 8-76
MAC station management data register (EMAC_STADAT), 8-78

MAC system control register (EMAC_SYSCTL), 8-93
MAC system status register (EMAC_SYSTAT), 8-95
MAC VLAN1 tag register (EMAC_VLAN1), 8-80
MAC VLAN2 tag register (EMAC_VLAN2), 8-80
MAC wakeup frame control and status register (EMAC_WKUP_CTL), 8-81
MAC wakeup frame filter commands register (EMAC_WKUP_FFCMD), 8-89
MAC wakeup frame filter CRC0/1 register (EMAC_WKUP_FFCRC0), 8-92
MAC wakeup frame filter CRC2/3 register (EMAC_WKUP_FFCRC1), 8-92
MAC wakeup frameX byte mask registers (EMAC_WKUP_FFMSKx), 8-84
MADDR[6:0] field, 11-37
Magic Packet, 8-34
mailbox configuration register 1 (CAN_MC1), 9-69
mailbox configuration register 2 (CAN_MC2), 9-69
mailbox direction register 1 (CAN_MD1), 9-70
mailbox direction register 2 (CAN_MD2), 9-70
mailboxes, CAN, 9-4
mailbox interrupt mask register 1 (CAN_MBIM1), 9-79
mailbox interrupt mask register 2 (CAN_MBIM2), 9-80
mailbox interrupts, CAN, 9-23
mailbox receive interrupt flag register 1 (CAN_MBRIF1), 9-81
mailbox receive interrupt flag register 2 (CAN_MBRIF2), 9-82
mailbox transmit interrupt flag register 1 (CAN_MBTIF1), 9-80
mailbox transmit interrupt flag register 2 (CAN_MBTIF2), 9-81
mailbox word 0 register (CAN_MBxx_DATA0), 9-67
mailbox word 1 register (CAN_MBxx_DATA1), 9-65
mailbox word 2 register (CAN_MBxx_DATA2), 9-63
mailbox word 3 register (CAN_MBxx_DATA3), 9-61
mailbox word 4 register (CAN_MBxx_LENGTH), 9-59
mailbox word 5 register (CAN_MBxx_TIMESTAMP), 9-57
mailbox word 6 register (CAN_MBxx_ID0), 9-55
mailbox word 7 register (CAN_MBxx_ID1), 9-53
manual
  conventions, -lv
  related documents, -l
mapping
  default interrupt, 4-11
  peripheral to DMA, 4-10
master control register (CAN_CONTROL), 9-43
masters
  DAB, 2-9
  PAB, 2-6
MBDI bit, 5-44, 5-100
MBIMn bit, 9-79, 9-80
MBPTR[4:0] field, 9-44
MBRIFn bit, 9-81, 9-82
MBRIRQ bit, 9-46
MBTIFn bit, 9-80, 9-81
MBTIRQ bit, 9-46
MCCRM[1:0] field, 12-67
MCDRXPE bit, 12-67

MCDTXPE bit, 12-67
MCMEN bit, 12-18, 12-67
MCOMP bit, 11-18, 11-43
MCOMPM bit, 11-41, 11-42
MCx bit, 9-69
MDCDIV[5:0] field, 8-93, 8-94
MDIO station management interface, 8-7
MDIR bit, 11-34, 11-36
MDMA controllers, 5-9
MDMA_ROUND_ROBIN_COUNT[4:0] field, 5-52, 5-106
MDMA_ROUND_ROBIN_PERIOD field, 5-52, 5-106
MDMA_yy_CONFIG (DMA configuration registers), 5-74
MDMA_yy_CURR_ADDR (current address registers), 5-83
MDMA_yy_CURR_DESC_PTR (current descriptor pointer registers), 5-96
MDMA_yy_CURR_X_COUNT (current inner loop count registers), 5-86, 5-87
MDMA_yy_CURR_Y_COUNT (current outer loop count registers), 5-91
MDMA_yy_IRQ_STATUS (interrupt status registers), 5-78, 5-79
MDMA_yy_NEXT_DESC_PTR (next descriptor pointer registers), 5-94
MDMA_yy_PERIPHERAL_MAP (peripheral map registers), 5-71
MDMA_yy_START_ADDR (start address registers), 5-82
MDMA_yy_X_COUNT (inner loop count registers), 5-85
MDMA_yy_X_MODIFY (inner loop address increment registers), 5-88
MDMA_yy_Y_COUNT (outer loop count registers), 5-90
MDMA_yy_Y_MODIFY (outer loop address increment registers), 5-93
MDn bit, 9-70
measurement report, general-purpose timers, 15-28, 15-29, 15-30
memory, 3-1 to 3-10
  accesses to internal, 3-2
  ADSP-BF534, 3-3
  ADSP-BF536, 3-4
  ADSP-BF537, 3-5
  architecture, 1-4, 3-1
  asynchronous, 3-2
  asynchronous interface, 21-5
  asynchronous region, 6-2
  boot ROM, 3-8
  configurations, 1-5, 3-1, 3-2
  external, 1-6, 3-8
  external memory, 6-6
  external memory map, 3-2
  external SDRAM, 6-27
  internal, 1-6
  internal bank, 6-33, G-12
  I/O space, 1-7
  L1, 2-4
  L1 data, 1-6, 3-7
  L1 data cache, 3-8
  L1 instruction, 1-6, 3-5
  L1 scratchpad, 1-6
  moving data between SPORT and, 12-38
  off-chip, 1-6
  on-chip, 1-6
  SDRAM, 3-2
  start locations of L1 instruction memory subbanks, 3-6
  structure, 1-4
  unpopulated, 6-10
memory conflict, DMA, 5-53
memory DMA, 1-8, 5-9
  bandwidth, 5-47
  buffers, 5-10
  channels, 5-9
  descriptor structures, 5-67

memory DMA *(continued)*
handshake operation, 5-11
priority, 5-51
scheduling, 5-51
timing, 5-48
transfer operation, starting, 5-11
transfer performance, 2-11, 2-12
transfers, 5-2, 5-7
word size, 5-10
memory map, external (figure), 6-3
memory-mapped registers, *See* MMRs
memory-to-memory transfer, 5-10
MEN bit, 11-34, 11-36
MERR bit, 11-18, 11-43
MERRM bit, 11-41, 11-42
MFD[3:0] field, 12-21, 12-67
MII
communications protocol, setting up, 8-48
management interface, 8-7
multiplexing, 8-5
pins, 8-5, 14-3
port H, 14-7
port J, 14-8
minutes[5:0] field, 18-20, 18-22
minutes event flag bit, 18-21
minutes interrupt enable bit, 18-20
MISO pin, 10-4, 10-5, 10-15, 10-17, 10-18, 10-19, 10-27
µ-law companding, 12-24, 12-29
MMC, continuous polling, 8-44
MMCE bit, 8-43, 8-124
MMCINT bit, 8-95, 8-97
MMC interrupt, 8-40, 8-45
MMRs, 1-7
address range, A-1
width, A-1
mode fault error, 10-21, 10-23
modes
boot, 1-25, 19-34
broadcast, 10-10, 10-17, 10-18
multichannel, 12-15
serial port, 12-11
SPI master, 10-17, 10-24
SPI slave, 10-18, 10-26
UART DMA, 13-17
UART non-DMA, 13-15
MODF bit, 10-21, 10-43
MOSI pin, 10-4, 10-5, 10-15, 10-17, 10-18, 10-19, 10-27
moving data, serial port, 12-38
MPEG compression, PPI, 7-39
MPKE bit, 8-34, 8-82, 8-83
MPKS bit, 8-34, 8-82
MPROG bit, 11-14, 11-38
MRB bit, 9-45
MRS command, 6-34, 6-49
MSEL[5:0] field, 20-3, 20-4, 20-26
MSTR bit, 10-19, 10-41
multibank operation, 6-43
MulticastFramesReceivedOK register, 8-56
MulticastFramesXmittedOK register, 8-62
multichannel frame, 12-20
multichannel frame delay field, 12-21
multichannel mode, 12-15
enable/disable, 12-18
frame syncs, 12-19
SPORT, 12-19
multichannel operation, SPORT, 12-15 to 12-25
multi-DXE, 19-26
MultipleCollisionFrames register, 8-60
multiple interrupt sources, 4-15
multiple slave SPI systems, 10-10
multiplexed SDRAM addressing scheme figure, 6-28

multiplexing, 14-1
MII and RMII, 8-5
port F, 14-5
port G, 14-6
port H, 14-7
port J, 14-8
PPI, 7-3
MVIP-90, 12-26

N

NAK bit, 11-31, 11-32
NDPH bit, 5-23
NDPL bit, 5-23
NDSIZE[3:0] field, 5-18, 5-74, 5-76
legal values, 5-34
next descriptor pointer registers
(DMAx_NEXT_DESC_PTR), 5-94
(MDMA_yy_NEXT_DESC_PTR), 5-94
NINT bit, 13-29
no-boot mode, 19-8
no boot on software reset bit, 19-5
nominal bit rate, CAN, 9-11
nominal bit time, CAN, 9-10
no operation command, 6-54
NOP command, 6-54
normal frame sync mode, 12-35
normal timing, serial port, 12-35
NTSC systems, 7-7

O

OctetsReceivedAll register, 8-59
OctetsReceivedOK register, 8-55
OctetsTransmittedAll register, 8-63
OctetsTransmittedOK register, 8-61
OE bit, 13-24
off-chip memory, 1-6
off-core accesses, 2-5
offsets, DMA descriptor elements, 5-19
OI bit, 5-100
OIE bit, 5-100
onboard regulation, bypassing, 20-20
on-chip memory, 1-6
on-chip switching regulator controller, 20-18
open drain drivers, 10-1
open drain outputs, 10-17
open page, 6-35, G-1
operating modes, 20-8
active, 1-23, 20-9
deep sleep, 1-24, 20-10
full on, 1-23, 20-9
hibernate state, 1-24, 20-11
PPI, 7-5
sleep, 1-23, 20-9
transition, 20-11, 20-12
OPSSn bit, 9-73
optimization, of DMA performance, 5-44
oscilloscope probes, 21-13
OUT_DIS bit, 15-43, 15-44, 15-51
outer loop address increment registers
(DMAx_Y_MODIFY), 5-93
(MDMA_yy_Y_MODIFY), 5-93
outer loop count registers
(DMAx_Y_COUNT), 5-90
(MDMA_yy_Y_COUNT), 5-90
OutOfRangeLengthField register, 8-57
output delay bit, 20-26
output pad disable, timer, 15-15
outputs, programmable pins, 21-12
overflow interrupt, DMA, 5-44
overwrite protection/single shot transmission register 1 (CAN_OPSS1), 9-73
overwrite protection/single shot transmission register 2 (CAN_OPSS2), 9-73
OVR bit, 7-31, 7-33

P

PAB, 2-6
arbitration, 2-6
bus agents (masters, slaves), 2-6
clocking, 20-2
and EBIU, 6-5
errors generated by SPORT, 12-39
performance, 2-7
PACK_EN bit, 7-27, 7-28
packing, serial port, 12-24
page hit, and SDC, 6-41
page miss, 6-52
page miss, and SDC, 6-41
page size, 6-64
PAL systems, 7-7
PAM bit, 8-64, 8-69
parallel peripheral interface, *See* PPI
PASR[1:0] field, 6-68, 6-70
PASR feature, 6-29
PAUSEMACCtrlFramesReceived register, 8-58
PBF bit, 8-64, 8-68
PC100 SDRAM standard, 6-1
PC133 SDRAM controller, 1-9
PC133 SDRAM standard, 6-1
PCF bit, 8-79
PDWN bit, 20-26
PE bit, 13-24, 13-25
PEN bit, 13-21
performance
DAB, 2-10
DEB, 2-11
DMA, 5-46
EAB, 2-11
general-purpose ports, 14-9
memory DMA, 5-47
memory DMA transfers, 2-11, 2-12
optimization, DMA, 5-44
PAB, 2-7
SDRAM, 6-31
PERIOD_CNT bit, 15-13, 15-22, 15-28, 15-44, 15-50
period value[15:0] field, 16-6
period value[31:16] field, 16-6
peripheral access bus, *See* PAB
peripheral DMA, 5-7
peripheral DMA channels, 5-45
peripheral DMA start address registers, 5-82
peripheral DMA transfers, 5-2
peripheral error interrupts, 5-80
peripheral interrupt request lines, 4-2
peripheral interrupts, 4-2, 4-4, 4-7 to 4-13
peripheral map registers
(DMAx_PERIPHERAL_MAP), 5-71
(MDMA_yy_PERIPHERAL_MAP), 5-71
peripheral pins, default configuration, 14-10
peripherals, 1-1
and buses, 1-2
compatible with SPI, 10-2
configuring for an IVG priority, 4-17
default mapping to DMA, 5-7
and DMA controller, 5-34
DMA support, 1-2
enabling, 14-3
interrupt events, 4-11
interrupt generated by, 4-14
interrupt IDs, 4-11
interrupts, clearing, 4-23
level-sensitivity of interrupts, 4-25
list of, 1-1
mapping to DMA, 4-10, 5-49
multiplexing, 14-1
remapping DMA assignment, 5-8
switching from DMA to non-DMA, 5-81
timing, 2-4
used to wake from idle, 4-9

PF0 bit, 5-6
PF0 pin, 14-13
PF1 bit, 5-6
PFDE bit, 5-6, 14-22
PFFE bit, 14-22
PFLAG[3:0] field, 19-14
-PFLAG switch, 19-19
PFS4E bit, 14-22
PFS5E bit, 14-22
PFS6E bit, 14-22
PFTE bit, 14-22
PFx pin, 10-8
PGRE bit, 14-22
PGSE bit, 14-22
PGTE bit, 14-22
PHY, 8-4
  configuring, 8-50
  control routines, 8-130
  initialization, minimum requirements, 8-132
  read access, 8-131
PHYAD[4:0] field, 8-76
PHYCLKOE bit, 8-47
PHYIE bit, 8-93, 8-94
PHYINT bit, 8-95, 8-97
PHYINT interrupt, 8-40
PHYWE bit, 20-27
pins, 21-1
  MAC, 8-5
  multiplexing, 14-1
  PPI, 7-4
  unused, 21-12
pin state during SDC commands (table), 6-48
pin terminations, SPORT, 12-9
pipeline, lengths of, 5-57
pipelining
  DMA requests, 5-41
  SDC supported, 6-75
PJCE[1:0] field, 14-22
PJSE bit, 14-22
PJx pin, 10-8
PLL, 20-1 to 20-34
  active mode, 20-9
  active mode, effect of programming for, 20-16
  applying power to the PLL, 20-13
  block diagram, 20-3
  BYPASS bit, 20-9, 20-17
  bypassing onboard regulation, 20-20
  CCLK derivation, 20-3
  changing CLKIN-to-VCO multiplier, 20-13
  changing clock ratio, 20-6
  clock dividers, 20-3
  clocking to SDRAM, 20-10
  clock multiplier ratios, 20-3
  code example, active mode to full on mode, 20-29
  code example, changing clock multiplier, 20-31
  code example, changing internal voltage levels, 20-33
  code example, full on mode to active mode, 20-30
  code example, setting wakeups and entering hibernate state, 20-32
  code examples, 20-28
  configuration, 20-3
  control bits, 20-11
  deep sleep mode, effect of programming for, 20-17
  design, 20-2
  disabled, 20-13
  divide frequency, 20-3
  DMA access, 20-9, 20-17
  dynamic power management controller (DPMC), 20-7

PLL *(continued)*
enabled, 20-13
enabled but bypassed, 20-9
full on mode, effect of programming for, 20-16
and internal clocks, 2-4
maximum performance mode, 20-9
modification, activating changes to DF or MSEL, 20-15
modification in active mode, 20-13
multiplier select (MSEL) field, 20-3
new multiplier ratio, 20-13
operating modes, operational characteristics, 20-8
operating mode transitions, 20-14
PDWN bit, 20-11
PLL_LOCKED bit, 20-16
PLL_OFF bit, 20-13
PLL status (table), 20-8
power domains, 20-18
powering down core, 20-22
power savings by operating mode (table), 20-8
processing during PLL programming sequence, 20-16
programming sequence, 20-15
registers, table, 20-25
relocking after changes, 20-16
removing power to the PLL, 20-13
RTC interrupt, 20-10, 20-17
SCLK derivation, 20-2, 20-3
sleep mode, 20-9, 20-16
STOPCK bit, 20-11
voltage control, 20-7, 20-21
wakeup signal, 20-16
PLL control register (PLL_CTL), 20-25, 20-26
PLL_CTL (PLL control register), 20-4, 20-25, 20-26
PLL divide register (PLL_DIV), 20-25
PLL_DIV (PLL divide register), 20-5, 20-25
PLL_LOCKCNT (PLL lock count register), 20-25, 20-27
PLL lock count register (PLL_LOCKCNT), 20-27
PLL_LOCKED bit, 20-26
PLL_OFF bit, 20-26
PLL_STAT (PLL status register), 20-25, 20-26
PLL status register (PLL_STAT), 20-26
PLL VCO frequency, changing, 6-45
PMAP[3:0] field, 5-7, 5-48, 5-49, 5-71
polarity, GPIO, 14-14
POLC bit, 7-5, 7-26, 7-27
polling DMA registers, 5-56
POLS bit, 7-5, 7-26, 7-27
PORT_CFG[1:0] field, 7-5, 7-27, 7-30
port connection, SPORT, 12-7
PORT_DIR bit, 7-5, 7-27, 7-30
PORT_EN bit, 7-27, 7-30
port F
and general-purpose timers, 15-4
GPIO, 14-5, 14-10
interrupt A channel, 14-19
multiplexing, 14-5
peripherals, 14-2
and PPI, 7-3, 14-4
and SPI, 10-4
structure, 14-4
timer port setup, 15-52
timers, 14-4
UART, 14-4
PORTF_FER and booting, 19-20
PORTF_FER (function enable register), 5-6, 7-4, 14-9

port G
GPIO, 14-6, 14-10
interrupt A channel, 14-19
multiplexing, 14-6
peripherals, 14-2, 14-5
and PPI, 7-3, 14-6
SPORT, 14-6
structure, 14-5
PORTG_FER (function enable register), 7-4, 14-9
port H
GPIO, 14-10
and MAC, 8-47, 14-7
MII, 14-7
multiplexing, 14-7
peripherals, 14-3
RMII, 14-7
structure, 14-6
timers, 14-7
PORTH_FER (function enable register), 14-9
port J
CAN, 14-8
and MAC, 8-47
MII, 14-8
multiplexing, 14-8
peripherals, 14-3, 14-8
and SPI, 10-4
SPI, 14-8
SPORT, 14-8
structure, 14-7
TWI, 14-8
port multiplexer control register (PORT_MUX), 14-9, 14-22
PORT_MUX (port multiplexer control register), 5-6, 7-4, 14-9, 14-22
port pins, 10-43, 14-3
port pins, test access, B-2
PORT_PREF0 bit, 3-9
PORT_PREF1 bit, 3-9
ports, 1-10
See also ports by name
port width, PPI, 7-28
PORTx_FER (function enable registers), 14-14, 14-23
PORTxIO_BOTH (GPIO set on both edges registers), 14-27
PORTxIO_CLEAR (GPIO clear registers), 14-25
PORTxIO_DIR (GPIO direction register), 14-23
PORTxIO_EDGE (interrupt sensitivity registers), 14-27
PORTxIO (GPIO data registers), 14-24
PORTxIO_INEN (GPIO input enable registers), 14-14, 14-24
PORTxIO_MASKA_CLEAR (GPIO mask interrupt A clear registers), 14-31
PORTxIO_MASKA (GPIO mask interrupt A registers), 14-28
PORTxIO_MASKA_SET (GPIO mask interrupt A set registers), 14-29
PORTxIO_MASKA_TOGGLE (GPIO mask interrupt A toggle registers), 14-33
PORTxIO_MASKB_CLEAR (GPIO mask interrupt B clear registers), 14-32
PORTxIO_MASKB (GPIO mask interrupt B registers), 14-28
PORTxIO_MASKB_SET (GPIO mask interrupt B set registers), 14-30
PORTxIO_MASKB_TOGGLE (GPIO mask interrupt B toggle registers), 14-34
PORTxIO_POLAR (GPIO polarity registers), 14-26
PORTxIO_SET (GPIO set registers), 14-25

PORTxIO_TOGGLE (GPIO toggle registers), 14-26
power dissipation, 20-18
power domains, 20-18
powering down core, 20-22
power management, 1-23, 20-1 to 20-34
  active mode to full on mode, 20-29
  changing internal voltage levels, 20-33
  full on mode to active mode, 20-30
  modification of PLL, 20-15
  setting wakeups and entering hibernate state, 20-32
power supply management, 20-18
powerup latency, SDC, 6-56
powerup sequence, 6-49, 6-73
PPI, 1-14, 7-1 to 7-40
  active video only mode, 7-11
  block diagram, 7-2
  clearing DMA completion interrupt, 7-39
  clock input, 7-3
  configure DMA registers, 7-37
  configuring registers, 7-38
  control byte sequences, 7-9
  control signal polarities, 7-26
  data input modes, 7-15 to 7-17
  data movement, 7-10
  data output modes, 7-18 to 7-19
  data width, 7-26
  delay before starting, 7-34
  DMA operation, 7-23
  edge-sensitive inputs, 7-21
  enabling, 7-3, 7-30, 7-38
  enabling DMA, 7-38
  entire field mode, 7-10
  external frame syncs, 7-16, 7-17, 7-18
  features, 7-1
  FIFO, 7-31
  flow diagram, 7-25

PPI *(continued)*
  frame start detect, 7-36
  frame synchronization with ITU-R 656, 7-12
  frame sync polarity with timer peripherals, 7-21
  frame sync signals, 7-3
  frame track error, 7-31, 7-35
  and general-purpose timers, 15-23
  general flow for GP modes, 7-15
  general-purpose modes, 7-13
  GP modes, 7-15
  GP modes, frame synchronization, 7-20
  GP output, 7-20
  hardware signalling, 7-16
  horizontal tracking, 7-32
  interlaced video, 7-7
  internal frame syncs, 7-17, 7-19
  ITU-R 656 modes, 7-6, 7-12
  loopback feature, 7-11
  MMRs, 7-26
  multiplexed with general-purpose timers, 15-5
  multiplexed with SPORT, 14-6
  multiplexing, 7-3
  no frame syncs, 7-16, 7-18
  number of lines per frame, 7-35
  number of samples, 7-34
  operating modes, 7-5, 7-26
  pins, 7-3, 7-4, 14-2
  port F, 14-4
  port G, 14-6
  port width, 7-28
  preamble, 7-8
  progressive video, 7-7
  submodes for ITU-R 656, 7-10
  and synchronization with DMA, 7-14
  timer pins, 7-21
  transfer delay, 7-19
  valid data detection, 7-16

PLL *(continued)*
vertical blanking interval only mode, 7-11
video data transfer, 7-39
video frame partitioning, 7-8
video processing, 7-6
video streams, 7-9
when data transfer begins, 7-30
PPI_CLK pin, 7-3
PPI_CLK signal, 7-26
PPI_CONTROL (PPI control register), 7-26, 7-27
PPI control register (PPI_CONTROL), 7-26, 7-27
PPI_COUNT[15:0] field, 7-35
PPI_COUNT (transfer count register), 7-34, 7-35
PPI_DELAY[15:0] field, 7-34
PPI_DELAY (delay count register), 7-34
PPI_FRAME[15:0] field, 7-36
PPI_FRAME (lines per frame register), 7-35, 7-36
PPI_FS1 signal, 7-26
PPI_FS2 signal, 7-26
PPI_FS3 signal, 7-31
PPI_STATUS (PPI status register), 7-30, 7-33
PPI status register (PPI_STATUS), 7-30, 7-33
PPORT[1:0] field, 19-14
PR bit, 8-64, 8-69
preamble, PPI, 7-8
precharge all command, 6-35, 6-52
precharge command, 6-35, G-18
PREN bit, 18-22
prescale[6:0] field, 11-30
prescaler, RTC, 18-1
prescaler enable register (RTC_PREN), 18-22
priorities
memory DMA operations, 5-51
peripheral DMA operations, 5-51
prioritization
DMA, 5-48 to 5-55
interrupts, 4-7
private instructions, B-4
probes, oscilloscope, 21-13
processor block diagram, 1-3, 1-4
programmable outputs, 21-12
programmable timing characteristics, EBIU, 6-12
program Pxn bit, 14-24
progressive video, 7-7
propagation segment, CAN, 9-11
PS bit, 5-100
PSF bit, 8-64, 8-68
PSM bit, 6-49, 6-57, 6-68, 6-72
PSSE bit, 6-45, 6-49, 6-57, 6-68, 6-73, 10-19, 10-41
public instructions, B-4
public JTAG scan instructions, B-5
PULSE_HI bit, 15-16, 15-18, 15-44, 15-50
PULSE_HI toggle mode, 15-18
pulse width count and capture mode, *See* WDTH_CAP mode
pulse width modulation mode, *See* PWM_OUT mode
PUPSD bit, 6-68, 6-72
PWM_CLK clock, 15-23
PWM_OUT mode, 15-13 to 15-25, 15-47
control bit and register usage, 15-50
error prevention, 15-48
externally clocked, 15-22
PULSE_HI toggle mode, 15-18
stopping the timer, 15-23
Pxn bit, 14-23
Pxn both edges bit, 14-27
Pxn direction bit, 14-23

Pxn input enable bit, 14-24
Pxn polarity bits, 14-26
Pxn sensitivity bit, 14-27

Q

QCIF, 7-9
query semaphore, 21-3

R

RAF bit, 8-64, 8-68
RAISE 1 instruction, 19-7
RBC bit, 5-41, 5-100
RBSY bit, 10-43
RBSY flag, 10-22
RCKFE bit, 12-33, 12-54, 12-57
RCVDATA16[15:0] field, 11-47
RCVDATA8[7:0] field, 11-46
RCVFLUSH bit, 11-38, 11-39
RCVINTLEN bit, 11-38, 11-39
RCVSERV bit, 11-17, 11-43
RCVSERVM bit, 11-41
RCVSTAT[1:0] field, 11-16, 11-40
RDIV
  field, 6-40, 6-56
  field, equation for value, 6-60
RDIV[11:0] field, 6-60
RDTYPE[1:0] field, 12-28, 12-54, 12-56
read, asynchronous, 6-13
read access, for EBIU asynchronous memory controller, 6-20
read command, 6-35
read transfers to SDRAM banks, 6-52
read/write access bit, 3-10
read/write command, 6-51
real-time clock, *See* RTC
RE bit, 8-43, 8-51, 8-64, 8-70
REC bit, 9-44
receive buffer[7:0] field, 13-26
receive configuration registers (SPORTx_RCR1, SPORTx_RCR2), 12-52
receive data[15:0] field, 12-62
receive data[31:16] field, 12-62
receive data buffer[15:0] field, 10-44
receive FIFO, SPORT, 12-60
receive message lost interrupt, CAN, 9-25
receive message lost register 1 (CAN_RML1), 9-72
receive message lost register 2 (CAN_RML2), 9-72
receive message pending register 1 (CAN_RMP1), 9-71
receive message pending register 2 (CAN_RMP2), 9-71
reception error, SPI, 10-22
refresh, SDRAM, 6-32
REGAD[4:0] field, 8-76
register-based DMA, 5-12
registers
  See also registers by name
  diagram conventions, -lvi
  reset to zero, 19-21
  system, A-1
regulator controller, switching, 20-18
related documents, -l
remote frame handling register 1 (CAN_RFH1), 9-78
remote frame handling register 2 (CAN_RFH2), 9-79
remote frames, CAN, 9-19
remote wakeup frame filters, MAC, 8-35
REP bit, 5-6, 5-42, 5-100
request data control command, DMA, 5-37
request data urgent control command, DMA, 5-38
reserved SDRAM, 6-2
reset, effect on SPI, 10-18
RESET_DOUBLE bit, 19-7

$\overline{\text{RESET}}$ pin, 19-4
resets
  core and system, 19-8
  core-only software, 19-3, 19-7
  hardware, 19-8
  interrupts, 19-9
  software, 19-5
  system software, 19-3
  watchdog timer, 19-5
RESET_SOFTWARE bit, 19-7
reset vector, 19-1, 19-8
reset vector addresses, 19-2
RESET_WDOG bit, 17-4, 19-7
resource sharing, with semaphores, 21-2
restart control command, DMA, 5-36
restart or finish control command, receive, 5-39
restart or finish control command, transmit, 5-38
restrictions
  DMA control commands, 5-38
  DMA work unit, 5-27
RESVECTbit, 19-14
resynchronization, CAN, 9-11
RETI register, 19-9
RFHn bit, 9-78, 9-79
RFS pins, 12-19, 12-30
RFSR bit, 12-31, 12-54, 12-56
RFSx signal, 12-5
RLSBIT bit, 12-54, 12-56
RMII
  multiplexing, 8-5
  pins, 8-5, 14-3
  port H, 14-7
RMII_10 bit, 8-64, 8-65
RMII bit, 8-64, 8-66
RMLIF bit, 9-25, 9-48
RMLIM bit, 9-25, 9-47
RMLIS bit, 9-25, 9-47
RMLn bit, 9-72
RMPn bit, 9-71
ROM, 1-6, 6-1
round robin operation, MDMA, 5-52
routing of interrupts, 4-3
ROVF bit, 12-62, 12-63, 12-64
row activation, SDRAM, 6-32
row address, 6-64
row precharge, SDRAM, 6-33
RPOLC bit, 13-31, 13-32
RRFST bit, 12-55, 12-57
RSCLKx pins, 12-30
RSCLKx signal, 12-5
RSFSE bit, 12-12, 12-55, 12-57
RSPEN bit, 12-10, 12-53, 12-54, 12-55
RSTART bit, 11-34, 11-35
RSTC bit, 8-44, 8-124, 8-125
RTC, 1-20, 18-1 to 18-27
  alarm clock features, 18-2
  alarm feature, 18-26
  clock rate, 18-4
  clock requirements, 18-4
  code examples, 18-23
  counters, 18-1
  digital watch features, 18-1
  disabling prescaler, 18-5
  enabling prescaler, 18-4, 18-23
  interfaces, 18-2
  interrupt structure, 18-15
  prescaler, 18-1
  programming model, 18-6
  registers, table, 18-19
  state transitions, 18-16
  stopwatch, 18-2, 18-24
  synchronization, 18-5
  test mode, 18-5
RTC alarm register (RTC_ALARM), 18-22
RTC_ALARM (RTC alarm register), 18-19, 18-22

RTC_ICTL (RTC interrupt control register), 18-19, 18-20
RTC interrupt control register (RTC_ICTL), 18-20
RTC interrupt status register (RTC_ISTAT), 18-21
RTC_ISTAT (RTC interrupt status register), 18-19, 18-21
RTC_PREN bit, 18-4
RTC_PREN (prescaler enable register), 18-4, 18-19, 18-22
RTC_STAT (RTC status register), 18-19, 18-20
RTC status register (RTC_STAT), 18-20
RTC stopwatch count register (RTC_SWCNT), 18-21
RTC_SWCNT (RTC stopwatch count register), 18-19, 18-21
RTR bit, 9-53
RUVF bit, 12-62, 12-63, 12-64
RWKE bit, 8-35, 8-82, 8-83
RWKS[3:0] field, 8-82
RX_ACCEPT bit, 8-98, 8-103, 8-104, 8-107
RX_ADDR bit, 8-98, 8-101, 8-103, 8-105, 8-107
RX_ALIGN bit, 8-98, 8-102, 8-103, 8-106, 8-107
RX_ALIGN_CNT bit, 8-116, 8-118
RX_ALLF_CNT bit, 8-116, 8-118
RX_ALLO_CNT bit, 8-116, 8-118
RX_BROAD, RX_MULTI field, 8-98
RX_BROAD bit, 8-100, 8-103, 8-104, 8-107
RX_BROAD_CNT bit, 8-116, 8-118
RXCKS bit, 8-93, 8-94
RX_COMP bit, 8-98, 8-102, 8-103, 8-106, 8-107
RX_CRC bit, 8-98, 8-101, 8-103, 8-106, 8-107
RX_CTL bit, 8-98, 8-100, 8-103, 8-104, 8-107
RX DMA direction error detected, 8-41
RXDMAERR bit, 8-95, 8-96
RX_DMAO bit, 8-98, 8-101, 8-103, 8-105, 8-107
RXDWA bit, 8-18, 8-51, 8-93, 8-94
RXECNT[7:0] field, 9-85
RX_EQ64_CNT bit, 8-116, 8-118
RX_FCS_CNT bit, 8-116, 8-118
RX_FRAG bit, 8-98, 8-101, 8-103, 8-105, 8-107
RX frame status interrupt, 8-40
RX_FRLEN[10:0] field, 8-98, 8-103
RXFSINT bit, 8-95, 8-96
RX_GE1024_CNT bit, 8-116, 8-118
RX hold register, 12-62
RX_IRL_CNT bit, 8-116, 8-118
RX_LATE bit, 8-98, 8-100, 8-103, 8-105, 8-107
RX_LEN bit, 8-98, 8-101, 8-103, 8-106, 8-107
RX_LONG bit, 8-98, 8-102, 8-103, 8-106, 8-107
RX_LONG_CNT bit, 8-116, 8-118
RX_LOST_CNT bit, 8-116, 8-118
RX_LT1024_CNT bit, 8-116, 8-118
RX_LT128_CNT bit, 8-116, 8-118
RX_LT256_CNT bit, 8-116, 8-118
RX_LT512_CNT bit, 8-116, 8-118
RX_MACCTL_CNT bit, 8-118
RX_MULTI bit, 8-100, 8-103, 8-105, 8-107
RX_MULTI_CNT bit, 8-116, 8-118
RXNE bit, 12-64
RX_OCTET_CNT bit, 8-116, 8-118
RX_OK bit, 8-98, 8-102, 8-103, 8-106, 8-107
RX_OK_CNT bit, 8-116, 8-118
RX_OPCODE_CNT bit, 8-116, 8-118

RX_ORL_CNT bit, 8-116, 8-118
RX_PAUSE_CNT bit, 8-116, 8-118
RX_PHY bit, 8-98, 8-100, 8-103, 8-105, 8-107
RX_RANGE bit, 8-98, 8-100, 8-103, 8-105, 8-107
RXREQ signal, 13-7
RXS bit, 10-28, 10-43
RXSE bit, 12-55, 12-57
RX_SHORT_CNT bit, 8-116, 8-118
RX_TYPE bit, 8-98, 8-99, 8-103, 8-104, 8-107
RX_TYPED_CNT bit, 8-116, 8-118
RX_UCTL bit, 8-98, 8-99, 8-103, 8-104, 8-107
RX_UNI_CNT bit, 8-116, 8-118
RX_VLAN1 bit, 8-98, 8-99, 8-103, 8-104, 8-107
RX_VLAN2 bit, 8-98, 8-99, 8-103, 8-104, 8-107

S

SA0 bit, 9-85
SA10 pin, 6-55
SADDR[6:0] field, 11-33
SAM bit, 9-46
SAMPLE/PRELOAD instruction, B-6
sampling, CAN, 9-11
sampling clock period, UART, 13-8
sampling edge, SPORT, 12-33
SAV, 7-32
SAV signal, 7-6
SB bit, 13-21
scale value[7:0] field, 16-7
scaling, of core timer, 16-7
scan paths, B-4
SCCB bit, 11-30
scheduling, memory DMA, 5-51
SCKE pin, 6-53
SCK signal, 10-4, 10-15, 10-17, 10-19
SCLK, 2-4, 20-5
  changing frequency, 6-45
  derivation, 20-2
  disabling, 20-22
  EBIU, 6-2
  status by operating mode (table), 20-8
SCLOVR bit, 11-34
SCL pin, 11-4
SCLSEN bit, 11-12, 11-38
SCOMP bit, 11-19, 11-43
SCOMPM bit, 11-41, 11-43
scratch[7:0] field, 13-31
scratchpad memory, and booting, 19-12
SCTLE bit, 6-53, 6-54, 6-57, 6-67, 6-68, 6-69, 6-74
SDAOVR bit, 11-34, 11-35
SDA pin, 11-4
SDASEN bit, 11-13, 11-38
SDC, 1-9, 6-4, 6-25
  address mapping, 6-27
  address muxing, 6-42
  architecture, 6-40
  code examples, 6-79
  commands, 6-47
  component configurations, 6-26
  configuration, 6-56
  core and DMA arbitration, 6-44
  core transfers to SDRAM, 6-79
  disabled CLKOUT, 6-81
  features, 6-25
  initialization, 6-82
  no burst mode, 6-41
  operation, 6-39
  pin state during commands, 6-48
  powerup latency, 6-56
  registers, 6-60
  SA10 pin, 6-55
  self-refresh mode, 6-81
  transfers to SDRAM using byte mask, 6-80

SDC commands, 6-47
SDCI bit, 6-78
SDEASE bit, 6-78, 6-79
SDIR bit, 11-15, 11-33
SDPUA bit, 6-78, 6-79
SDQM[1:0] encodings during writes, 6-52
SDRAM, 1-6
  address connections, 6-42
  address mapping, 6-27
  auto-refresh command, 6-35, 6-52
  bank activate command, 6-35
  banks, 3-8
  bank size, 6-2
  burst length, 6-33
  burst type, 6-34
  and busmastership, 6-8
  CAS latency, 6-34
  clock enables, setting, 6-67
  column read/write, 6-32
  commands, 6-34
  configuration, 6-25
  ERMS command, 6-34
  external bank, 6-33
  external memory, 6-27
  interface signals, 6-30
  internal banks, 6-29
  internal memory banks, 6-33
  latency, 6-73
  memory banks, 6-3
  memory size, 6-33
  memory space, 6-2
  MRS command, 6-34
  multibank operation, 6-43
  no operation command, 6-54
  operation parameters, initializing, 6-49
  parallel connection, 6-29
  performance, 6-31
  power-up sequence, 6-45
  powerup sequence, 6-73
  precharge all command, 6-35

SDRAM *(continued)*
  precharge command, 6-35
  read command, 6-35
  read transfers, 6-52
  read/write command, 6-51
  refresh, 6-32
  reserved, 6-2
  row activation, 6-32
  row precharge, 6-33
  self-refresh mode, 6-35, 6-36, 6-53
  shared, 6-46
  sharing external, 6-72
  size configuration, 6-62
  sizes supported, 3-8, 6-25
  smaller than 16M byte, 6-65
  specification of system, 6-29
  stall cycles, 6-32
  start address, 6-2
  system block diagram, 6-58, 6-59
  timing specs, 6-36
  write command, 6-35
SDRAM controller, *See* SDC
SDRAM control status register (EBIU_SDSTAT), 6-78
SDRAM memory, 3-2
SDRAM memory bank control register (EBIU_SDBCTL), 6-62, 6-63
SDRAM memory global control register (EBIU_SDGCTL), 6-67, 6-68
SDRAM refresh rate control register (EBIU_SDRRC), 6-60
SDRS bit, 6-56, 6-78, 6-79
SDSRA bit, 6-78
seconds (1 Hz) event flag bit, 18-21
seconds (1Hz) interrupt enable bit, 18-20
seconds[5:0] field, 18-20, 18-22
self-refresh command, 6-53
self-refresh mode, 6-35, 6-36, 6-53, 6-81
  entering, 6-53
  exiting, 6-54

semaphores, 21-2
  example code, 21-3
  query, 21-3
SEN bit, 11-31, 11-32
SER bit, 9-85
serial clock divide modulus[15:0] field, 12-64, 12-65
serial clock frequency, 10-20
serial communications, 13-4
serial data transfer, 12-3
serial peripheral interface, *See* SPI
serial scan paths, B-4
SERR bit, 11-19, 11-43
SERRM bit, 11-41, 11-42
set index[5:0] field, 3-10
SET_PHYAD macro, 8-130
set Pxn bit, 14-25
set Pxn interrupt A enable bit, 14-29
set Pxn interrupt B enable bit, 14-30
SET_REGAD macro, 8-130
setup
  for EBIU asynchronous memory controller, 6-20
  SDRAM clock enables, 6-67
shared interrupts, 4-7
SIC_IAR0 (system interrupt assignment register 0), 4-18
SIC_IAR1 (system interrupt assignment register 1), 4-18
SIC_IAR2 (system interrupt assignment register 2), 4-19
SIC_IAR3 (system interrupt assignment register 3), 4-19
SIC_IMASK (system interrupt mask register), 4-7, 4-20
SIC_ISR (system interrupt status register), 4-8, 4-21
SIC_IWR (system interrupt wakeup-enable register), 4-9, 4-22
signal integrity, 21-8
sine wave input, 1-22
SingleCollisionFrames register, 8-60
single precharge command, 6-52
single pulse generation, timer, 15-15
single shot transmission, CAN, 9-15
SINIT bit, 11-19, 11-43
SINITM bit, 11-41, 11-43
SIZE bit, 10-18, 10-41
size of accesses, timer registers, 15-38
SJW[1:0] field, 9-11, 9-46
SKIP_EN bit, 7-26, 7-27
SKIP_EO bit, 7-27, 7-28
slaves
  EBIU, 6-5
  PAB, 2-6
slave select, SPI, 10-42
slave SPI device, 10-6
sleep mode, 1-23, 20-9
  CAN, 9-38
  MAC operation in, 8-33
SLEN[4:0] field, 12-49, 12-50, 12-55, 12-56
  restrictions, 12-28
  word length formula, 12-28
small descriptor mode, DMA, 5-17
small model mode, DMA, 5-76
SMR bit, 9-43
software interrupts, 4-4
software management of DMA, 5-55
software reset, 19-5
software reset, and CAN, 9-12
software reset register (SWRST), 19-7
software watchdog timer, 1-21, 17-1
source channels, memory DMA, 5-9
SOVF bit, 11-18, 11-43
SOVFM bit, 11-41, 11-42
SPE bit, 10-19, 10-41
speculative read, and MAC, 8-44

SPI, 1-18, 10-1 to 10-53
beginning and ending transfers, 10-28
bit mapping to port pins, 10-9
block diagram, 10-3
booting memory detection, 19-44
clock phase, 10-14, 10-15, 10-19
clock polarity, 10-14, 10-19
clock signal, 10-3, 10-19
code examples, 10-45
compatible peripherals, 10-2
data corruption, avoiding, 10-17
data interrupt, 10-23
data transfer, 10-17
detecting transfer complete, 10-20
and DMA, 10-12
DMA initialization, 10-49
DMA transfers, 10-48
effect of reset, 10-18
error interrupt, 10-23
error signals, 10-20 to 10-23
features, 10-1
general operation, 10-17 to 10-27
initialization, 10-45
internal interfaces, 10-12
interrupt outputs, 10-23
interrupts, 10-47
master boot from flash, 19-28
master boot mode, 19-42
master mode, 10-17, 10-24
master mode DMA operation, 10-30
mode fault error, 10-21
multiple slave systems, 10-10
pins, 14-2, 14-3
port F, 10-4
port J, 14-8
reception error, 10-22
registers, table, 10-40
SCK signal, 10-4
slave boot mode, 19-48
slave device, 10-6

SPI *(continued)*
slave mode, 10-18, 10-26
slave mode DMA operation, 10-32
slave select enable setup, 10-8
slave-select function, 10-42
slave transfer preparation, 10-27
SPI_FLG mapping to port pins, 10-43
starting DMA transfer, 10-51
starting transfer, 10-46
stopping, 10-48
stopping DMA transfers, 10-51
switching between transmit and receive, 10-29
timing, 10-7
transfer formats, 10-14 to 10-15
transfer initiate command, 10-24, 10-25
transfer modes, 10-25
transfer protocol, 10-15, 10-16
transmission error, 10-22
transmission/reception errors, 10-20
transmit collision error, 10-22
using DMA, 10-12
word length, 10-18
SPI baud rate registers (SPI_BAUD), 10-20, 10-40
SPI_BAUD (SPI baud rate registers), 10-20, 10-40
SPI_BAUD values, 10-20
SPI control register (SPI_CTL), 10-18, 10-40, 10-41
SPI_CTL (SPI control register), 10-4, 10-18, 10-40, 10-41
SPIF bit, 10-11, 10-28, 10-43
SPI flag register (SPI_FLG), 10-8, 10-40, 10-42
SPI_FLG bit, 10-9
SPI_FLG (SPI flag register), 10-8, 10-10, 10-40, 10-42
SPI_RDBR shadow[15:0] field, 10-44

SPI RDBR shadow register (SPI_SHADOW), 10-14, 10-40, 10-44
SPI_RDBR (SPI receive data buffer register), 10-14, 10-40, 10-44
SPI receive data buffer register (SPI_RDBR), 10-14, 10-40, 10-44
SPI_SHADOW (SPI RDBR shadow register), 10-14, 10-40, 10-44
SPI slave select, 10-42
SPISS signal, 10-6, 10-10, 10-15
SPI_STAT (SPI status register), 10-20, 10-40, 10-43
SPI status register (SPI_STAT), 10-20, 10-40, 10-43
SPI_TDBR (SPI transmit data buffer register), 10-13, 10-40, 10-44
SPI transmit data buffer register (SPI_TDBR), 10-13, 10-40, 10-44
SPORT, 1-16, 12-1 to 12-78
  active low vs. active high frame syncs, 12-33
  channels, 12-15
  clock, 12-30
  clock frequency, 12-26, 12-64
  clock rate, 12-2
  clock rate restrictions, 12-28
  clock recovery control, 12-26
  companding, 12-29
  configuration, 12-11
  data formats, 12-28
  data word formats, 12-57
  delay when enabled, 12-11
  disabling, 12-11
  DMA data packing, 12-24
  enable/disable, 12-10
  enabling functionality, 12-4
  enabling multichannel mode, 12-18
  framed serial transfers, 12-31
  framed vs. unframed, 12-31

SPORT *(continued)*
  frame sync, 12-32, 12-35
  frame sync frequencies, 12-26
  framing signals, 12-30
  general operation, 12-10
  H.100 standard protocol, 12-25
  initialization code, 12-56
  internal memory access, 12-38
  internal vs. external frame syncs, 12-32
  late frame sync, 12-18
  modes, 12-11
  moving data to memory, 12-38
  multichannel frame, 12-20
  multichannel operation, 12-15 to 12-25
  multichannel transfer timing, 12-17
  multiplexed pins, 12-4
  multiplexed with PPI, 14-6
  PAB error, 12-39
  packing data, multichannel DMA, 12-24
  pins, 12-4, 14-2, 14-3
  port connection, 12-7
  port G, 12-4, 14-6
  port J, 12-4, 14-8
  receive and transmit functions, 12-4
  receive clock signal, 12-30
  receive FIFO, 12-60
  receive word length, 12-61
  register writes, 12-47
  RX hold register, 12-62
  sampling edge, 12-33
  selecting bit order, 12-28
  serial data communication protocols, 12-1
  shortened active pulses, 12-11
  signals, 12-5
  single clock for both receive and transmit, 12-30
  single word transfers, 12-38
  stereo serial connection, 12-9
  stereo serial frame sync modes, 12-18

SPORT *(continued)*
stereo serial operation, 12-12
support for standard protocols, 12-25
termination, 12-9
timing, 12-39
transmit clock signal, 12-30
transmitter FIFO, 12-58
transmit word length, 12-58
TX hold register, 12-59
TX interrupt, 12-59
unframed data flow, 12-31
unpacking data, multichannel DMA, 12-24
window offset, 12-22
word length, 12-28
SPORT error interrupt, 12-38
SPORT multichannel transmit select registers (SPORTx_MTCSn), 12-24
SPORT RX interrupt, 12-38, 12-62
SPORT TX interrupt, 12-38
SPORTx_CHNL (SPORTx current channel register), 12-67
SPORTx current channel register (SPORTx_CHNL), 12-67
SPORTx_MCMCn (SPORTx multichannel configuration registers), 12-66
SPORTx_MRCSn (SPORTx multichannel receive select registers), 12-68
SPORTx_MTCSn (SPORTx multichannel transmit select registers), 12-70
SPORTx multichannel configuration registers (SPORTx_MCMCn), 12-66
SPORTx multichannel receive select registers (SPORTx_MRCSn), 12-23, 12-24, 12-68
SPORTx multichannel transmit select registers (SPORTx_MTCSn), 12-23, 12-70
SPORTx_RCLKDIV (SPORTx receive serial clock divider registers), 12-64
SPORTx_RCR1 (SPORTx receive configuration registers), 12-52
SPORTx_RCR2 (SPORTx receive configuration register), 12-55
SPORTx_RCR2 (SPORTx receive configuration registers), 12-52
SPORTx receive configuration 2 registers (SPORTx_RCR2), 12-55
SPORTx receive data registers (SPORTx_RX), 12-19, 12-60
SPORTx receive frame sync divider registers (SPORTx_RFSDIV), 12-65
SPORTx receive serial clock divider registers (SPORTx_RCLKDIV), 12-64
SPORTx_RFSDIV (SPORTx receive frame sync divider registers), 12-65
SPORTx_RX (SPORTx receive data registers), 12-60
SPORTx_STAT (SPORTx status registers), 12-63
SPORTx status registers (SPORTx_STAT), 12-63
SPORTx_TCLKDIV (SPORTx transmit serial clock divider registers), 12-64
SPORTx_TCR1 (transmit configuration registers), 12-47
SPORTx_TCR2 (transmit configuration registers), 12-47
SPORTx_TFSDIV (SPORTx transmit frame sync divider registers), 12-65
SPORTx transmit data registers (SPORTx_TX), 12-19, 12-37, 12-58
SPORTx transmit frame sync divider registers (SPORTx_TFSDIV), 12-65

SPORTx transmit serial clock divider registers (SPORTx_TCLKDIV), 12-64
SPORTx_TX (SPORTx transmit data registers), 12-58
SRAM, 1-6, 6-1
  interface, 21-5
SRAM ADDR[13:12] field, 3-10
SRFS bit, 6-53, 6-68, 6-69, 6-74
SRS bit, 9-43
SSEL[3:0] field, 2-4, 20-5, 20-25
STABUSY bit, 8-76, 8-77
STADATA[15:0] field, 8-78
STADISPRE bit, 8-76, 8-77
STAIE bit, 8-76
stall cycles, SDRAM, 6-32
STAOP bit, 8-76, 8-77
start address registers
  (DMAx_START_ADDR), 5-82
  (MDMA_yy_START_ADDR), 5-82
state transitions, RTC, 18-16
station management read transfer, MAC, 8-49
station management transfer done, 8-41
station management write transfer, MAC, 8-49
STATUS[1:0] field, 13-29
STB bit, 13-21
STDVAL bit, 11-31, 11-32
stereo serial data, 12-2
stereo serial device, SPORT connection, 12-9
stereo serial frame sync modes, 12-18
stereo serial operation, SPORT, 12-12
STI instruction, 20-17
STMDONE bit, 8-95
STOP bit, 11-36
STOPCK bit, 20-26
stop mode, DMA, 5-13, 5-75
stopping DMA transfers, 5-31
stopwatch count[15:0] field, 18-21
stopwatch event flag bit, 18-21
stopwatch function, RTC, 18-2
stopwatch interrupt enable bit, 18-20
STP bit, 13-21
streams, memory DMA, 5-10
subbank access[1:0] field, 3-10
subbanks
  L1 data memory, 3-7
  L1 instruction memory, 3-6
supervisor mode, 19-9
support, technical or customer, -xlvii
suspend mode, CAN, 9-37
switching frequency values, 20-20
switching regulator controller, 20-18
SWRST (software reset register), 19-7
SYNC bit, 5-28, 5-30, 5-66, 5-74, 5-77, 13-17
synchronization
  interrupt-based methods, 5-56
  of descriptor queue, 5-62
  of DMA, 5-55 to 5-65
synchronized transition, DMA, 5-30
synchronous serial data transfer, 12-3
synchronous serial ports, *See* SPORT
SYSCR (system reset configuration register), 19-4, 19-5
SYSRST bit, 19-7
system and core event mapping (table), 4-4
system clock, changing during runtime, 6-44
system clock (SCLK), 20-2
  managing, 21-2
system design, 21-1 to 21-14
  high frequency considerations, 21-8
  recommendations and suggestions, 21-9
  recommended reading, 21-13
system interrupt assignment register 0 (SIC_IAR0), 4-18

system interrupt assignment register 1 (SIC_IAR1), 4-18
system interrupt assignment register 2 (SIC_IAR2), 4-19
system interrupt assignment register 3 (SIC_IAR3), 4-19
system interrupt controller (SIC), 4-1, 4-2
  registers, 4-16
system interrupt mask register (SIC_IMASK), 4-7, 4-20
system interrupt processing, 4-14
system interrupts, 4-1, 4-2
system interrupt status register (SIC_ISR), 4-8, 4-21
system interrupt wakeup-enable register (SIC_IWR), 4-9, 4-22
system peripheral clock, *See* SCLK
system peripherals, 1-1
system reset, 19-1 to 19-60
SYSTEM_RESET[2:0] field, 19-7
system reset configuration register (SYSCR), 19-4, 19-5
system software reset, 19-3
SZ bit, 10-27, 10-41

T

TACIx pins, 15-5, 15-34
TACLKx pins, 15-5
TAn bit, 9-77
TAP registers
  boundary-scan, B-6
  bypass, B-6
  instruction, B-2, B-4
TAP (test access port), B-1, B-2
  controller, B-2
TAUTORLD bit, 16-3, 16-4
TCKFE bit, 12-33, 12-48, 12-52
TCNTL (core timer control register), 16-3, 16-4
TCOUNT (core timer count register), 16-2, 16-5
TCP/IP-style checksums, MAC, 8-21
TCSR bit, 6-68, 6-77
TDA bit, 9-21, 9-78
TDM interfaces, 12-3
TDPTR[4:0] field, 9-78
TDR bit, 9-21, 9-78
TDTYPE[1:0] field, 12-28, 12-48, 12-50
TE bit, 8-43, 8-51, 8-64, 8-68
technical support, -xlvii
temporary mailbox disable feature register (CAN_MBTD), 9-78
TEMT bit, 13-6, 13-24, 13-25
termination, DMA, 5-31
terminations, SPORT pin/line, 12-9
test access port (TAP), B-1, B-2
  controller, B-2
test clock (TCK), B-6
test features, B-1 to B-6
testing circuit boards, B-1, B-5
test-logic-reset state, B-4
test point access, 21-12
TESTSET instruction, 2-10, 21-3
TFS pins, 12-30, 12-37
TFSR bit, 12-31, 12-48, 12-51
TFS signal, 12-20
TFSx signal, 12-5
THRE bit, 13-11, 13-24, 13-25
THRE flag, 13-5, 13-16
throughput
  DAB, 2-10
  DMA, 5-45
  from DMA system, 5-44
  general-purpose ports, 14-9
  SPORT, 12-7
TIMDISx bit, 15-39, 15-40
time-division-multiplexed (TDM) mode, 12-15
  See also SPORT, multichannel operation

TIMENx bit, 15-38, 15-39
timer configuration registers (TIMERx_CONFIG), 15-43, 15-44
timer counter[15:0] field, 15-46
timer counter[31:16] field, 15-46
timer counter registers (TIMERx_COUNTER), 15-45, 15-46
TIMER_DISABLE bit, 15-50
timer disable register (TIMER_DISABLE), 15-39, 15-40
TIMER_DISABLE (timer disable register), 15-7, 15-39, 15-40
TIMER_ENABLE bit, 15-50
timer enable register (TIMER_ENABLE), 15-38, 15-39
TIMER_ENABLE (timer enable register), 15-7, 15-38, 15-39
timer period[15:0] field, 15-48
timer period[31:16] field, 15-48
timer period registers (TIMERx_PERIOD), 15-47, 15-48
timers, 1-18, 15-1 to 15-61
  core, 16-1 to 16-8
  EXT_CLK mode, 15-34 to 15-36
  pins, 14-2
  port F, 14-4
  port H, 14-7
  UART, 13-4
  watchdog, 1-21, 17-1 to 17-10
  WDTH_CAP mode, 13-15
timer status register (TIMER_STATUS), 15-40, 15-42
TIMER_STATUS (timer status register), 15-40, 15-42
timer width[15:0] field, 15-49
timer width[31:16] field, 15-49
timer width registers (TIMERx_WIDTH), 15-47, 15-49
TIMERx_CONFIG (timer configuration registers), 15-43, 15-44
TIMERx_COUNTER (timer counter registers), 15-6, 15-45, 15-46
TIMERx_PERIOD (timer period registers), 15-47, 15-48
TIMERx_WIDTH (timer width registers), 15-47, 15-49
time stamps, CAN, 9-20
TIMILx bits, 15-6, 15-42
timing
  auto-refresh, 6-60
  memory DMA, 5-48
  multichannel transfer, 12-17
  peripherals, 2-4
  SPI, 10-7
timing examples, for SPORTs, 12-39
timing parameters, CAN, 9-11
timing specs, SDRAM, 6-36
TIMOD[1:0] field, 10-18, 10-23, 10-25, 10-41
TIN_SEL bit, 15-34, 15-44, 15-51
TINT bit, 16-3, 16-4
TLSBIT bit, 12-48, 12-50
TMODE[1:0] field, 15-13, 15-44, 15-50
TMPWR bit, 16-3, 16-4
TMRCLK input, 15-5
TMREN bit, 16-3, 16-4
TMR pin, 15-51
TMRx pins, 15-4, 15-18, 15-34
TOGGLE_HI bit, 15-44, 15-51
TOGGLE_HI mode, 15-18
toggle Pxn bit, 14-26
toggle Pxn interrupt A enable bit, 14-33
toggle Pxn interrupt B enable bit, 14-34
tools, development, 1-29
TOVF bit, 12-59, 12-63, 12-64
TOVF_ERRx bits, 15-6, 15-10, 15-17, 15-42, 15-43, 15-52

TPERIOD (core timer period register), 16-6
TPOLC bit, 13-31, 13-32
traffic control, DMA, 5-48 to 5-55
transfer count register (PPI_COUNT), 7-34, 7-35
transfer frame protocol, MAC, 8-8
transfer initiate command, 10-24, 10-25
transfer initiation from SPI master, 10-25
transfer rate
  memory DMA channels, 5-45
  peripheral DMA channels, 5-45
transitions
  continuous DMA, 5-27
  DMA work unit, 5-27
  operating mode, 20-11, 20-12
  synchronized DMA, 5-27
transmission acknowledge register 1 (CAN_TA1), 9-77
transmission acknowledge register 2 (CAN_TA2), 9-77
transmission error, SPI, 10-22
transmission request reset register 1 (CAN_TRR1), 9-75
transmission request reset register 2 (CAN_TRR2), 9-75
transmission request set register 1 (CAN_TRS1), 9-74
transmission request set register 2 (CAN_TRS2), 9-74
transmit clock, serial (TSCLKx) pins, 12-30
transmit collision error, SPI, 10-22
transmit configuration registers (SPORTx_TCR1 and SPORTx_TCR2), 12-47
transmit data[15:0] field, 12-60
transmit data[31:16] field, 12-60
transmit data buffer[15:0] field, 10-44
transmit hold[7:0] field, 13-26
t_{RAS}, 6-36
TRAS[3:0] field, 6-68, 6-70
t_{RC}, 6-38
t_{RCD}, 6-37
TRCD[2:0] field, 6-68, 6-71
t_{REF}, 6-39
t_{REFI}, 6-39
t_{RFC}, 6-38
TRFST bit, 12-49, 12-52
triggering DMA transfers, 5-65
TRM bit, 9-44
t_{RP}, 6-38
TRP[2:0] field, 6-68, 6-71
t_{RRD}, 6-37
TRRn bit, 9-75
TRSn bit, 9-74
TRUNx bits, 15-23, 15-40, 15-42, 15-52
TSCALE (core timer scale register), 16-7
TSCLKx signal, 12-5
TSEG1[3:0] field, 9-10, 9-46
TSEG2[2:0] field, 9-10, 9-46
TSFSE bit, 12-12, 12-49, 12-52
TSPEN bit, 12-10, 12-47, 12-48, 12-49
TSV[15:0] field, 9-57
TUVF bit, 12-37, 12-59, 12-63, 12-64
TWI, 1-11, 11-1 to 11-59
  block diagram, 11-2
  bus arbitration, 11-7
  clock generation, 11-6
  electrical specifications, 11-59
  fast mode, 11-10
  features, 11-1
  general call address, 11-9
  general setup, 11-20
  I^2C compatibility, 1-11
  master boot from flash, 19-29
  master boot mode, 19-53
  master mode clock setup, 11-22
  master mode receive, 11-23
  master mode transmit, 11-22

TWI *(continued)*
peripheral interface, 11-5
pins, 11-4
port J, 14-8
repeated start condition, 11-24
slave boot mode, 19-55
slave mode operation, 11-20
start and stop conditions, 11-8
synchronization, 11-6
transfer protocol, 11-6
TWI_ENA bit, 11-30
TWI FIFO control register (TWI_FIFO_CTL), 11-38
TWI_FIFO_CTL (TWI FIFO control register), 11-38
TWI FIFO receive data double byte register (TWI_RCV_DATA16), 11-46
TWI FIFO receive data single byte register (TWI_RCV_DATA8), 11-45
TWI_FIFO_STAT (TWI FIFO status register), 11-16
TWI FIFO status register (TWI_FIFO_STAT), 11-16
TWI FIFO transmit data double byte register (TWI_XMT_DATA16), 11-44
TWI FIFO transmit data single byte register (TWI_XMT_DATA8), 11-43
TWI interrupt mask register (TWI_INT_MASK), 11-40
TWI interrupt status register (TWI_INT_STAT), 11-17
TWI_INT_MASK (TWI interrupt mask register), 11-40
TWI_INT_STAT (TWI interrupt status register), 11-17
TWI_MASTER_ADDR (TWI master mode address register), 11-37
TWI_MASTER_CTL (TWI master mode control register), 11-34
TWI master mode address register (TWI_MASTER_ADDR), 11-37
TWI master mode control register (TWI_MASTER_CTL), 11-34
TWI master mode status register (TWI_MASTER_STAT), 11-12
TWI_MASTER_STAT (TWI master mode status register), 11-12
TWI pins, 14-3
TWI_RCV_DATA16 (TWI FIFO receive data double byte register), 11-46
TWI_RCV_DATA8 (TWI FIFO receive data single byte register), 11-45
TWI_SLAVE_ADDR (TWI slave mode address register), 11-33
TWI_SLAVE_CTL (TWI slave mode control register), 11-31
TWI slave mode address register (TWI_SLAVE_ADDR), 11-33
TWI slave mode control register (TWI_SLAVE_CTL), 11-31
TWI slave mode status register (TWI_SLAVE_STAT), 11-15, 11-33
TWI_SLAVE_STAT (TWI slave mode status register), 11-15, 11-33
TWI_XMT_DATA16 (TWI FIFO transmit data double byte register), 11-44
TWI_XMT_DATA8 (TWI FIFO transmit data single byte register), 11-43
two-dimensional DMA, 5-14
two-wire interface, *See* TWI
t_{WR}, 6-37
TWR[1:0] field, 6-68, 6-72
TX_ABORTC_CNT bit, 8-120, 8-122
TX_ABORT_CNT bit, 8-120, 8-122
TX_ALLF_CNT bit, 8-120, 8-122
TX_ALLO_CNT bit, 8-120, 8-122

TX_BROAD bit, 8-110, 8-112, 8-113, 8-114
TX_BROAD_CNT bit, 8-120, 8-122
TX_CCNT[3:0] field, 8-108, 8-110
TXCOL bit, 10-43
TXCOL flag, 10-22
TX_COMP bit, 8-108, 8-111, 8-112, 8-114
TX_CRS bit, 8-108, 8-109, 8-112, 8-113, 8-114
TX_CRS_CNT bit, 8-120, 8-122
TX_DEFER bit, 8-108, 8-109, 8-112, 8-113, 8-114
TX_DEFER_CNT bit, 8-120, 8-122
TX DMA direction error detected, 8-41
TXDMAERR bit, 8-30, 8-95, 8-96
TX_DMAU bit, 8-108, 8-110
TXDWA bit, 8-27, 8-94
TXE bit, 10-22, 10-43
TXECNT[7:0] field, 9-85
TX_ECOLL bit, 8-108, 8-111, 8-112, 8-113, 8-114
TX_EDEFER bit, 8-108, 8-110, 8-112, 8-113, 8-114
TX_EQ64_CNT bit, 8-120, 8-122
TX_ER pin, 8-6
TX_EXDEF_CNT bit, 8-120, 8-122
TXF bit, 12-59, 12-63, 12-64
TX frame status interrupt, 8-41
TX_FRLEN[10:0] field, 8-108
TXFSINT bit, 8-95, 8-96
TX_GE1024_CNT bit, 8-120, 8-122
TX hold register, 12-59
TXHRE bit, 12-64
TX_LATE bit, 8-108, 8-111, 8-112, 8-113, 8-114
TX_LATE_CNT bit, 8-120, 8-122
TX_LOSS bit, 8-108, 8-109, 8-112, 8-114
TX_LOST_CNT bit, 8-120, 8-122
TX_LT1024_CNT bit, 8-120, 8-122
TX_LT128_CNT bit, 8-120, 8-122
TX_LT256_CNT bit, 8-120, 8-122
TX_LT512_CNT bit, 8-120, 8-122
TX_MACCTL_CNT bit, 8-120, 8-122
TX_MACE bit, 8-112, 8-113, 8-114
TX_MCOLL_CNT bit, 8-120, 8-122
TX_MULTI bit, 8-110, 8-112, 8-113, 8-114
TX Multicast, TX Broadcast[1:0] field, 8-108
TX_MULTI_CNT bit, 8-120, 8-122
TX_OCTET_CNT bit, 8-120, 8-122
TX_OK bit, 8-108, 8-111, 8-112, 8-114
TX_OK_CNT bit, 8-120, 8-122
TXREQ signal, 13-6
TX_RETRY bit, 8-108, 8-109, 8-112, 8-114
TXS bit, 10-28, 10-43
TX_SCOLL_CNT bit, 8-120, 8-122
TXSE bit, 12-49, 12-52
t_{XSR}, 6-38
TX_UNI_CNT bit, 8-120, 8-122
type definitions, MAC, 8-126
TypedFramesReceived register, 8-59

U

UART, 1-19, 13-1 to 13-41
  assigning interrupt priority, 13-12
  autobaud detection, 13-13, 13-35, 15-34
  baud rate, 13-6
  baud rate examples, 13-12
  bit rate detection, 15-5
  bit rate examples, 13-12
  bitstream, 13-5
  block diagram, 13-3
  booting, 13-13
  character transmission, 13-36
  clearing interrupt latches, 13-29

UART *(continued)*
clock, 13-12
clock rate, 2-4
code examples, 13-32
data words, 13-4
divisor reset, 13-30
DMA channels, 13-17
DMA mode, 13-17
errors during reception, 13-7
external interfaces, 13-2
features, 13-1
glitch filtering, 13-9
initialization, 13-32
internal interfaces, 13-3
interrupt channels, 13-27
interrupt conditions, 13-28
interrupts, 13-10
IrDA mode, 13-2
IrDA receiver, 13-9
IrDA receiver pulse detection, 13-10
IrDA transmit pulse, 13-8
IrDA transmitter, 13-8
and ISRs, 13-16
loopback mode, 13-23
mixing modes, 13-18
modem status, 13-2
non-DMA interrupt operation, 13-38
non-DMA mode, 13-15
pins, 14-2
polling, 13-37
port F, 14-4
receive operation, 13-6
receive sampling window, 13-9
registers, table, 13-20
sampling clock period, 13-8
slave boot mode, 19-56
standard, 13-1
string transmission, 13-36
switching from DMA to non-DMA, 13-18

UART *(continued)*
switching from non-DMA to DMA, 13-19
and system DMA, 13-27
timers, 13-4
transmission, 13-5
transmission SYNC bit use, 13-39
UART divisor latch high byte registers (UARTx_DLH), 13-30
UART divisor latch low byte registers (UARTx_DLL), 13-30
UART global control registers (UARTx_GCTL), 13-31
UART interrupt enable registers (UARTx_IER), 13-27, 13-28
UART interrupt identification registers (UARTx_IIR), 13-12, 13-29
UART line control registers (UARTx_LCR), 13-21
UART line status registers (UARTx_LSR), 13-24
UART modem control registers (UARTx_MCR), 13-23
UART receive buffer registers (UARTx_RBR), 13-6, 13-26
UART scratch registers (UARTx_SCR), 13-31
UART transmit holding registers (UARTx_THR), 13-26
UARTx_DLH (UART divisor latch high byte registers), 13-20, 13-30
UARTx_DLL (UART divisor latch low byte registers), 13-20, 13-30
UARTx_GCTL (UART global control registers), 13-20, 13-31
UARTx_IER (UART interrupt enable registers), 13-20, 13-27, 13-28
UARTx_IIR (UART interrupt identification registers), 13-12, 13-20, 13-29

UARTx_LCR (UART line control registers), 13-20, 13-21
UARTx_LSR (UART line status registers), 13-20, 13-24
UARTx_MCR (UART modem control registers), 13-20, 13-23
UARTx_RBR (UART receive buffer registers), 13-6, 13-20, 13-26
UARTx_SCR (UART scratch registers), 13-20, 13-31
UARTx_THR (UART transmit holding registers), 13-5, 13-20, 13-26
UCCNF[3:0] field, 9-26, 9-83
UCCNT[15:0] field, 9-84
UCCT bit, 9-83
UCE bit, 9-83
UCEIF bit, 9-24, 9-48
UCEIM bit, 9-24, 9-47
UCEIS bit, 9-24, 9-47
UCEN bit, 13-6, 13-12, 13-30, 13-31, 13-32
UCRC[15:0] field, 9-84
UCRC bit, 9-83
UIAIF bit, 9-25, 9-48
UIAIM bit, 9-25, 9-47
UIAIS bit, 9-25, 9-47
UNDR bit, 7-31, 7-33
unframed/framed, serial data, 12-31
UnicastFramesReceivedOK register, 8-56
UnicastFramesXmittedOK register, 8-62
universal asynchronous receiver/transmitter, *See* UART
universal counter, CAN, 9-26
universal counter configuration mode register (CAN_UCCNF), 9-83
universal counter exceeded interrupt, CAN, 9-24
universal counter register (CAN_UCCNT), 9-84
universal counter reload/capture register (CAN_UCRC), 9-84
unpopulated memory, 6-10
UnsupportedOpcodesReceived register, 8-58
unused pins, 21-12
urgent DMA transfers, 5-50
user mode, 19-9
UTE bit, 5-43, 5-100
UTHE[15:0] field, 5-104

V

VCO, multiplication factors, 20-4
VDK, 1-30
vertical blanking, 7-7
vertical blanking interval only submode, 7-10, 7-11
video data transfers using PPI, 7-39
video frame partitioning, 7-8
VisualDSP++, 1-29, 19-11
  debugger, 1-29
VLAN1TAG[15:0] field, 8-81
VLAN2TAG[15:0] field, 8-81
VLEV[3:0] field, 20-21, 20-27
voltage, 20-18
  changing, 20-21
  control, 20-7
  dynamic control, 20-18
voltage controlled oscillator (VCO), 20-3
voltage level values, 20-21
voltage regulator, 1-24
voltage regulator control register (VR_CTL), 20-19, 20-27
VR_CTL (voltage regulator control register), 9-39, 20-19, 20-25, 20-27

W

wait states, additional, 6-16
WAKE bit, 20-27

WAKEDET bit, 8-36, 8-95, 8-96
wake from hibernate, MAC, 8-31
wake from sleep, MAC, 8-32
wakeup filter 0 address type bit, 8-89, 8-90
wakeup filter 0 pattern CRC[15:0] field, 8-92
wakeup filter 0 pattern offset[7:0] field, 8-91
wakeup filter 1 address type bit, 8-89, 8-90
wakeup filter 1 pattern CRC[15:0] field, 8-92
wakeup filter 1 pattern offset[7:0] field, 8-91
wakeup filter 2 address type bit, 8-89, 8-90
wakeup filter 2 pattern CRC[15:0] field, 8-92
wakeup filter 2 pattern offset[7:0] field, 8-91
wakeup filter 3 address type bit, 8-89
wakeup filter 3 pattern CRC[15:0] field, 8-92
wakeup filter 3 pattern offset[7:0] field, 8-91
wakeup frame detected, 8-41
wakeup function, 4-10
wakeup interrupt, CAN, 9-25
WAKEUP signal, 20-16
watchdog control register (WDOG_CTL), 17-7, 17-8
watchdog count[15:0] field, 17-6
watchdog count[31:16] field, 17-6
watchdog count register (WDOG_CNT), 17-5, 17-6
watchdog mode, CAN, 9-19
watchdog status[15:0] field, 17-7
watchdog status[31:16] field, 17-7
watchdog status register (WDOG_STAT), 17-6, 17-7
watchdog timer, 1-21, 17-1 to 17-10
  block diagram, 17-2
  disabling, 17-5
  and emulation mode, 17-2
  features, 17-1
  registers, 17-5
  and reset, 17-4
  reset, 19-5
  starting, 17-4
  zero value, 17-5
waveform generation, pulse width modulation, 15-16
WBA bit, 9-43
WDEN[7:0] field, 17-7
WDEV[1:0] field, 17-3, 17-7
WDOG_CNT (watchdog count register), 17-3, 17-5, 17-6
WDOG_CTL (watchdog control register), 17-7, 17-8
WDOG_STAT (watchdog status register), 17-3, 17-6, 17-7
WDSIZE[1:0] field, 5-74, 5-77
WDTH_CAP mode, 15-26, 15-47
  control bit and register usage, 15-50
WLS[1:0] field, 13-21
WNR bit, 5-74, 5-78
WOFF[9:0] field, 12-22, 12-66
WOM bit, 10-17, 10-41
word length
  SPI, 10-18
  SPORT, 12-28
  SPORT receive data, 12-61
  SPORT transmit data, 12-58
work unit
  completion, 5-25
  DMA, 5-17
  interrupt timing, 5-28
  restrictions, 5-27
  transitions, 5-27
WR bit, 9-44

write, asynchronous, 6-14
write access for EBIU asynchronous memory controller, 6-20
write command, 6-35
write complete bit, 18-21
write complete interrupt enable bit, 18-20
write operation, GPIO, 14-11
write pending status bit, 18-21
write with data mask command, 6-51
WSIZE[3:0] field, 12-21, 12-66
WT bit, 9-44
WUIF bit, 9-25, 9-48
WUIM bit, 9-25, 9-47
WUIS bit, 9-25, 9-47

X

X_COUNT[15:0] field, 5-85
XFR_TYPE[1:0] field, 7-5, 7-27, 7-30
X_MODIFY[15:0] field, 5-89
XMTDATA16[15:0] field, 11-45
XMTDATA8[7:0] field, 11-44
XMTFLUSH bit, 11-38, 11-40
XMTINTLEN bit, 11-38, 11-39
XMTSERV bit, 11-18, 11-43
XMTSERVM bit, 11-41
XMTSTAT[1:0] field, 11-17, 11-40

Y

YCbCr format, 7-28
Y_COUNT[15:0] field, 5-90
Y_MODIFY[15:0] field, 5-93

Z

ZEROFILL bit, 19-14